EJERCICIOS, PROBLEMAS, CASOS PRÁCTICOS

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Ejercicios básicos de programación resueltos en Python

Francisco Jesús Martínez Mimbrera
David Díaz Jiménez

20/01/2026

Inteligencia artificial

EJERCICIOS BÁSICOS DE PROGRAMACIÓN RESUELTOS EN PYTHON

Universidad de Jaén
Escuela Politécnica Superior de Jaén

Francisco Jesús Martínez Mimbrera
David Díaz Jiménez

Título: Ejercicios básicos de programación resueltos en Python
Autores: Francisco J. Martínez Mimbrera y David Díaz Jiménez
Versión: 1
Fecha: 20 de enero de 2026

Este manual ha sido desarrollado y maquetado utilizando LaTeX 2_ε.

Índice general

1. Introducción **5**

2. E/S por teclado y pantalla, y operaciones aritméticas **9**

2.1. Mi primer programa - ¡Hola, mundo! 10
2.2. Saludo personalizado 10
2.3. Captura de distintos tipos de información desde teclado y visualización por pantalla 11
2.4. Calcular el área de un cuadrado a partir de un lado 12
2.5. Conversor de pesetas a euros 12
2.6. Suma, Resta, Multiplicación y División de dos números 13
2.7. Precio medio de un producto en función de sus precios en otros establecimientos 14
2.8. Suma de los "n" primeros números naturales consecutivos 15
2.9. Pasar de pulgadas a milímetros 16
2.10. Pasar de grados Celsius a grados Fahrenheit 16
2.11. Calcular estadísticas de suspensos, aprobados, notables y sobresalientes de la clase 17
2.12. Cálculo del precio de venta de un coche a partir del coste de fabricación, ganancia de la empresa e IVA 18
2.13. Calcular el perímetro de una circunferencia a partir del radio 19
2.14. Calcular la raíz cuadrada de un número dado 19
2.15. Calcular la raíz n-ésima de un número 20
2.16. Calcular el área de un triángulo rectángulo 20
2.17. Calcular el área de un triángulo equilátero 21
2.18. Calcular el área de cualquier triángulo a partir de sus lados mediante la fórmula de Herón 21
2.19. Calcular el volumen de un cilindro a partir de su altura y diámetro 22
2.20. Calcular el valor del lado "a" de un triángulo rectángulo dados el valor de la hipotenusa "h" y del lado "b" 23
2.21. Calcular la altura de un triángulo equilátero 24

2.22. Obtener el valor del binomio de suma al cuadrado $(a+b)^2$ 24

3. Flujos de control, funciones, cadenas de caracteres, arrays y matrices 25

3.1. ¿Número par o impar? 26
3.2. Calcular si un número es múltiplo de 3 y a la vez par 26
3.3. Calcular si un año es bisiesto 27
3.4. Programa que calcule el Máximo Común Divisor de dos números enteros no negativos mediante el algoritmo de Euclides 27
3.5. Función para calcular del Máximo Común Divisor de dos números enteros no negativos mediante el algoritmo de Euclides 28
3.6. Función para calcular si un número entero es divisor de otro 29
3.7. Pasar a función el ejercicio propuesto número 2.18 30
3.8. Función para calcular el área de una circunferencia de radio válido 31
3.9. Calcular la suma de los "n" primeros números naturales (Con bucle) 31
3.10. Realizar un programa que sume números introducidos por teclado hasta que se escriba un número concreto de parada 32
3.11. Dados 10 números por teclado decir cuál es el mayor de todos utilizando una función 33
3.12. Calcular el tipo de triángulo en función de los lados 34
3.13. Resolver ecuaciones de segundo grado mediante la fórmula general (sólo soluciones reales) 34
3.14. Dados 10 números por teclado sumar los pares y los impares de manera separada y mostrar ambos resultados 36
3.15. Función para calcular el factorial de un número (iterativo) 37
3.16. Función para calcular el factorial de un número (recursivo) 38
3.17. Función para calcular el número combinatorio de N sobre M 38
3.18. Función para resolver $(a+b)^n$ utilizando el desarrollo del binomio de Newton . . 39
3.19. Implementar la función de Fibonacci para valores enteros positivos 41
3.20. Implementar un procedimiento para seleccionar y visualizar los K bits de orden inferior de un número entero proporcionado 42
3.21. Implementar una función que invierta los últimos K bits de orden inferior de un número entero proporcionado 43
3.22. Invertir una cadena de texto leída por teclado, sin utilizar las funciones del propio lenguaje de programación 44
3.23. Leer una cadena de caracteres desde teclado, decir longitud y concatenar con otra utilizando las funciones nativas de Python 45
3.24. Convertir cadenas a letras mayúsculas y contar sus vocales 45
3.25. Función para realizar la trasposición de una matriz de números enteros dada . . . 47

3.26. Función para multiplicar 2 matrices cuadradas de números enteros 48

4. Manejo de Ficheros y Conexión a base de datos **49**
4.1. Crear y escribir en un fichero 50
4.2. Leer y mostrar contenido de un fichero de texto 50
4.3. Contar caracteres y palabras de un fichero de texto 51
4.4. Guardar una estructura en un fichero binario y cargarla en memoria para ser mostrada por pantalla 52
4.5. Insertar al final de un fichero de texto 53
4.6. Conexión con una base de datos desde Python 54

5. Algoritmos de ordenación y búsqueda **55**
5.1. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo de la **Burbuja** 56
5.2. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo **Quicksort** 57
5.3. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo **TimSort** utilizando el método propio de Python 58
5.4. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo **MergeSort** 58
5.5. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo **Shellsort** 60
5.6. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo **RadixSort** 61
5.7. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo **HeapSort** 62
5.8. Dado un conjunto de números enteros, realizar la búsqueda de un número mediante el algoritmo de **Búsqueda Lineal** 64
5.9. Dado un conjunto de números enteros ordenados, realizar la búsqueda de un número mediante el algoritmo de **Búsqueda Binaria** 65
5.10. Dado un conjunto de números enteros no negativos ordenados, realizar la búsqueda de un número mediante el algoritmo de **Búsqueda por Interpolación** . 66

6. Ejercicios de Criptografía Básica **69**
6.1. Implementar el Cifrado César (Desplazamiento) 70
6.2. Descifrar el Código César con Clave Conocida 71
6.3. Ataque de Fuerza Bruta al Cifrado César 72
6.4. Cifrado y Descifrado mediante Operación XOR 73
6.5. Implementar el Cifrado de Vigenère 74

6.6. Cifrado por Sustitución Monoalfabética con Clave Aleatoria 75
6.7. Cifrado por Transposición Columnar 77
6.8. Análisis de Frecuencias Básico 78
6.9. Generación de un Hash Simple (Checksum) 80
6.10. Generador de Contraseñas Aleatorias Seguras 81

7. Procesamiento básico de información visual con OpenCV 83
7.1. Leer, mostrar e imprimir información sobre una imagen 84
7.2. Convertir una imagen a escala de grises 84
7.3. Rotar una imagen 85
7.4. Invertir colores de una imagen 86
7.5. Ajustar el brillo y contraste de una imagen 87
7.6. Recortar una región de interés (ROI) en una imagen 88
7.7. Reflejar una imagen horizontal o verticalmente 89
7.8. Cargar y mostrar información de metadatos de una imagen 90
7.9. Aplicar un efecto de desenfoque a una imagen 91
7.10. Calcular e imprimir el histograma en escala de grises. 91
7.11. Calcular e imprimir los histogramas de cada canal de color 92
7.12. Detección de bordes con el Operador Sobel 93
7.13. Filtro de Relieve (Emboss) 94
7.14. Filtro Bilateral: Suavizado con preservación de bordes 95
7.15. Eliminación de ruido "Sal y Pimienta"(Filtro de Mediana) 96

8. Introducción al Aprendizaje Automático con Scikit-Learn 97
8.1. Carga y exploración básica de un Dataset 98
8.2. División del conjunto de datos (Train/Test Split) 100
8.3. Preprocesamiento: Estandarización de datos 100
8.4. Regresión Lineal Simple 101
8.5. Clasificación con Árboles de Decisión 102
8.6. Evaluación de modelos: Matriz de Confusión y Accuracy 102
8.7. Aprendizaje No Supervisado: Clustering con K-Means 103
8.8. Validación Cruzada (Cross-Validation) 104
8.9. Preprocesamiento: Codificación de variables categóricas (One-Hot Encoding) . . . 104
8.10. Persistencia del Modelo: Guardar y Cargar 106

Bibliografía 107

Capítulo 1

Introducción

El propósito general de este manual es proveer a los alumnos de las disciplinas de ingeniería informática, telecomunicaciones e industriales de un libro con problemas resueltos básicos de programación en Python. La idea es que dicho material pueda servir como apoyo al estudio de las asignaturas a las que se enfrentarán durante su etapa en la Universidad. El lenguaje de programación Python ha ido evolucionando a lo largo de los años desde su concepción a finales de la década de 1980 y su posterior implementación en diciembre de 1989 por Guido van Rossum [23] como sucesor del lenguaje ABC para el sistema operativo Amoeba [24]. El nombre «Python» no tiene relación con reptiles, sino que fue elegido por la afición de Van Rossum al grupo de comedia británico Monty Python's Flying Circus [23]. Aunque la versión 0.9.0 se publicó en 1991, Python 1.0, lanzada en enero de 1994 [15] marcó un hito fundamental en su desarrollo. Python es gestionado por la Python Software Foundation (PSF) a través de los Python Enhancement Proposals (PEP). En la actualidad, Python 3.14 es la versión estable de referencia (lanzada a finales de 2025) [17]. Este lenguaje es altamente utilizado en ciencia de datos, inteligencia artificial (IA), aprendizaje automático y desarrollo web, destacando por su legibilidad y su filosofía de «baterías incluidas». Según el índice TIOBE (https://www.tiobe.com/tiobe-index/) Python se sitúa como el 1º lenguaje de programación más utilizado en el mundo en el año 2025, manteniendo una posición dominante gracias al auge de la IA generativa y desplazando a lenguajes más tradicionales en la enseñanza y el prototipado rápido. Para la elaboración de este libro se han utilizado las versiones más recientes del lenguaje y de los compiladores.

Estructura del manual

Para alcanzar el propósito planteado, este manual se va a estructurar en los siguientes capítulos para una mejor comprensión por parte del lector.

- **Capítulo 2. E/S por teclado y pantalla, y operaciones aritméticas**. En este capítulo se pretende realizar ejercicios básicos de toma de contacto con el lenguaje y los primeros trabajos con las entradas de información por teclado y las salidas por pantalla. Asimismo, se verá cómo resolver problemas que implican operaciones aritméticas básicas.
- **Capítulo 3. Flujos de control, funciones, cadenas de caracteres, arrays y matrices**. En este capítulo se van a presentar ejercicios relativos al trabajo con estructuras de flujos de control, trabajo con funciones y procedimientos, cadenas de caracteres , arrays y matrices.
- **Capítulo 4. Manejo de Ficheros y Base de datos**. Este capítulo lo dedicaremos a resolver problemas de manejo de ficheros para guardar información y problemas que trabajen con bases de datos.
- **Capítulo 5. Algoritmos de ordenación y búsqueda** . Como parte interesante de la resolución de problemas básicos de programación, siempre está bien el tener conocimiento de algoritmos de ordenación y búsqueda que nos ayuden a resolver problemas simples que sin dichos algoritmos su tiempo de ejecución se volvería inabarcable.
- **Capítulo 6. Ejercicios de criptografía básica**. En este capítulo se incluyen problemas clásicos de criptografía para introducir al alumno en los fundamentos de la materia.
- **Capítulo 7. Procesamiento básico de información visual con OpenCV**. En este capítulo veremos algunos conceptos básicos sobre tratamiento de información visual utilizando la librería OpenCV.
- **Capítulo 8. Introducción al Aprendizaje Automático con Scikit-Learn**. Como cierre del manual, se incluye un capítulo dedicado a problemas básicos de Machine Learning utilizando la librería Scikit-Learn.

Finalmente, el manual concluye con una **Recopilación Bibliográfica** de las fuentes más destacadas para repasar conceptos y sintaxis de programación en Python.

Herramientas de trabajo para programar en Python

Para la elaboración de los problemas de este manual, se ha utilizado el intérprete Python 3.13 [16] y el IDE Visual Studio Code de Microsoft [12] en la versión Debian 13 de Linux. Ambos están disponibles para los tres sistemas operativos más importantes del mercado, Windows, macOS y Linux.

Para ejecutar los problemas que se encuentran en el libro, en primer lugar se debe crear el fichero.py con el problema a resolver y posteriormente lanzar el programa utilizando el interprete python. También se puede descargar directamente el fichero.py del repositorio GitHub habilitado para ello. Esta operación se puede hacer tanto en la terminal utilizando los comandos “python fichero.py” o desde un IDE como Visual Studio Code o PyCharm [6].

Código fuente de los ejemplos

Para que el lector pueda utilizar más cómodamente los ejemplos de este manual, éstos han sido subidos a un repositorio público para que puedan ser descargados. Los ficheros.py están codificados con el juego de caracteres UTF-8, tal y como indican las normas modernas de codificación. En la siguiente dirección se podrán descargar los problemas resueltos.

`https://github.com/franmartinezmimbrera/problemaspython`

Capítulo 2

E/S por teclado y pantalla, y operaciones aritméticas

En este capítulo se van a resolver una serie de problemas utilizando los distintos tipos de datos, operadores y expresiones en los lenguajes propuestos. Para estos problemas, las entradas de datos se realizarán mediante el teclado o con precarga de variables, mientras que las salidas se realizarán por pantalla.

Los ejercicios resueltos que contiene servirán de base para familiarizarse con las operaciones básicas que se suelen realizar en un lenguaje de programación, así como para empezar a utilizar algunas librerías externas, tales como la librería para ciertas operaciones matemáticas.

Como se ha resaltado en el capítulo de introducción, este manual no contiene teoría ni especificaciones sobre el lenguaje de programación, mostrando únicamente soluciones a problemas propuestos. Para revisar la teoría y especificaciones propias del lenguaje para E/S por teclado y pantalla, y operaciones aritméticas, se propone al lector revisar algunos de los libros más destacados en la bibliografía, tales como [17], [9], [18], [11] y [7].

A continuación, se presentan los ejercicios resueltos para este capítulo.

2.1. Mi primer programa - ¡Hola, mundo!

Este es el primer ejercicio/contacto de un programador con cualquier lenguaje de programación. Se debe realizar un programa que muestre "¡Hola, mundo!" por pantalla.

```
# fichero holamundo.py
# Este programa muestra el saludo hola mundo por pantalla.

print ("Hola mundo")
```

2.2. Saludo personalizado

Este ejercicio trata de capturar información desde el teclado (un nombre), para posteriormente almacenarla en una cadena de caracteres y mostrarla por pantalla en forma de saludo personalizado.

En este caso, se ha hecho uso de los f-strings para mostrar el saludo personalizado. Los **f-strings** (abreviatura de Formatted String Literals) son una característica introducida en Python 3.6 que revolucionó la forma de trabajar con textos. Básicamente, permiten incrustar expresiones de código Python directamente dentro de una cadena de texto, haciéndola más legible y rápida. Asimismo, se puede seguir haciendo de la manera tradicional `print("¿Qué tal estás?, " + nombre)`.

```
# fichero saludo.py
# Este programa hace un saludo personalizado

nombre = input("¡Hola! ¿Cómo te llamas? ")
print(f"¿Qué tal estás, {nombre}?")
```

2.3. Captura de distintos tipos de información desde teclado y visualización por pantalla

Este ejercicio trata de capturar distintos tipos de información desde el teclado, para posteriormente almacenarla en su variable correspondiente y mostrar dicha información por pantalla.

```
# fichero capturar.py
# Este programa recopila datos del usuario de forma segura

try:

    nombre = input("Introduce tu nombre (texto): ")
    edad = int(input("Introduce tu edad (entero): "))
    altura = float(input("Introduce tu altura en metros (decimal con
       punto): "))

    print(f"Nombre: {nombre}")
    print(f"Edad: {edad} años")
    print(f"Altura: {altura} m")

except ValueError:
    print("\n[!] ERROR CRÍTICO:")
    print("Has introducido un texto donde se esperaba un número.")
    print("Por favor, reinicia el programa e introduce datos válidos
       (ej: edad 25, altura 1.75).")
```

2.4. Calcular el área de un cuadrado a partir de un lado

Este ejercicio calcula el área de un cuadrado a partir de un lado dado por teclado y muestra el resultado por pantalla. La fórmula del área de un cuadrado [5] viene dada por $A = l^2$.

```
# fichero areacuadrado.py
# Programa que calcula el área de un cuadrado a partir de un lado

try:
    lado = float(input("Introduce el valor del lado del cuadrado: "))
    area = lado ** 2
    print(f"El área del cuadrado es: {area}")

except ValueError:
    print("Error: Entrada no válida. Por favor, introduce un número
       (ej: 5 o 2.5).")
```

2.5. Conversor de pesetas a euros

Este ejercicio propone crear un conversor de pesetas a euros. Partiendo de un número dado por teclado, se calculará su equivalencia a euros y se mostrará su cantidad expresada en euros por pantalla. El cambio de euros a pesetas está fijado a razón de 1€ = 166,386 pesetas.

```
# fichero pesetaseuros.py
# Este programa realiza la conversión de pesetas a euros

FACTOR_CONVERSION = 166.386

try:
    pesetas = float(input("Introduce la cantidad en pesetas: "))

    euros = pesetas / FACTOR_CONVERSION

    print(f"{pesetas} pesetas equivalen a {euros:.2f} euros.")

except ValueError:
    print("Error: Entrada no válida. Por favor, introduce un valor
       numérico.")
```

2.6. Suma, Resta, Multiplicación y División de dos números

Este ejercicio propone, dados dos números por teclado, realizar todas las operaciones aritméticas básicas sobre ellos y mostrar los resultados por pantalla.

```
# fichero operacionesaritmeticas.py
# Este programa realiza operaciones aritméticas dados 2 números

try:

    n1 = float(input("Introduce el primer número: "))
    n2 = float(input("Introduce el segundo número: "))

    print(f"Suma:           {n1} + {n2} = {n1 + n2}")
    print(f"Resta:          {n1} - {n2} = {n1 - n2}")
    print(f"Multiplicación: {n1} * {n2} = {n1 * n2}")

    if n2 != 0:
        print(f"División:       {n1} / {n2} = {n1 / n2}")
    else:
        print("División:       No se puede dividir por cero.")

except ValueError:
    print("Error: Debes introducir números válidos (ej: 10, 5.5).")
```

2.7. Precio medio de un producto en función de sus precios en otros establecimientos

Este ejercicio propone, dados por teclado los precios de un producto en 3 establecimientos, calcular el precio medio del producto y mostrarlo por pantalla.

```
# fichero preciomedio.py
# Este programa calcula el precio medio de un producto a partir de 3
    precios

try:
    print("Por favor, introduce el precio del producto en 3
        establecimientos distintos:")

    p1 = float(input("Precio en establecimiento 1: "))
    p2 = float(input("Precio en establecimiento 2: "))
    p3 = float(input("Precio en establecimiento 3: "))


    media = (p1 + p2 + p3) / 3

    print(f"\nEl precio medio del producto es: {media:.2f} €")

except ValueError:
    print("Error: Entrada inválida. Asegúrate de introducir números
        (ej: 12.50).")
```

2.8. Suma de los “n” primeros números naturales consecutivos

Este ejercicio propone, dado un número “n” por teclado, calcular los n primeros números naturales consecutivos y mostrar el resultado por pantalla.

Para ello, se utilizará la fórmula de las Series Aritméticas Notables de números naturales [5], conocida como la fórmula de Gauss:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + ... + n = \frac{n \cdot (n+1)}{2}$$

```
# fichero sumannumerosnaturales.py
# Este programa calcula la suma de los "n" primeros números naturales

try:

    n = int(input("Introduce un número entero 'n': "))

    if n < 0:
        print("Por favor, introduce un número entero positivo.")
    else:
        # Aplicamos la fórmula de Gauss: n * (n + 1) / 2
        # Usamos // para asegurar que la división es entera
        suma = (n * (n + 1)) // 2

        print(f"La suma de los primeros {n} números naturales es:
            {suma}")

except ValueError:
    print("Error: Debes introducir un número entero válido.")
```

2.9. Pasar de pulgadas a milímetros

Este ejercicio propone que, dado un valor numérico expresado en pulgadas por teclado, este se convierta a milímetros y el resultado sea mostrado por pantalla. Para ello se tendrá en cuenta que 1 pulgada = 25.4 mm.

```
# fichero pulgadasmilimetros.py
# Este programa cambia pulgadas por milímetros
FACTOR = 25.4
try:
    pulgadas = float(input("Introduce el valor en pulgadas: "))
    milimetros = pulgadas * FACTOR
    print(f"{pulgadas} pulgadas son {milimetros} milímetros.")

except ValueError:
    print("Error: Entrada no válida. Por favor, introduce un número.")
```

2.10. Pasar de grados Celsius a grados Fahrenheit

Este ejercicio propone que, dado por teclado un valor numérico expresado en grados Celsius ($^\circ C$) (históricamente conocidos como centígrados), este sea pasado a grados Fahrenheit ($^\circ F$) y mostrado por pantalla. Para ello se tendrá en cuenta que:

$$^\circ F = \frac{9}{5} \cdot ^\circ C + 32$$

```
# fichero cambiogrados.py
# Este programa cambia grados centígrados por Fahrenheit

try:
    celsius = float(input("Introduce grados Celsius: "))
    fahrenheit = (9/5 * celsius) + 32
    print(f"{celsius} ºC equivalen a {fahrenheit:.2f} ºF")

except ValueError:
    print("Error: Entrada no válida. Por favor, introduce un número.")
```

2.11. Calcular estadísticas de suspensos, aprobados, notables y sobresalientes de la clase

Este ejercicio propone, dado por teclado el número de suspensos, aprobados, notables y sobresalientes de una clase, calcular:

- El porcentaje de alumnos que superan la asignatura.
- El porcentaje de alumnos que suspenden la asignatura.
- El porcentaje de alumnos que sacan notable en la asignatura.
- El porcentaje de alumnos que sacan sobresaliente en la asignatura.
- El porcentaje de alumnos que han sacado un aprobado en la asignatura.

```
# fichero estadisticas.py
# Este programa calcula estadísticas sobre alumnos

try:
    print("Introduce el número de alumnos por categoría:")
    suspensos = int(input("Suspensos: "))
    aprobados = int(input("Aprobados (5-6): "))
    notables = int(input("Notables (7-8): "))
    sobresalientes = int(input("Sobresalientes (9-10): "))
    total_alumnos = suspensos + aprobados + notables + sobresalientes

    # Verificamos si hay alumnos para evitar dividir por cero
    if total_alumnos == 0:
        print("\nNo hay alumnos en la clase. No se pueden calcular
            estadísticas.")
    else:
        superan = aprobados + notables + sobresalientes
        porc_superan = (superan / total_alumnos) * 100
        porc_suspensos = (suspensos / total_alumnos) * 100
        porc_notables = (notables / total_alumnos) * 100
        porc_sobresalientes = (sobresalientes / total_alumnos) * 100
        porc_aprobados_strict = (aprobados / total_alumnos) * 100

        print(f"\nTotal alumnos: {total_alumnos}")
        print(f"% que superan la asignatura: {porc_superan:.2f}%")
```

```
        print(f"% de suspensos:              {porc_suspensos:.2f}%")
        print(f"% de notables:               {porc_notables:.2f}%")
        print(f"% de sobresalientes:
            {porc_sobresalientes:.2f}%")
        print(f"% de aprobados (nota 5-6):
            {porc_aprobados_strict:.2f}%")

except ValueError:
    print("Error: Debes introducir números enteros válidos.")
```

2.12. Cálculo del precio de venta de un coche a partir del coste de fabricación, ganancia de la empresa e IVA

Este ejercicio propone, dado por teclado un valor numérico que expresa el coste de fabricar un coche, calcular el precio total de venta al que tendría que venderse, si se desea una ganancia fija de un 15 % y el valor del IVA del coche es del 21 %.

```
# fichero costecoche.py
# Este programa calcula el precio total de un coche
GANANCIA = 0.15 # 15%
IVA = 0.21      # 21%

try:
    coste_fabricacion = float(input("Introduce el coste de
        fabricación: "))
    # Calculamos el precio base con la ganancia
    precio_con_ganancia = coste_fabricacion * (1 + GANANCIA)
    # Aplicamos el IVA al precio anterior
    precio_final = precio_con_ganancia * (1 + IVA)

    print(f"\nCoste fabricación: {coste_fabricacion:.2f} €")
    print(f"Precio venta final (con 15% ganancia y 21% IVA):
        {precio_final:.2f} €")

except ValueError:
    print("Error: Debes introducir un valor numérico para el coste.")
```

2.13. Calcular el perímetro de una circunferencia a partir del radio

Este ejercicio propone, dado por teclado el radio de una circunferencia, calcular el perímetro de la misma y mostrarlo por pantalla. Para ello se tendrá en cuenta que el perímetro de una circunferencia [5] viene dado por la siguiente fórmula $P = 2\pi r$.

```
# fichero perimetrocir.py
# Este programa calcula el perímetro de una circunferencia
import math # Librería matemática para usar el valor de PI
try:
    radio = float(input("Introduce el radio de la circunferencia: "))
    perimetro = 2 * math.pi * radio
    print(f"El perímetro de la circunferencia es: {perimetro:.4f}")

except ValueError:
    # Gestión de errores
    print("Error: Entrada inválida. Por favor, introduce un número.")
```

2.14. Calcular la raíz cuadrada de un número dado

Este ejercicio propone, dado por teclado un número, calcular su raíz cuadrada y mostrarla por pantalla. Para ello se utilizará la función `sqrt` que proporciona la librería `math` de Python.

```
# fichero raizcuadrada.py
# Este programa calcula la raíz cuadrada de un número
import math
try:
    numero = float(input("Introduce un número: "))
    if numero < 0:
        print("Error: No se puede calcular la raíz cuadrada real de un
            número negativo.")
    else:
        raiz = math.sqrt(numero)
        print(f"La raíz cuadrada de {numero} es {raiz:.4f}")
except ValueError:
    print("Error: Entrada inválida. Por favor, introduce un número.")
```

2.15. Calcular la raíz n-ésima de un número

Este ejercicio propone, dado por teclado un número entero y un exponente, calcular la raíz n-ésima de dicho número y mostrarla por pantalla. Para ello, se debe tener en cuenta que la raíz n-ésima de un número $\sqrt[n]{a}$ se puede reescribir como $a^{\frac{1}{n}}$ [5].

```
# fichero raizenesima.py
# Este programa calcula la raíz n-ésima de un número
try:
    base = float(input("Introduce el número (base): "))
    n = float(input("Introduce el índice de la raíz (n): "))
    if n == 0:
        print("Error: No existe la raíz 0-ésima.")
    else:
        resultado = base ** (1/n)
        print(f"La raíz {n}-ésima de {base} es: {resultado:.4f}")
except ValueError:
    print("Error: Entrada no válida. Asegúrate de introducir números.")
```

2.16. Calcular el área de un triángulo rectángulo

Este ejercicio propone, dados la base de un triángulo y la altura por teclado, calcular el área del mismo y mostrarla por pantalla.

Se utilizará la fórmula del área de un triángulo rectángulo [25]: $A = \frac{b \cdot a}{2}$

```
# fichero areatriangulo1.py
# Calcula el área de un triángulo rectángulo a partir de base y altura
try:
    base = float(input("Introduce la base: "))
    altura = float(input("Introduce la altura: "))
    area = (base * altura) / 2
    print(f"El área del triángulo rectángulo es: {area}")
except ValueError:
    print("Error: Entrada no válida. Por favor, introduce números.")
```

2.17. Calcular el área de un triángulo equilátero

Este ejercicio propone, dado un lado de un triángulo equilátero por teclado, calcular el área del mismo y mostrarla por pantalla. Utilizaremos la fórmula del área de un triángulo equilátero [25]: $A = \frac{\sqrt{3}}{4} \cdot a^2$

```
# fichero areatriangulo2.py
# Calcula el área de un triángulo equilátero a partir de uno de sus
    lados

import math

try:

    lado = float(input("Introduce el lado del triángulo equilátero: "))
    area = (math.sqrt(3) / 4) * (lado ** 2)

    print(f"El área del triángulo equilátero es: {area:.4f}")

except ValueError:
    print("Error: Entrada no válida. Por favor, introduce un número.")
```

2.18. Calcular el área de cualquier triángulo a partir de sus lados mediante la fórmula de Herón

Este ejercicio propone, dados por teclado los valores de los lados de un triángulo, calcular su área y mostrarla por pantalla utilizando la fórmula de Herón. El área de un triángulo según la fórmula de Herón [25] viene dada por:

$$A = \sqrt{sp \cdot (sp - l1) \cdot (sp - l2) \cdot (sp - l3)}$$

donde

$$sp = \frac{(l1 + l2 + l3)}{2}$$

```python
# fichero areatriangulo.py
# Este programa calcula el área de un triángulo a partir de sus lados
    mediante la fórmula de Herón
import math

try:
    print("Por favor, introduce los 3 lados del triángulo:")
    l1 = float(input("Lado 1: "))
    l2 = float(input("Lado 2: "))
    l3 = float(input("Lado 3: "))

    # Calcular el semiperímetro (s)
    sp = (l1 + l2 + l3) / 2

    # Calcular el radicando de la fórmula de Herón: s(s-a)(s-b)(s-c)
    radicando = sp * (sp - l1) * (sp - l2) * (sp - l3)

    if radicando < 0:
        print("Error: Los lados introducidos no forman un triángulo
            válido (desigualdad triangular).")
    else:
        area = math.sqrt(radicando)
        print(f"El área del triángulo es: {area:.4f}")

except ValueError:
    print("Error: Entrada no válida. Por favor, introduce números.")
```

2.19. Calcular el volumen de un cilindro a partir de su altura y diámetro

Este ejercicio propone, dados por teclado el diámetro y la altura de un cilindro, calcular el volumen del mismo y mostrarlo por pantalla. Para ello se tendrá en cuenta que $V = \pi \cdot r^2 \cdot H$ [5], siendo r el radio de la base y H la altura del cilindro. El radio se obtiene a partir del diámetro tal que $r = d/2$.

```
# fichero volumencilindro.py
# Este programa calcula el volumen de un cilindro
import math
try:
    diametro = float(input("Introduce el diámetro del cilindro: "))
    altura = float(input("Introduce la altura del cilindro: "))
    # Calcular radio (diámetro / 2)
    radio = diametro / 2
    # Calcular volumen (pi * r^2 * h)
    volumen = math.pi * (radio ** 2) * altura
    print(f"El volumen del cilindro es: {volumen:.4f}")
except ValueError:
    print("Error: Entrada no válida. Por favor, introduce números.")
```

2.20. Calcular el valor del lado “a” de un triángulo rectángulo dados el valor de la hipotenusa “h” y del lado “b”

Este ejercicio propone, dados por teclado el valor del lado “b” de un triángulo rectángulo y el valor de su hipotenusa “h”, calcular el valor del lado “a” y mostrarlo por pantalla. Para ello se utilizará el Teorema de Pitágoras $h^2 = a^2 + b^2$ [25], de donde, al despejar, se obtiene que $a = \sqrt{h^2 - b^2}$.

```
# fichero ladotriangulo.py
# Este programa calcula el valor del cateto 'a' de un triángulo
   rectángulo dados la hipotenusa 'h' y el otro cateto 'b'
import math
try:
    h = float(input("Introduce la hipotenusa (h): "))
    b = float(input("Introduce el otro cateto (b): "))
    if h <= b:
        print("Error: La hipotenusa debe ser > que el cateto.")
    else:
        a = math.sqrt(h**2 - b**2)
        print(f"El valor del cateto 'a' es: {a:.4f}")
except ValueError:
    print("Error: Entrada no válida. Por favor, introduce números.")
```

2.21. Calcular la altura de un triángulo equilátero

Este ejercicio propone, dado por teclado el lado de un triángulo equilátero, calcular la altura h del mismo y mostrar el resultado por pantalla. Para calcular la altura de un triángulo equilátero se aplicará el Teorema de Pitágoras, obteniendo que $h = \frac{\sqrt{3}a}{2}$ [25].

```
# fichero alturatriangulo.py
# Este programa calcula la altura de un triángulo equilátero
import math
try:
    lado = float(input("Introduce el lado del triángulo equilátero: "))

    if lado <= 0:
        print("Error: El lado debe ser mayor que 0.")
    else:
        altura = (math.sqrt(3) * lado) / 2
        print(f"La altura del triángulo es: {altura:.4f}")
except ValueError:
    print("Error: Entrada no válida. Por favor, introduce un número.")
```

2.22. Obtener el valor del binomio de suma al cuadrado $(a+b)^2$

Este ejercicio propone, dados por teclado dos números, calcular el binomio de suma al cuadrado $(a+b)^2$ [5] y mostrar el resultado por pantalla.

```
# fichero binomiosuma.py
# Este programa calcula el binomio de suma de (a + b) al cuadrado
try:
    a = float(input("Introduce el valor de a: "))
    b = float(input("Introduce el valor de b: "))
    # Fórmula (a^2 + b^2 + 2ab)
    resultado = (a ** 2) + (b ** 2) + (2 * a * b)
    print(f"El resultado de ({a} + {b})^2 es: {resultado}")
except ValueError:
    print("Error: Entrada no válida. Por favor, introduce números.")
```

Capítulo 3

Flujos de control, funciones, cadenas de caracteres, arrays y matrices

Este capítulo presenta una serie de problemas resueltos que ilustran distintos tipos de flujos de control. También se incluyen ejercicios sobre cómo resolver problemas utilizando procedimientos o funciones, así como su integración en el programa principal. Por último, se abordarán problemas que implican el trabajo con cadenas de caracteres, arrays y matrices. Para revisar la teoría y especificaciones propias del lenguaje para abordar flujos de control, funciones, cadenas de caracteres, arrays y matrices, se propone al lector revisar algunos de los libros más destacados en la bibliografía tales como [9], [18] y [7].

A continuación, se presentan los ejercicios resueltos para este capítulo.

3.1. ¿Número par o impar?

Este ejercicio propone, dado un número entero, calcular si éste es par o impar y mostrarlo por pantalla. Para ello se utilizará el flujo de control IF - ELSE.

```
# fichero parimpar.py
# Programa que verifica si un número es par o impar
try:
    numero = int(input("Introduce un número entero: "))

    if numero % 2 == 0:
        print(f"El número {numero} es PAR.")
    else:
        print(f"El número {numero} es IMPAR.")

except ValueError:
    print("Error: Debes introducir un número entero válido.")
```

3.2. Calcular si un número es múltiplo de 3 y a la vez par

Este ejercicio propone, dado un número, calcular si es múltiplo de 3 y a la vez par y mostrarlo por pantalla.

```
# fichero multiplo3.py
# Programa que verifica si un número es múltiplo de 3 y PAR a la vez
try:
    numero = int(input("Introduce un número entero: "))
    # Múltiplo de 3 AND Múltiplo de 2 (Par)
    # Matemáticamente, esto equivale a ser múltiplo de 6
    if (numero % 3 == 0) and (numero % 2 == 0):
        print(f"El número {numero} es múltiplo de 3 y PAR.")
    else:
        print(f"El número {numero} NO cumple ambas condiciones.")
except ValueError:
    print("Error: Entrada no válida. Por favor, introduce un número
       entero.")
```

3.3. Calcular si un año es bisiesto

Este ejercicio propone, dado un año, calcular si es bisiesto o no y mostrarlo por pantalla. **Nota:** Un año es bisiesto en el calendario Gregoriano si es divisible entre 4, excepto aquellos divisibles entre 100 pero no entre 400.

```
# fichero bisiesto.py
# Programa que comprueba si un año es bisiesto
try:
    anio = int(input("Introduce un año: "))
    # Condición de bisiesto:
    # Divisible entre 4 Y NO divisible entre 100
    # O
    # Divisible entre 400
    if (anio % 4 == 0 and anio % 100 != 0) or (anio % 400 == 0):
        print(f"El año {anio} ES bisiesto.")
    else:
        print(f"El año {anio} NO es bisiesto.")
except ValueError:
    print("Error: Entrada no válida. Por favor, introduce un año
       (número entero).")
```

3.4. Programa que calcule el Máximo Común Divisor de dos números enteros no negativos mediante el algoritmo de Euclides

Este ejercicio propone, dados dos números enteros no negativos, calcular el máximo común divisor de dichos números utilizando el algoritmo de Euclides (más eficiente) y mostrarlo por pantalla. El **Algoritmo de Euclides** es un método eficiente para encontrar el Máximo Común Divisor (**MCD**) de dos números enteros no negativos a y b [5]. Se basa en el principio de que el MCD de dos números también divide a su diferencia, lo que permite reducir el problema a números más pequeños. El algoritmo se basa en la siguiente propiedad [5]: Si a y b son dos números enteros positivos con $a > b$, entonces $\text{MCD}(a, b) = \text{MCD}(b, r)$, donde r es el resto de la división euclídea de a entre b. Es decir: $\text{MCD}(a, b) = \text{MCD}(b, a \bmod b)$

```python
# fichero MCD.py
# Programa que calcula el Máximo Común Divisor (Algoritmo de Euclides)

try:
    a = int(input("Introduce el primer número: "))

    b = int(input("Introduce el segundo número: "))

    if a < 0 or b < 0:
        print("Error: Por favor, introduce números enteros positivos.")
    else:
        original_a, original_b = a, b

        # Algoritmo de Euclides
        # Se repite hasta que el resto (b) sea 0
        while b > 0:
            a, b = b, a % b

        # Cuando b es 0, 'a' contiene el MCD
        print(f"El MCD de {original_a} y {original_b} es: {a}")

except ValueError:

    print("Error: Entrada no válida. Introduce números enteros.")
```

3.5. Función para calcular del Máximo Común Divisor de dos números enteros no negativos mediante el algoritmo de Euclides

Este ejercicio propone crear una función para, dados dos números enteros no negativos, calcular el máximo común divisor de dichos números utilizando el algoritmo de Euclides (visto en el ejercicio anterior) y que devuelva el resultado. Para ello, se convertirá el ejercicio anterior (3.4) a una función y se mostrará el resultado.

```
#fichero MCD1.py
def mcd_euclides(a, b):
    while b > 0:
        a, b = b, a % b
    return a
if __name__ == "__main__":
    try:
        n1 = int(input("Número 1: "))
        n2 = int(input("Número 2: "))
        resultado = mcd_euclides(n1, n2)
        print(f"El MCD calculado por función es: {resultado}")
    except ValueError:
        print("Error en la entrada de datos.")
```

3.6. Función para calcular si un número entero es divisor de otro

Este ejercicio propone realizar una función que calcule si a es divisor de b. Un número entero d es divisor de otro número entero N si, al dividir N por d, el resto de la división es cero [5]. Para ello, se utilizará el operador módulo `%` de Python, el cual devuelve el resto de la división entre 2 números.

```
# fichero divisor.py
def es_divisor(n, d):
    """Devuelve True si d es divisor de n, False en caso contrario."""
    if d == 0:
        return False # Evitar división por cero
    return n % d == 0

if __name__ == "__main__":
    num = int(input("Introduce el número (N): "))
    div = int(input("Introduce el posible divisor (d): "))
    if es_divisor(num, div):
        print(f"{div} ES divisor de {num}.")
    else:
        print(f"{div} NO es divisor de {num}.")
```

3.7. Pasar a función el ejercicio propuesto número 2.18

Este ejercicio toma como base el ejercicio 2.18 para crear una función que devuelva el valor del área de un triángulo usando la fórmula de Herón.

```
# fichero areatriangulo2.py
import math

def area_heron(a, b, c):
    # Semiperímetro
    s = (a + b + c) / 2
    # Fórmula de Herón: raiz(s * (s-a) * (s-b) * (s-c))
    try:
        area = math.sqrt(s * (s - a) * (s - b) * (s - c))
        return area
    except ValueError:
        return -1 # Retornamos -1 si los lados no forman un triángulo
            válido

if __name__ == "__main__":
    print("Introduce los lados del triángulo:")
    l1 = float(input("Lado A: "))
    l2 = float(input("Lado B: "))
    l3 = float(input("Lado C: "))

    resultado = area_heron(l1, l2, l3)

    if resultado != -1:
        print(f"El área del triángulo es: {resultado:.2f}")
    else:
        print("Los lados proporcionados no forman un triángulo
            válido.")
```

3.8. Función para calcular el área de una circunferencia de radio válido

Este ejercicio propone calcular el área de una circunferencia. Al igual que en el ejercicio anterior, en primer lugar aparece la función que se ha desarrollado y seguidamente un ejemplo de su uso.

```
# fichero areacircunferencia.py
import math
def area_circunferencia(radio):
    if radio < 0:
        return None
    return math.pi * (radio ** 2)
if __name__ == "__main__":
    r = float(input("Introduce el radio: "))
    area = area_circunferencia(r)
    if area is not None:
        print(f"El Área de la circunferencia es: {area:.4f}")
    else:
        print("El radio no puede ser negativo.")
```

3.9. Calcular la suma de los “n” primeros números naturales (Con bucle)

Este ejercicio propone realizar la suma de los “n” primeros números naturales y mostrar el resultado de la suma por pantalla. Para ello, se ha utilizado un bucle **FOR** para practicar con los mismos. Como se puede observar, es mucho menos eficiente que el ejercicio planteado en 2.8.

```
# fichero sumanumerosnaturales.py
def suma_n_naturales():
    n = int(input("Introduce N (cantidad de números a sumar): "))
    suma = 0
    for i in range(1, n + 1): #Suma n primeros números N con for
        suma += i
    print(f"La suma de los primeros {n} números es: {suma}")
if __name__ == "__main__":
    suma_n_naturales()
```

3.10. Realizar un programa que sume números introducidos por teclado hasta que se escriba un número concreto de parada

Este ejercicio propone realizar la suma de números introducidos por teclado, hasta que se introduzca un número de parada (en este caso el número -50) y muestre el resultado de la suma de dichos números por pantalla.

```
# fichero sumanumeros.py
#  Este programa calcula la suma de los números introducidos por
   teclado mientras no se introduzca el número -50

def sumar_hasta_parada():

    suma_total = 0
    numero = 0
    NUMERO_PARADA = -50

    print(f"Introduce números para sumar. Escribe {NUMERO_PARADA} para
       terminar.")

    while True:
        try:
            numero = int(input("Introduce número: "))
            if numero == NUMERO_PARADA:
                break # Salimos del bucle

            suma_total += numero
        except ValueError:
            print("Por favor, introduce un número válido.")

    print(f"La suma total acumulada es: {suma_total}")

if __name__ == "__main__":
    sumar_hasta_parada()
```

3.11. Dados 10 números por teclado decir cuál es el mayor de todos utilizando una función

Este ejercicio propone, dados 10 números introducidos por teclado, calcular cuál es el mayor de todos los introducidos. Para ello, se resolverá guardando los datos introducidos en un array de enteros y posteriormente aplicando una función, la cual debe decir cuál es el número mayor introducido. Finalmente, se mostrará dicho número. Obviamente este ejercicio se puede resolver de una manera mucho más sencilla, pero se pretende practicar el uso de funciones y de arrays.

```
# fichero mayor.py
# Este programa calcula cuál es el número mayor de 10 introducidos por
    teclado

def encontrar_mayor(lista_numeros):
    if not lista_numeros:
        return None

    mayor = lista_numeros[0]
    for num in lista_numeros:
        if num > mayor:
            mayor = num
    return mayor

if __name__ == "__main__":
    numeros = []
    print("Por favor, introduce 10 números enteros:")

    for i in range(10):
        val = int(input(f"Número {i+1}: "))
        numeros.append(val)

    el_mayor = encontrar_mayor(numeros)
    print(f"El número mayor introducido es: {el_mayor}")
```

3.12. Calcular el tipo de triángulo en función de los lados

Este ejercicio propone, dados los lados de un triángulo, calcular el tipo de triángulo que es y mostrarlo por pantalla.

```
# fichero tipotriangulo.py
# Este programa calcula el tipo de triángulo en función de los lados
def tipo_triangulo():
    print("Introduce los tres lados del triángulo:")
    a = float(input("Lado A: "))
    b = float(input("Lado B: "))
    c = float(input("Lado C: "))

    if a == b and b == c:
        print("El triángulo es EQUILÁTERO.")
    elif a == b or a == c or b == c:
        print("El triángulo es ISÓSCELES.")
    else:
        print("El triángulo es ESCALENO.")

if __name__ == "__main__":
    tipo_triangulo()
```

3.13. Resolver ecuaciones de segundo grado mediante la fórmula general (sólo soluciones reales)

Este ejercicio propone resolver ecuaciones de segundo grado, utilizando para ello la fórmula general $ax^2 + bx + c = 0$ [5], donde siempre $a \neq 0$. Para una ecuación de segundo grado con coeficientes reales, existen siempre dos soluciones, no necesariamente distintas, llamadas raíces. A continuación, se presenta la fórmula general para obtener las raíces reales [5]:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

3.13. Resolver ecuaciones de segundo grado mediante la fórmula general (sólo soluciones reales)

```python
# fichero ecuaciones.py

import math

def resolver_ecuacion_segundo_grado():

    print("Resolución de ax^2 + bx + c = 0")
    a = float(input("Introduce a: "))
    b = float(input("Introduce b: "))
    c = float(input("Introduce c: "))

    if a == 0:
        print("Si a=0 no es una ecuación de segundo grado.")
        return

    # Calculamos el discriminante
    discriminante = b**2 - 4*a*c

    if discriminante > 0:
        x1 = (-b + math.sqrt(discriminante)) / (2*a)
        x2 = (-b - math.sqrt(discriminante)) / (2*a)
        print(f"Dos soluciones reales: x1 = {x1:.2f}, x2 = {x2:.2f}")
    elif discriminante == 0:
        x = -b / (2*a)
        print(f"Una solución real doble: x = {x:.2f}")
    else:
        print("No existen soluciones reales (soluciones complejas).")

if __name__ == "__main__":
    resolver_ecuacion_segundo_grado()
```

3.14. Dados 10 números por teclado sumar los pares y los impares de manera separada y mostrar ambos resultados

Este ejercicio requiere diseñar un programa que lea una serie de números enteros introducidos por el usuario y los almacene en un array. Posteriormente, el programa deberá sumar por separado los números pares y los números impares presentes en el array, y finalmente, mostrar ambos resultados (la suma total de pares y la suma total de impares) por pantalla. Aquí se enseñará por primera vez la forma de trabajar con bucles `for` en Python para estructuras de datos iterables como un array (lista).

```
#fichero sumaparesimpares.py
#Calcula la suma de 10 números pasados por teclado distinguiendo entre
    los pares e impares
def sumar_pares_impares():
    numeros = []
    print("Introduce 10 números:")

    for i in range(10):
        val = int(input(f"Dato {i+1}: "))
        numeros.append(val)

    suma_pares = 0
    suma_impares = 0

    for num in numeros:
        if num % 2 == 0:
            suma_pares += num
        else:
            suma_impares += num

    print(f"Suma de PARES: {suma_pares}")
    print(f"Suma de IMPARES: {suma_impares}")

if __name__ == "__main__":
    sumar_pares_impares()
```

3.15. Función para calcular el factorial de un número (iterativo)

Este ejercicio presenta la resolución del cálculo del factorial de un número de manera iterativa. Se ha de tener en cuenta el tipo de entero con el que se esté trabajando, ya que el número factorial crece de manera muy rápida.

Nota: En lenguajes con enteros de tamaño fijo (como C, C++ o Java), el factorial $n!$ puede causar desbordamiento porque el valor máximo representable es limitado. Por ejemplo, un `int` de 32 bits puede almacenar hasta $\approx 2,1 \cdot 10^9$, por lo que $12!$ cabe, pero $13!$ ya lo desborda [1] [22]. Sin embargo, en Python esto no ocurre, ya que los enteros (`int`) tienen precisión arbitraria. Python puede representar números factoriales muy grandes, limitados únicamente por la memoria disponible [7].

```python
# fichero factorialI.py
#Calcula el factorial de un número entero no negativo de forma
    iterativa

def factorial_iterativo(n):
    if n < 0:
        return None # No existe factorial de negativos
    if n == 0:
        return 1

    resultado = 1
    # Multiplicamos desde 1 hasta n
    for i in range(1, n + 1):
        resultado *= i

    return resultado

if __name__ == "__main__":
    num = int(input("Introduce un número para calcular su factorial:
        "))
    res = factorial_iterativo(num)
    if res is not None:
        print(f"{num}! = {res}")
    else:
        print("El número no puede ser negativo.")
```

3.16. Función para calcular el factorial de un número (recursivo)

Este ejercicio presenta la resolución del factorial de forma recursiva. Anteriormente se ha visto la resolución mediante factorial iterativo 3.15. Aunque ambas versiones producen el mismo resultado, la implementación recursiva tiene una limitación importante en Python: *la profundidad máxima de recursión*. Python no optimiza la recursión de cola (*tail recursion*) y por tanto, para valores grandes de n, la versión recursiva producirá un `RecursionError` mucho antes de cualquier límite numérico. La versión iterativa es generalmente preferida en Python porque:

- Es más eficiente: evita la sobrecarga asociada a crear múltiples *frames* de recursión.
- Es más segura: no puede provocar desbordamiento de pila (*stack overflow*), ya que solo utiliza un único marco de ejecución.

```python
#fichero factorialR.py
#Calcula el factorial de un número entero no negativo de forma
    recursiva.

def factorial_recursivo(n):
    if n < 0:
        return None
    # Caso base
    if n == 0 or n == 1:
        return 1
    # Llamada recursiva
    return n * factorial_recursivo(n - 1)

if __name__ == "__main__":
    num = int(input("Introduce un número (recursivo): "))
    print(f"Factorial: {factorial_recursivo(num)}")
```

3.17. Función para calcular el número combinatorio de N sobre M

Este ejercicio propone realizar el cálculo de un número combinatorio dados los valores N y M por teclado. Para realizar el cálculo de un número combinatorio de N sobre M, se utilizará la conocida como fórmula del coeficiente binomial $\binom{N}{M} = \frac{N!}{M!(N-M)!}$ [5], la cual usa la función realizada en 3.15 del factorial para obtener los resultados.

```python
# fichero combinatorio.py
# Calcula el número combinatorio C(n, r) o n sobre r, usando
    factoriales.
# Reutilizamos la lógica del factorial
def factorial(n):
    r = 1
    for i in range(1, n + 1):
        r *= i
    return r

def combinatorio(n, m):
    if m > n:
        return 0
    # Formula: N! / (M! * (N-M)!)
    return factorial(n) // (factorial(m) * factorial(n - m))

if __name__ == "__main__":
    try:
        n = int(input("Introduce N: "))
        m = int(input("Introduce M: "))
        print(f"El combinatorio de {n} sobre {m} es: {combinatorio(n,
            m)}")
    except ValueError:
        print("Entrada inválida.")
```

3.18. Función para resolver $(a+b)^n$ utilizando el desarrollo del binomio de Newton

Este ejercicio propone resolver $(a+b)^n$, dados los valores a, b y n. Para ello se pide utilizar la fórmula del **binomio de Newton** [5]. La fórmula empleada para la resolución viene dada por [5]: $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$ donde $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Esta función utiliza el cálculo combinatorio visto en 3.17. Cabe mencionar que esta implementación tiene un límite práctico impuesto por la eficiencia del cálculo factorial para números muy grandes, aunque Python maneja enteros de precisión arbitraria.

```python
#fichero newton.py
#Calcula mediante el binomio de newton los datos datos

def factorial(n):
    res = 1
    for i in range(1, n + 1):
        res *= i
    return res

def combinatorio(n, k):
    return factorial(n) // (factorial(k) * factorial(n - k))

def binomio_newton(a, b, n):
    print(f"Desarrollo de ({a} + {b})^{n} = ", end="")
    resultado_total = 0

    for k in range(n + 1):
        comb = combinatorio(n, k)
        termino = comb * (a**(n-k)) * (b**k)
        resultado_total += termino

        if k > 0:
            print(" + ", end="")
        print(f"{comb}*{a}^{n-k}*{b}^{k}", end="")

    print(f"\nResultado calculado suma total: {resultado_total}")
    print(f"Comprobación directa ((a+b)^n): {(a+b)**n}")

if __name__ == "__main__":
    binomio_newton(2, 3, 4)
```

3.19. Implementar la función de Fibonacci para valores enteros positivos

Este ejercicio propone, dado un número entero positivo, calcular su valor de la sucesión de Fibonacci [5]. Para ello se utilizará la fórmula matemática de Fibonacci [5]:

$$F(n) = F(n-1) + F(n-2)$$

Nota: La implementación recursiva es más directa, siguiendo la definición matemática para Fibonacci, pero es ineficiente para valores grandes de n debido a la repetición de cálculos. Por tanto, se implementará la función de manera iterativa.

```python
# fichero fibo.py
# Función iterativa para calcular el n-ésimo número de Fibonacci

def fibonacci_iterativo(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b

if __name__ == "__main__":
    k = int(input("Introduce qué término de Fibonacci quieres
       calcular: "))
    print(f"Fibonacci({k}) = {fibonacci_iterativo(k)}")
```

3.20. Implementar un procedimiento para seleccionar y visualizar los K bits de orden inferior de un número entero proporcionado

Este ejercicio tiene como objetivo extraer y mostrar por pantalla los K bits menos significativos (los bits de más a la derecha o "últimos bits") de un número entero dado como entrada.

```
# fichero nkbits.py
# Procedimiento que escribe por pantalla los últimos k bits de un
   número entero.
def mostrar_k_bits(numero, k):
    print(f"Número: {numero} (Binario: {bin(numero)})")
    print(f"Los últimos {k} bits son:")

    # Recorremos desde el bit K-1 hasta el 0 (de izquierda a derecha
       en los inferiores)
    # O simplemente de 0 a K-1 e imprimimos
    bits_str = ""
    for i in range(k):
        # Desplazar el bit 'i' a la posición 0 y hacer AND con 1 para
           extraerlo
        bit = (numero >> i) & 1
        bits_str = str(bit) + bits_str # Prependemos para que se lea
           en orden correcto

    print(f"Bits: {bits_str}")

if __name__ == "__main__":
    num = int(input("Introduce un número entero: "))
    bits = int(input("Cuántos bits inferiores mostrar: "))
    mostrar_k_bits(num, bits)
```

3.21. Implementar una función que invierta los últimos K bits de orden inferior de un número entero proporcionado

En este ejercicio se pretende realizar una función que, dado un número entero y los k bits últimos que se desean invertir, devuelva el nuevo número. Para ello, la inversión de bits se logra eficientemente creando una máscara donde los últimos K bits están a 1 y el resto a 0, y luego aplicando el operador XOR entre el número y esa máscara. **Creación de la Máscara**

$1 \ll k$: Genera un número con el bit en la posición k encendido. Ejemplo con $k = 4$: 00010000 (valor decimal 16)

$(1 \ll k) - 1$: Al restar 1 de cualquier potencia de 2 (o valor con un solo bit encendido), se ponen a 1 todos los bits a la derecha de esa posición. Ejemplo con $k = 4$: $00010000 - 1 = 00001111$ (valor decimal 15)

Esta máscara, 00001111, tiene los 4 bits a la derecha listos para la inversión, y los 0's a la izquierda aseguran que el resto del número permanezca inalterado. **Aplicación del XOR**

La inversión se realiza mediante el operador $\oplus$ (XOR bit a bit):

Número Original (45):	$\ldots 0010\mathbf{1101}$
$\oplus$ Máscara ($k = 4$):	$\ldots 0000\mathbf{1111}$
Resultado (34):	$\ldots 0010\mathbf{0010}$

```python
# fichero mascara.py
# Invierte (cambia 0->1 y 1->0) los últimos k bits de un número entero
def invertir_bits(numero, k):
    mascara = (1 << k) - 1

    print(f"Original: {bin(numero)} ({numero})")
    print(f"Máscara : {bin(mascara)}")
    resultado = numero ^ mascara
    print(f"Invertido: {bin(resultado)} ({resultado})")
    return resultado

if __name__ == "__main__":
    n = 45 # Ejemplo del enunciado
    k = 4
    invertir_bits(n, k)
```

3.22. Invertir una cadena de texto leída por teclado, sin utilizar las funciones del propio lenguaje de programación

Este ejercicio propone realizar la inversión de una cadena de texto leída por teclado, pero sin utilizar funciones específicas del lenguaje. La inversión de una cadena de texto sin usar funciones de la clase `string` de alto nivel, requiere un proceso manual, típicamente utilizando un bucle y la técnica de intercambio de caracteres (swap) tal y como se verá a continuación.

```python
# fichero invierte.py
#Este ejercicio implementa la inversión de una cadena de texto dada

def invertir_manual(cadena):
    # En Python las cadenas son inmutables, así que las convertimos a
       lista
    lista_caracteres = list(cadena)
    inicio = 0
    fin = len(lista_caracteres) - 1

    # Bucle Swap
    while inicio < fin:
        temp = lista_caracteres[inicio]
        lista_caracteres[inicio] = lista_caracteres[fin]
        lista_caracteres[fin] = temp

        inicio += 1
        fin -= 1

    # Convertir lista de vuelta a cadena
    return "".join(lista_caracteres)

if __name__ == "__main__":
    texto = input("Introduce texto: ")
    invertido = invertir_manual(texto)
    print(f"Texto invertido: {invertido}")
```

3.23. Leer una cadena de caracteres desde teclado, decir longitud y concatenar con otra utilizando las funciones nativas de Python

Este ejercicio propone leer una cadena de caracteres, obtener su longitud y concatenarla con otra cadena (por ejemplo, ".txt"). En Python, estas operaciones se realizan de forma nativa mediante las funciones y operadores integrados (`len()` y `+`), que son eficientes y fáciles de usar. Aunque sería posible implementarlas manualmente mediante bucles, esto sería innecesario y menos eficiente, ya que Python ya proporciona mecanismos optimizados para trabajar con cadenas.

```
# fichero leelongi.py
# Este ejercicio lee una cadena desde teclado y dice la longitud y la
    concatena con otra.

def operaciones_string():

    cadena = input("Introduce una cadena: ")

    # Longitud
    longitud = len(cadena)
    print(f"La longitud es: {longitud}")

    # Concatenación
    nueva_cadena = cadena + ".txt"
    print(f"Concatenada: {nueva_cadena}")

if __name__ == "__main__":
    operaciones_string()
```

3.24. Convertir cadenas a letras mayúsculas y contar sus vocales

Este ejercicio consiste en leer varias cadenas de caracteres desde teclado, seguidamente convertirlas a letras mayúsculas y posteriormente contar sus vocales. En él se usarán tanto cadenas de caracteres como vectores (listas). Para este ejercicio no se utiliza la biblioteca estándar `string`, sino manipulación directa.

```
# fichero conteo.py
# Este ejercicio nos presenta la conversión de cadenas a letras
   mayúsculas y contar sus vocales

def procesar_cadenas():
    # Usamos una lista como si fuera un vector de strings
    frases = []

    print("Introduce 3 frases:")
    for i in range(3):
        frases.append(input(f"Frase {i+1}: "))

    vocales = "AEIOU"

    print("\n--- Resultados ---")
    for frase in frases:
        # Convertir a mayúsculas
        frase_upper = frase.upper()

        # Contar vocales
        num_vocales = 0
        for letra in frase_upper:
            if letra in vocales:
                num_vocales += 1

        print(f"Original: {frase}")
        print(f"Mayúsculas: {frase_upper}")
        print(f"Número de vocales: {num_vocales}")
        print("-" * 20)

if __name__ == "__main__":
    procesar_cadenas()
```

3.25. Función para realizar la trasposición de una matriz de números enteros dada

En este ejercicio se pretende crear una función para realizar la trasposición de una matriz de números enteros. Para ello se tendrá en cuenta la siguiente información: La **trasposición de una matriz** A (denotada como A^T) [2] es una operación que **cambia las filas por las columnas** y viceversa. Si la matriz original A tiene dimensiones $m \times n$ (filas $\times$ columnas), su matriz transpuesta A^T tendrá dimensiones $n \times m$. El elemento en la posición (i, j) de la matriz original A se convierte en el elemento en la posición (j, i) de la matriz transpuesta A^T [2].

$$A_{i,j} = (A^T)_{j,i}$$

```python
# fichero trasposicion.py
def imprimir_matriz(m):
    for fila in m:
        print(fila)
def trasponer_matriz(matriz):
    filas = len(matriz)
    columnas = len(matriz[0])
    matriz_T = [[0] * filas for _ in range(columnas)]
    for i in range(filas):
        for j in range(columnas):
            matriz_T[j][i] = matriz[i][j]
    return matriz_T
if __name__ == "__main__":
    # Matriz 2x3 de ejemplo
    A = [
        [1, 2, 3],
        [4, 5, 6]
    ]
    print("Matriz Original:")
    imprimir_matriz(A)

    AT = trasponer_matriz(A)

    print("\nMatriz Traspuesta:")
    imprimir_matriz(AT)
```

3.26. Función para multiplicar 2 matrices cuadradas de números enteros

En este ejercicio se pretende crear una función para realizar la multiplicación de 2 matrices cuadradas de números enteros. Para ello, se seguirá la explicación dada en [2], la cual indica que la multiplicación de dos matrices cuadradas, A y B, de dimensión $N \times N$, da como resultado una matriz C, también de dimensión $N \times N$. Para multiplicar $A \cdot B$, el número de columnas de A debe ser igual al número de filas de B. Si A y B son ambas $N \times N$, la condición siempre se cumple. Cada elemento de la matriz resultante, C_{ij}, se calcula como el producto escalar (la suma de productos) de la fila i de A por la columna j de B [2].

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$$

```python
# fichero multiplicamatrices.py
def multiplicar_matrices(A, B):
    N = len(A)
    C = [[0] * N for _ in range(N)]
    for i in range(N):
        for j in range(N):
            suma = 0
            for k in range(N):
                suma += A[i][k] * B[k][j]
            C[i][j] = suma
    return C
if __name__ == "__main__":
    m1 = [[1, 2], [3, 4]]
    m2 = [[2, 0], [1, 2]]

    print("Matriz A:", m1)
    print("Matriz B:", m2)
    res = multiplicar_matrices(m1, m2)
    print("Resultado A*B:")
    for fila in res:
        print(fila)
```

Capítulo 4

Manejo de Ficheros y Conexión a base de datos

Los lenguajes de programación ofrecen herramientas extremadamente potentes para interactuar directamente con los recursos fundamentales del sistema operativo: la memoria y el almacenamiento persistente sobre ficheros y bases de datos. Dominar estos dos conceptos es crucial para desarrollar aplicaciones que no sólo sean eficientes en el uso de recursos, sino también capaces de manejar grandes volúmenes de datos y mantener información entre sesiones. Hoy en día cada vez interactuan menos los lenguajes con ficheros, siendo estas funciones sustituidas por conexiones a bases de datos mediante librerías específicas o APIs. Aún así, es muy socorrido poder manejar ficheros donde guardar información en local (configuraciones básicas o archivos de logs). Con respecto a las conexiones de base de datos, veremos cómo desde Python es posible conectar mediante librerías a las principales bases de datos del mercado. Para revisar las especificaciones propias del lenguaje proponemos al lector revisar algunos de los libros más destacados en la bibliografía tales como [9] [18] [7].

4.1. Crear y escribir en un fichero

Este ejercicio nos propone crear un fichero de texto y escribir sobre él. Se pretende aprender los comandos básicos para crear un fichero (abrir) y cerrar el fichero.

```
# creafiche.py
def crear_fichero():
    try:
        # 'w' significa write (escritura). Si no existe, lo crea.
        with open("datos.txt", "w", encoding="utf-8") as fichero:
            fichero.write("Hola mundo.\n")
            fichero.write("Esto es una prueba de escritura en
               Python.\n")
            fichero.write("Python hace que el manejo de ficheros sea
               muy simple.\n")

        print("Fichero 'datos.txt' creado y escrito con éxito.")

    except IOError as e:
        print(f"Ocurrió un error al crear el fichero: {e}")

if __name__ == "__main__":
    crear_fichero()
```

4.2. Leer y mostrar contenido de un fichero de texto

Este ejercicio nos propone una procedimiento que lee un fichero de texto y nos muestra el contenido por pantalla

```
# leerfiche.py
def leer_fichero():
    try:
        # 'r' significa read (lectura)
        with open("datos.txt", "r", encoding="utf-8") as fichero:
            # Podemos iterar directamente sobre el objeto fichero
            for linea in fichero:
```

```
                print(linea.strip()) # strip() elimina el salto de
                    línea doble

    except FileNotFoundError:
        print("El fichero 'datos.txt' no existe. Ejecuta el ejercicio
            anterior primero.")

if __name__ == "__main__":
    leer_fichero()
```

4.3. Contar caracteres y palabras de un fichero de texto

Este ejercicio nos propone leer el archivo datos.txt (previamente creado en el ejercicio 4.1) y contar el total de caracteres y el número de espacios en blanco (que podemos usar como un estimador de palabras).

```
# contarcara.py
def analizar_fichero():
    try:
        with open("datos.txt", "r", encoding="utf-8") as fichero:
            contenido = fichero.read() # Lee todo el contenido en una
                sola variable string

            total_caracteres = len(contenido)
            total_espacios = contenido.count(' ')

            print(f"Total de caracteres: {total_caracteres}")
            print(f"Total de espacios en blanco: {total_espacios}")

    except FileNotFoundError:
        print("El fichero no existe.")

if __name__ == "__main__":
    analizar_fichero()
```

4.4. Guardar una estructura en un fichero binario y cargarla en memoria para ser mostrada por pantalla

Este ejercicio nos propone definir una estructura de datos, inicializar una instancia y guardarla directamente en un archivo binario (registro.dat) usando la librería *pickle* y los modos de lectura y escritura binaria de Python .

```python
# creabinario.py
import pickle

def manejo_binario():
    # Definimos la estructura de datos (un diccionario en este caso)
    persona = {
        "id": 1,
        "nombre": "Juan Perez",
        "altura": 1.75
    }

    nombre_fichero = "registro.dat"

    # 1. Guardar en binario ('wb' = write binary)
    with open(nombre_fichero, "wb") as f_salida:
        pickle.dump(persona, f_salida)
        print("Datos guardados en binario.")

    # 2. Cargar del binario ('rb' = read binary)
    with open(nombre_fichero, "rb") as f_entrada:
        datos_recuperados = pickle.load(f_entrada)

        print("\n Datos recuperados del binario - Ahora en Memoria")
        print(f"ID: {datos_recuperados['id']}")
        print(f"Nombre: {datos_recuperados['nombre']}")
        print(f"Altura: {datos_recuperados['altura']}")

if __name__ == "__main__":
    manejo_binario()
```

4.5. Insertar al final de un fichero de texto

Este ejercicio nos propone, dado un fichero de texto datos.txt, escribir una nueva línea al final del mismo. Utilizar para ello el modo de apertura de los ficheros en Python.

```
# escribefinal.py
def anadir_al_final():
    try:
        # 'a' significa append (añadir al final)
        with open("datos.txt", "a", encoding="utf-8") as fichero:
            fichero.write("\nEsta es una nueva línea insertada al
                final con Python.")
            print("Línea añadida correctamente.")

    except IOError:
        print("Error al abrir el archivo.")

if __name__ == "__main__":
    anadir_al_final()
```

4.6. Conexión con una base de datos desde Python

Este ejercicio nos propone, realizar una conexión con una base de datos SQLite, crear una tabla que se llame usuarios, insertar un valor de un usuario y posteriormente leerlo y mostrarlo.

```
import sqlite3

def conectar_bd():
    # Crea una conexión a una base de datos local (o en memoria)
    conexion = sqlite3.connect("ejemplo.db")
    cursor = conexion.cursor()

    # Crear tabla
    cursor.execute("CREATE TABLE IF NOT EXISTS usuarios (id INTEGER,
        nombre TEXT)")

    # Insertar datos
    cursor.execute("INSERT INTO usuarios VALUES (1, 'Ana')")
    conexion.commit()

    # Leer datos
    cursor.execute("SELECT * FROM usuarios")
    print(cursor.fetchall())

    # Cerrar conexión
    conexion.close()

if __name__ == "__main__":
    conectar_bd()
```

Capítulo 5

Algoritmos de ordenación y búsqueda

En este capítulo vamos a realizar un repaso a los algoritmos más destacados de la literatura sobre ordenación y búsqueda [13] [8] [26]. Dichos algoritmos vienen a mejorar los métodos tradicionales para abordar este tipo de problemas y ayudan a mejorar los tiempos de ejecución, haciendo nuestros programas mucho más eficientes en coste de tiempo y recursos. Cuando trabajamos con datos, es muy importante que las estructuras que utilicemos tengan unos tiempos de búsqueda eficientes, para ello, normalmente, se suelen utilizar métodos avanzados de inserción en la estructura de datos y de búsqueda. Muchas veces el método de inserción hace uso de métodos de ordenación para conseguir esa eficiencia en las búsquedas, de ahí la importancia de ver estos algoritmos utilizados en problemas.

5.1. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo de la Burbuja

Este ejercicio nos propone, dado un conjunto de números enteros desordenado, devolver el conjunto ordenado utilizando el **algoritmo de ordenación de la burbuja**, también conocido como **Bubble Sort** [13]. La simplicidad del algoritmo tiene un coste en eficiencia. La complejidad temporal del Método de la Burbuja es: $O(n^2)$. Donde n es el número de elementos. Esto lo hace poco práctico para conjuntos de datos grandes en comparación con otros algoritmos más eficientes como *Quick Sort* o *Merge Sort*.

```python
# fichero burbuja.py
def ordenacion_burbuja(lista):
    n = len(lista)
    # Recorremos todos los elementos de la lista
    for i in range(n):
        intercambio = False

        # Últimos i elementos ya están ordenados
        for j in range(0, n - i - 1):
            # Intercambiar si el elemento encontrado es mayor que el
                siguiente
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
                intercambio = True

        # Si no hubo intercambios en la pasada, la lista ya está
            ordenada
        if not intercambio:
            break
    return lista

if __name__ == "__main__":
    datos = [64, 34, 25, 12, 22, 11, 90]
    print("Original:", datos)
    ordenado = ordenacion_burbuja(datos.copy())
    print("Ordenado (Burbuja):", ordenado)
```

5.2. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo Quicksort

Este ejercicio nos propone, dado un conjunto de números desordenado, devolver el conjunto ordenado utilizando el **algoritmo de ordenación Quicksort** [13]. Quicksort es un algoritmo de ordenación extremadamente eficiente y ampliamente utilizado. Es conocido por ser uno de los más rápidos en la práctica para grandes conjuntos de datos. En el caso **Promedio y Mejor Caso**, su orden de complejidad es de $O(n \log n)$ y en el **peor de los casos** es de $O(n^2)$ [13]. Este último caso resulta en subproblemas muy desequilibrados ($n-1$ y 0), forzando al algoritmo a realizar un número de comparaciones similar al del *Bubble Sort*. En este ejercicio haremos la implementación del algoritmo Quicksort.

Nota: La función qsort no está disponible en Python como en otros lenguajes de programación. Python utiliza el método sorted que implementa *TimSort* [14][19], que es un algoritmo híbrido muy eficiente que combina Merge Sort (Ordenación por mezcla) e Insertion Sort (Ordenación por inserción).

```python
# fichero qsort.py

def quicksort(lista):
    # Caso base: listas de 0 o 1 elemento ya están ordenadas
    if len(lista) <= 1:
        return lista

    # Elegimos el pivote (aquí tomamos el último elemento)
    pivote = lista[-1]

    # Partición
    menores = [x for x in lista[:-1] if x <= pivote]
    mayores = [x for x in lista[:-1] if x > pivote]

    # Llamada recursiva
    return quicksort(menores) + [pivote] + quicksort(mayores)

if __name__ == "__main__":
    datos = [10, 7, 8, 9, 1, 5]
    print("Original:", datos)
    print("Ordenado (Quicksort):", quicksort(datos))
```

5.3. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo TimSort utilizando el método propio de Python

Este ejercicio nos propone, dado un conjunto de números desordenados, devolver el conjunto ordenado utilizando el **algoritmo de ordenación TimSort** definido en Python, tal y como hemos explicado en el ejercicio anterior 5.2.

```
# fichero timsort.py

def ordenacion_timsort_nativa(lista):

    # Python utiliza TimSort en sus métodos built-in
    nueva_lista = sorted(lista)
    return nueva_lista

if __name__ == "__main__":
    datos = [34, 2, 15, 88, 1, 60]
    print("Original:", datos)

    resultado = ordenacion_timsort_nativa(datos)

    print("Ordenado (Python/TimSort):", resultado)
```

5.4. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo MergeSort

Este ejercicio nos propone, dado un conjunto de números desordenado, devolver el conjunto ordenado utilizando el **Algoritmo de ordenación MergeSort** [13]. **MergeSort** es otro algoritmo de ordenación muy eficiente, está basado en el principio de "Divide y Vencerás"[13]. A diferencia de Quicksort, su rendimiento es consistente: siempre $O(n \log n)$ en el mejor, promedio y peor caso [13].

5.4. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo MergeSort

```
#fichero mergesort.py

def mergesort(lista):
    if len(lista) <= 1:
        return lista

    # División
    medio = len(lista) // 2
    izquierda = mergesort(lista[:medio])
    derecha = mergesort(lista[medio:])

    # Fusión (Merge)
    return merge(izquierda, derecha)

def merge(izquierda, derecha):
    resultado = []
    i = j = 0

    # Comparar elementos y ordenarlos
    while i < len(izquierda) and j < len(derecha):
        if izquierda[i] < derecha[j]:
            resultado.append(izquierda[i])
            i += 1
        else:
            resultado.append(derecha[j])
            j += 1

    # Añadir los elementos restantes
    resultado.extend(izquierda[i:])
    resultado.extend(derecha[j:])

    return resultado

if __name__ == "__main__":
    datos = [38, 27, 43, 3, 9, 82, 10]
    print("Original:", datos)
    print("Ordenado (Mergesort):", mergesort(datos))
```

5.5. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo Shellsort

Este ejercicio nos propone, dado un conjunto de números desordenado, devolver el conjunto ordenado utilizando el **algoritmo de ordenación de ShellSort** [26].

El rendimiento del algoritmo **Shellsort** depende crucialmente de la elección de la secuencia de saltos o intervalos h. Una de las secuencias más eficientes y comunes, que optimiza significativamente el tiempo de ejecución en el peor caso, es la propuesta por D. Knuth [8].

La secuencia de intervalos de Knuth [8] se define recursivamente mediante la fórmula $h_{k+1} = 3h_k + 1$. El punto de partida de la secuencia es $h_1 = 1$. Los primeros términos de esta secuencia son: $1, 4, 13, 40, 121, 364, 1093, \ldots$

Utilizar la secuencia de Knuth [8] proporciona al Shellsort una complejidad temporal significativamente mejor que las secuencias triviales (como $h = N/2$): Se ha demostrado que con esta secuencia, el Shellsort alcanza un rendimiento cercano a $O(n^{3/2})$ en el peor caso, o incluso mejor, en torno a $O(n \log^2 n)$, superando con creces la complejidad $O(n^2)$ de los algoritmos de ordenación simples.

```python
# fichero shellsort.py
def shellsort_knuth(lista):
    n = len(lista)
    h = 1
    while h < n // 3:
        h = 3 * h + 1
    while h >= 1:
        for i in range(h, n):
            temp = lista[i]
            j = i
            while j >= h and lista[j - h] > temp:
                lista[j] = lista[j - h]
                j -= h

            lista[j] = temp
        h //= 3
    return lista
if __name__ == "__main__":
    datos = [12, 34, 54, 2, 3]
    print("Original:", datos)
    print("Ordenado (ShellSort Knuth):", shellsort_knuth(datos.copy()))
```

5.6. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo RadixSort

Este ejercicio nos propone, dado un conjunto de números desordenado, devolver el conjunto ordenado utilizando el **algoritmo de ordenación de RadixSort** [26] [8].

Radix Sort (ordenación por base o por dígitos) es un algoritmo de ordenación no comparativo que organiza los elementos procesando sus dígitos de forma secuencial [26]. En lugar de comparar directamente los valores entre sí (como hace, por ejemplo, *Quicksort* o *Mergesort*), este algoritmo clasifica los números según cada posición decimal (unidades, decenas, centenas, etc.), utilizando un algoritmo de ordenación **estable** en cada paso, normalmente *Counting Sort*. Counting Sort (ordenación por conteo) [26] [8] es un algoritmo no comparativo que se utiliza para ordenar números enteros dentro de un rango limitado. En lugar de comparar elementos entre sí, cuenta cuántas veces aparece cada valor, y a partir de esas frecuencias reconstruye el array ordenado. La complejidad temporal es: $O(d \cdot (n+k))$ y la complejidad espacial es: $O(n+k)$. El algoritmo *Radix Sort* es un método de ordenación eficiente para números enteros cuando los valores son de longitud limitada. Su rendimiento cercano a $O(n)$ lo convierte en una buena opción para grandes volúmenes de datos, siempre que la base y la longitud de los números no sean demasiado grandes [26] [8].

```python
#fichero radixsort.py
def counting_sort_para_radix(lista, exp):
    n = len(lista)
    salida = [0] * n
    conteo = [0] * 10 # Base 10 (dígitos 0-9)
    # Contar ocurrencias según el dígito actual (exp)
    for i in range(n):
        indice = (lista[i] // exp) % 10
        conteo[indice] += 1
    # Calcular posiciones reales en salida
    for i in range(1, 10):
        conteo[i] += conteo[i - 1]
    # Construir el array de salida (recorriendo hacia atrás para
       estabilidad)
    i = n - 1
    while i >= 0:
        indice = (lista[i] // exp) % 10
        salida[conteo[indice] - 1] = lista[i]
```

```
        conteo[indice] -= 1
        i -= 1
    # Copiar al original
    for i in range(n):
        lista[i] = salida[i]

def radixsort(lista):
    if not lista:
        return lista
    # Encontrar el número máximo para saber cantidad de dígitos
    maximo = max(lista)
    # Aplicar counting sort para cada posición decimal (1, 10, 100...)
    exp = 1
    while maximo // exp > 0:
        counting_sort_para_radix(lista, exp)
        exp *= 10
    return lista

if __name__ == "__main__":
    datos = [170, 45, 75, 90, 802, 24, 2, 66]
    print("Original:", datos)
    print("Ordenado (RadixSort):", radixsort(datos.copy()))
```

5.7. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo HeapSort

Este ejercicio nos propone, dado un conjunto de números enteros desordenado, devolver el conjunto ordenado utilizando el **algoritmo de ordenación de HeapSort** [26]. **Heapsort** (Ordenación por Montículo) es un algoritmo de ordenación por comparación que utiliza la estructura de datos **montículo binario (heap)** para ordenar elementos. Es un algoritmo eficiente, con una complejidad temporal de $\mathcal{O}(n \log n)$ en el mejor, promedio y peor caso. Heapsort se basa en la propiedad de un **Max-Heap** (Montículo Máximo). Un Max-Heap [26] es un árbol binario completo donde:

- El **valor de cada nodo es mayor o igual** que los valores de sus hijos.
- El **elemento más grande** de todo el montículo siempre está en la **raíz** del árbol.

5.7. Dado un conjunto de números enteros desordenado, ordenar mediante el algoritmo HeapSort

```python
# fichero heapsort.py
def heapify(lista, n, i):
    mas_grande = i      # Inicializar el más grande como raíz
    izq = 2 * i + 1     # Hijo izquierdo
    der = 2 * i + 2     # Hijo derecho

    # Ver si hijo izquierdo existe y es mayor que la raíz
    if izq < n and lista[izq] > lista[mas_grande]:
        mas_grande = izq
    # Ver si hijo derecho existe y es mayor que el más grande hasta
       ahora
    if der < n and lista[der] > lista[mas_grande]:
        mas_grande = der
    # Si la raíz no es el más grande, intercambiar
    if mas_grande != i:
        lista[i], lista[mas_grande] = lista[mas_grande], lista[i]
        # Recursivamente hacer heapify en el subárbol afectado
        heapify(lista, n, mas_grande)

def heapsort(lista):
    n = len(lista)
    for i in range(n // 2 - 1, -1, -1):
        heapify(lista, n, i)
    for i in range(n - 1, 0, -1):
        lista[i], lista[0] = lista[0], lista[i]
        heapify(lista, i, 0)

    return lista

if __name__ == "__main__":
    datos = [12, 11, 13, 5, 6, 7]
    print("Original:", datos)
    print("Ordenado (HeapSort):", heapsort(datos.copy()))
```

5.8. Dado un conjunto de números enteros, realizar la búsqueda de un número mediante el algoritmo de Búsqueda Lineal

Este ejercicio nos propone, dado un conjunto de números enteros, buscar un número perteneciente al conjunto utilizando el **algoritmo de búsqueda lineal**. La Búsqueda Lineal (o secuencial) [13] es el algoritmo de búsqueda más simple. Su complejidad es de orden **lineal** ($O(n)$) porque el tiempo de ejecución crece directamente con el número de elementos (n) del array. La complejidad es $O(1)$ si el elemento se encuentra en la **primera posición**. En promedio, se espera buscar $\frac{n}{2}$ elementos, lo que sigue siendo proporcional a n, que es el peor de los casos. Para conjuntos de datos grandes y **ordenados**, la **Búsqueda Binaria** es mucho más eficiente, con una complejidad temporal de $O(\log n)$.

```python
# fichero blineal.py

def busqueda_lineal(lista, objetivo):
    #Devuelve el  índice del elemento o -1 si no existe.
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

if __name__ == "__main__":
    datos = [10, 50, 30, 70, 80, 20, 90, 40]
    buscado = 30

    posicion = busqueda_lineal(datos, buscado)

    if posicion != -1:
        print(f"Número {buscado} encontrado en índice {posicion}.")
    else:
        print(f"Número {buscado} no encontrado.")
```

5.9. Dado un conjunto de números enteros ordenados, realizar la búsqueda de un número mediante el algoritmo de Búsqueda Binaria

Este ejercicio nos propone, dado un conjunto de números ordenados, buscar un número perteneciente al conjunto utilizando el **algoritmo de búsqueda binaria** [13]. La Búsqueda Binaria utiliza la estrategia de Divide y Vencerás [13]. La principal ventaja de la Búsqueda Binaria sobre la Búsqueda Lineal es su velocidad. La complejidad temporal es logarítmica, lo que significa que el tiempo de ejecución crece muy lentamente a medida que aumenta el tamaño de los datos, siendo esta $O(\log n)$.

```python
# fichero bbinaria.py
def busqueda_binaria(lista, objetivo):

    izquierda = 0
    derecha = len(lista) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

if __name__ == "__main__":
    datos_ordenados = [2, 3, 4, 10, 40, 55, 60, 70]
    buscado = 10

    pos = busqueda_binaria(datos_ordenados, buscado)

    if pos != -1:
        print(f"Número {buscado} encontrado en índice {pos}.")
    else:
        print("Número no encontrado.")
```

5.10. Dado un conjunto de números enteros no negativos ordenados, realizar la búsqueda de un número mediante el algoritmo de Búsqueda por Interpolación

Este ejercicio nos propone, dado un conjunto de números enteros no negativos ordenado, buscar un número perteneciente al conjunto utilizando el **algoritmo de búsqueda por interpolación** [26] [8]. Éste, es un algoritmo de búsqueda diseñado para conjuntos de datos ordenados que asume una distribución uniforme de los elementos. Es una mejora de la Búsqueda Binaria, ya que no solo divide el espacio de búsqueda a la mitad, sino que estima la posición del elemento objetivo basándose en su valor.

En el caso promedio, y cuando los datos están uniformemente distribuidos, la Búsqueda por Interpolación es más rápida que la Búsqueda Binaria ($O(\log \log n)$ vs $O(\log n)$). En el peor de los casos son idénticos $O(n)$.

```
# fichero binterpola.py
def busqueda_interpolacion(lista, objetivo):
    #Requiere lista ORDENADA y distribución UNIFORME.

    bajo = 0
    alto = len(lista) - 1

    while bajo <= alto and objetivo >= lista[bajo] and objetivo <=
        lista[alto]:
        if bajo == alto:
            if lista[bajo] == objetivo:
                return bajo
            return -1

        # Fórmula de interpolación para estimar posición
        # pos = lo + [ (x-arr[lo]) * (hi-lo) / (arr[hi]-arr[lo]) ]
        pos = bajo + int(((float(alto - bajo) / (lista[alto] -
            lista[bajo])) * (objetivo - lista[bajo])))

        if lista[pos] == objetivo:
            return pos

        if lista[pos] < objetivo:
```

```
            bajo = pos + 1
        else:
            alto = pos - 1

    return -1

if __name__ == "__main__":
    # Funciona mejor con datos uniformemente distribuidos
    datos = [10, 20, 30, 40, 50, 60, 70, 80, 90]
    buscado = 70

    idx = busqueda_interpolacion(datos, buscado)

    if idx != -1:
        print(f"Elemento {buscado} encontrado en índice {idx} mediante
            interpolación.")
    else:
        print("Elemento no encontrado.")
```

Además de todos los algoritmos revisados anteriormente, existen otros algoritmos de búsqueda sobre estructuras de datos más complejas como árboles o grafos, los cuales quedan fuera de este libro de problemas básicos. Para más información sobre algoritmos avanzados véase [26] [13] [8].

Capítulo 6

Ejercicios de Criptografía Básica

La criptografía es la ciencia que se encarga de proteger la información mediante el uso de algoritmos matemáticos, asegurando que el mensaje solo pueda ser leído por el destinatario previsto. En el ámbito de la programación en Python, la implementación de algoritmos criptográficos básicos es un excelente ejercicio para dominar el manejo de cadenas (strings), la aritmética modular y las operaciones a nivel de bits (bitwise).

Históricamente, la criptografía se divide en clásica y moderna. En este capítulo nos centraremos principalmente en la **criptografía clásica** y simétrica básica, que, aunque no es segura para estándares modernos, sienta las bases lógicas de los sistemas actuales. Para profundizar en la teoría matemática y estándares modernos, se recomienda consultar obras de referencia como las de Bruce Schneier [20], Douglas Stinson [21] o Manuel Lucena [10].

A continuación, presentamos una serie de ejercicios que van desde el cifrado por desplazamiento hasta la manipulación de bits mediante operaciones XOR.

6.1. Implementar el Cifrado César (Desplazamiento)

Este ejercicio propone implementar el famoso **Cifrado César** [10], uno de los métodos de codificación más antiguos y simples. Es un tipo de cifrado por sustitución en el que una letra en el texto original es reemplazada por otra letra que se encuentra un número fijo de posiciones más adelante en el alfabeto. Matemáticamente, para cifrar una letra x con un desplazamiento n:

$$E_n(x) = (x + n) \mod 26$$

El programa debe leer una cadena y un entero n (la clave), y mostrar la cadena cifrada. Se debe tener cuidado con el desbordamiento del alfabeto (volver de la 'z' a la 'a').

```
# fichero cesar_cifrar.py
# Implementación del Cifrado César (Desplazamiento)
def cifrar_cesar(mensaje, desplazamiento):
    resultado = ""
    for ch in mensaje:
        if ch.isalpha():
            # Determinar la base ('A' para mayúsculas, 'a' para
                minúsculas)
            base = ord('A') if ch.isupper() else ord('a')
            # Fórmula del cifrado: E(x) = (x + n) mod 26
            # Usamos ord() para convertir char a int y chr() para
                volver a char
            nuevo_caracter = chr((ord(ch) - base + desplazamiento) %
                26 + base)
            resultado += nuevo_caracter
        else:
            resultado += ch
    return resultado
if __name__ == "__main__":
    try:
        mensaje = input("Introduce el mensaje a cifrar: ")
        n_input = input("Introduce el desplazamiento (clave n): ")
        n = int(n_input)
        mensaje_cifrado = cifrar_cesar(mensaje, n)
        print(f"\nMensaje Cifrado: {mensaje_cifrado}")
    except ValueError:
        print("\nError: La clave debe ser un número entero.")
```

6.2. Descifrar el Código César con Clave Conocida

Este ejercicio es la operación inversa al anterior 6.1. Dado un texto cifrado y la clave n utilizada, se debe recuperar el mensaje original. La fórmula para el descifrado es [10]:

$$D_n(x) = (x - n) \mod 26$$

A diferencia de otros lenguajes como C++, Python maneja el operador módulo (%) con números negativos de forma matemática estándar (ej. $-1 \mod 26 = 25$), lo cual simplifica enormemente la implementación de la rotación inversa sin necesidad de ajustes adicionales.

```python
# fichero cesar_descifrar.py
# Implementación del Descifrado César (Inversa del desplazamiento)
def descifrar_cesar(mensaje, desplazamiento):
    resultado = ""

    for ch in mensaje:
        if ch.isalpha():
            # Determinar la base ('A' o 'a')
            base = ord('A') if ch.isupper() else ord('a')
            # Fórmula de descifrado: D(x) = (x - n) mod 26
            nuevo_valor = (ord(ch) - base - desplazamiento) % 26
            resultado += chr(base + nuevo_valor)
        else:
            resultado += ch

    return resultado

if __name__ == "__main__":

    try:
        mensaje = input("Introduce el mensaje cifrado: ")
        n_input = input("Introduce la clave original (n): ")
        n = int(n_input)
        mensaje_descifrado = descifrar_cesar(mensaje, n)
        print(f"\nMensaje Descifrado: {mensaje_descifrado}")

    except ValueError:
        print("\nError: La clave debe ser un número entero.")
```

6.3. Ataque de Fuerza Bruta al Cifrado César

Este ejercicio propone, dado un texto cifrado mediante el método César del cual desconocemos la clave, mostrar por pantalla **todas las posibles combinaciones** de descifrado (las 25 rotaciones posibles). Este es un ejemplo básico de criptoanálisis mediante fuerza bruta, demostrando la debilidad de los cifrados con un espacio de claves pequeño. El usuario deberá leer la salida y determinar visualmente cuál es el mensaje con sentido.

```
# fichero cesar_bruta.py
# Ataque de fuerza bruta al Cifrado César
# Prueba todas las claves posibles (1-25) para que el usuario
   identifique el mensaje
def imprimir_intento(mensaje, key):
    resultado = ""
    for ch in mensaje:
        if ch.isalpha():
            # Determinar si es mayúscula o minúscula
            base = ord('A') if ch.isupper() else ord('a')
            # Fórmula de descifrado: D(x) = (x - key) mod 26
            nuevo_valor = (ord(ch) - base - key) % 26
            resultado += chr(base + nuevo_valor)
        else:
            resultado += ch

    print(f"Clave {key:2}: {resultado}")

if __name__ == "__main__":

    try:
        mensaje = input("Introduce el texto cifrado para atacar por
           fuerza bruta: ")
        print("\n Resultados del Criptoanálisis \n")
        # Bucle para probar todas las rotaciones posibles (del 1 al 25)
        for k in range(1, 26):
            imprimir_intento(mensaje, k)

    except KeyboardInterrupt:
        print("\nOperación cancelada.")
```

6.4. Cifrado y Descifrado mediante Operación XOR

Este ejercicio introduce el concepto de cifrado de flujo simétrico utilizando la operación **XOR** ($\oplus$) [20]. La operación XOR tiene una propiedad fundamental para la informática: $(A \oplus K) \oplus K = A$ es su propia inversa. El programa debe solicitar una cadena de texto y una clave (carácter o cadena corta). Se aplicará la operación XOR (operador ^ en Python) a cada carácter del mensaje con la clave. Al aplicar la misma operación al resultado, se obtendrá el mensaje original.

```
# fichero xor_cifra_descifra.py
# Cifrado y Descifrado mediante operación XOR
def xor_cifrado(texto, clave):
    resultado = []
    len_clave = len(clave)
    for i, char in enumerate(texto):
        char_clave = clave[i % len_clave]
        # Operación XOR a nivel de bits
        xor_valor = ord(char) ^ ord(char_clave)
        resultado.append(chr(xor_valor))
    return "".join(resultado)
if __name__ == "__main__":
    try:
        mensaje = input("Introduce el texto: ")
        clave = input("Introduce la clave: ")
        if not clave:
            print("Error: La clave no puede estar vacía.")
        else:
            # Cifrado
            mensaje_cifrado = xor_cifrado(mensaje, clave)
            print("\nTexto Cifrado (Visualización Hexadecimal):")
            hex_output = " ".join(f"{ord(c):02X}" for c in
                mensaje_cifrado)
            print(hex_output)
            # Descifrado
            mensaje_descifrado = xor_cifrado(mensaje_cifrado, clave)
            print(f"\nTexto Descifrado (original):
                {mensaje_descifrado}")
    except ValueError:
        print("Error en el procesamiento de datos.")
```

6.5. Implementar el Cifrado de Vigenère

Este ejercicio propone implementar el **Cifrado de Vigenère**, un método de cifrado polialfabético que mejora la seguridad del César utilizando una palabra clave en lugar de un solo número. La clave se repite tantas veces como sea necesario para igualar la longitud del mensaje. Si el mensaje es M y la clave es K, el carácter cifrado C_i en la posición i es: $C_i = (M_i + K_{i \pmod L}) \bmod 26$ donde L es la longitud de la clave.

```
# fichero vigenere.py
# Implementación del Cifrado de Vigenere (Polialfabético)
def cifrar_vigenere(mensaje, clave):
    resultado = []
    len_clave = len(clave)
    # Si la clave está vacía, devolvemos el mensaje tal cual
    if len_clave == 0:
        return mensaje
    # Aseguramos que la clave esté en mayúsculas para simplificar el
       cálculo del desplazamiento
    clave = clave.upper()
    indice_clave = 0
    for char in mensaje:
        if char.isalpha():
            # Determinar la base ('A' o 'a')
            base = ord('A') if char.isupper() else ord('a')
            # Calculamos el desplazamiento 'k' basado en la letra
               actual de la clave
            # Restamos 'A' para obtener un valor entre 0 y 25
            char_clave = clave[indice_clave % len_clave]
            k = ord(char_clave) - ord('A')
            # Fórmula Vigenere: C = (M + K) mod 26
            nuevo_valor = (ord(char) - base + k) % 26
            resultado.append(chr(base + nuevo_valor))
            # Avanzamos el índice de la clave SOLO si hemos cifrado
               una letra
            indice_clave += 1
        else:
            # Si no es letra, se añade sin cambios y NO avanzamos el í
               ndice de la clave
            resultado.append(char)
```

```
    return "".join(resultado)
if __name__ == "__main__":
    try:
        mensaje = input("Introduce el mensaje: ")
        clave_input = input("Introduce la palabra clave (solo letras):
            ")
        # Limpiamos la clave para quedarnos solo con letras (para
            evitar errores matemáticos)
        clave = "".join(c for c in clave_input if c.isalpha())
        if not clave:
            print("Error: La clave debe contener al menos una letra.")
        else:
            mensaje_cifrado = cifrar_vigenere(mensaje, clave)
            print(f"\nMensaje Cifrado Vigenere: {mensaje_cifrado}")
    except Exception as e:
        print(f"Error inesperado: {e}")
```

6.6. Cifrado por Sustitución Monoalfabética con Clave Aleatoria

A diferencia del César, donde el alfabeto se desplaza, este ejercicio propone utilizar un alfabeto completamente desordenado como clave. Se debe generar (o introducir) una cadena de 26 caracteres únicos que represente la permutación del alfabeto normal. Ejemplo de mapeo: A $\rightarrow$ Q, B $\rightarrow$ W, C $\rightarrow$ E ...

El programa sustituirá cada letra del mensaje original por su correspondiente en el alfabeto de sustitución.

```
# fichero sustitucion.py
# Implementación del Cifrado por Sustitución Monoalfabética
# Utilizamos un alfabeto desordenado (clave) para sustituir al
   original.
ALFABETO_NORMAL = "abcdefghijklmnopqrstuvwxyz"
ALFABETO_CLAVE  = "qwertyuiopasdfghjklzxcvbnm"

def cifrar_sustitucion(mensaje):
    resultado = []
    for char in mensaje:
        if char.isalpha():
            es_mayuscula = char.isupper()
```

```
            char_lower = char.lower()
            # Buscamos la posición en el alfabeto normal
            if char_lower in ALFABETO_NORMAL:
                posicion = ALFABETO_NORMAL.index(char_lower)
                # Obtenemos el sustituto del alfabeto clave
                sustituto = ALFABETO_CLAVE[posicion]
                # Restauramos mayúscula si es necesario
                if es_mayuscula:
                    sustituto = sustituto.upper()
                resultado.append(sustituto)
            else:
                resultado.append(char)
        else:
            resultado.append(char)
    return "".join(resultado)

if __name__ == "__main__":
    print(f"Alfabeto Clave: {ALFABETO_CLAVE}")
    try:
        mensaje = input("Introduce el mensaje a cifrar: ")
        mensaje_cifrado = cifrar_sustitucion(mensaje)
        print(f"Mensaje con Sustitución: {mensaje_cifrado}")

    except Exception as e:
        print(f"Error: {e}")
```

Aunque el algoritmo anterior es ideal para aprender la lógica paso a paso, Python ofrece herramientas nativas optimizadas para estas tareas. Podemos reducir toda la función de sustitución a solo dos líneas utilizando `maketrans`:

```
# Crear la tabla de traducción
tabla = str.maketrans(ALFABETO_NORMAL, ALFABETO_CLAVE)

# Aplicar la traducción
print(mensaje.lower().translate(tabla))
```

6.7. Cifrado por Transposición Columnar

Este ejercicio se enfoca en la **transposición** [10], donde no se cambian las letras, sino su posición. Se inspira en la antigua Escítala espartana. El algoritmo consiste en escribir el mensaje en una matriz de ancho fijo (determinado por la clave) y leer el mensaje columna por columna.

```
# fichero transposicion.py
# Implementación del Cifrado por Transposición Columnar
def cifrar_transposicion(mensaje, columnas):
    longitud = len(mensaje)
    filas = (longitud + columnas - 1) // columnas
    resultado = []
    # Recorremos columna por columna
    for c in range(columnas):
        # Y dentro de cada columna, bajamos por las filas
        for f in range(filas):
            # Calculamos el índice como si fuera una matriz aplanada
            index = f * columnas + c
            # Verificamos que no nos salimos del mensaje
            if index < longitud:
                resultado.append(mensaje[index])
    return "".join(resultado)
if __name__ == "__main__":
    try:
        entrada = input("Introduce el mensaje (sin espacios): ")
        mensaje = entrada.replace(" ", "")
        columnas_input = input("Introduce el número de columnas
            (clave): ")
        columnas = int(columnas_input)
        if columnas < 1:
            print("Error: Las columnas deben ser al menos 1.")
        else:
            cifrado = cifrar_transposicion(mensaje, columnas)
            print(f"\nMensaje Original:    {mensaje}")
            print(f"Mensaje Transpuesto: {cifrado}")

    except ValueError:
        print("\nError: La clave debe ser un número entero.")
```

Aqui podemos ver cómo Python nos da una herramienta para hacer el cifrado de manera más moderna y eficiente al estilo Python.

```
def cifrar_transposicion_pro(mensaje, columnas):
    # Une las columnas generadas por saltos
    # range(columnas) nos da el inicio de cada columna (0, 1, 2...)
    # mensaje[i::columnas] toma caracteres saltando de 'columnas' en
       'columnas'
    return "".join(mensaje[i::columnas] for i in range(columnas))
```

6.8. Análisis de Frecuencias Básico

Este ejercicio es una herramienta de criptoanálisis. Dado un texto cifrado (presumiblemente por sustitución), el programa debe contar la aparición de cada letra y mostrar el porcentaje de frecuencia de cada una. En el idioma español, las letras 'E' y 'A' son las más frecuentes. Este programa ayuda a identificar qué símbolo del texto cifrado corresponde probablemente a estas letras, facilitando el descifrado sin conocer la clave.

```
# fichero frecuencia.py
# Análisis de Frecuencias de caracteres
# Calcula cuántas veces aparece cada letra y su porcentaje respecto al
   total.
def analizar_frecuencias(mensaje):
    conteo = [0] * 26
    total_letras = 0
    for char in mensaje:
        if char.isalpha():
            # Convertimos a minúscula para normalizar
            char_lower = char.lower()
            # Calculamos el índice (0 para 'a', 1 para 'b', etc.)
            index = ord(char_lower) - ord('a')
            # Incrementamos el contador en esa posición
            conteo[index] += 1
            total_letras += 1
    return conteo, total_letras
if __name__ == "__main__":
    try:
        mensaje = input("Introduce el texto cifrado para analizar: ")
```

```
        conteo, total = analizar_frecuencias(mensaje)
        print("\n Resultados del Análisis \n")
        print(f"Total de letras analizadas: {total}")
        if total > 0:
            for i in range(26):
                if conteo[i] > 0:
                    porcentaje = (conteo[i] / total) * 100
                    # Recuperamos la letra en mayúscula para mostrarla
                    letra = chr(ord('A') + i)
                    print(f"Letra '{letra}': {conteo[i]:3} veces
                        ({porcentaje:.2f}%)")
        else:
            print("No se encontraron letras en el texto.")
    except Exception as e:
        print(f"Error: {e}")
```

Aunque usar una lista fija [0]*26 es muy didáctico para entender cómo funcionan el análisis de frecuencias con los arrays en memoria, en el mundo real de Python se utiliza la clase Counter de la librería collections. Aquí podemos ver un ejemplo.

```
from collections import Counter

mensaje = "Hola mundo"
# Cuenta automáticamente todo y devuelve un diccionario
frecuencias = Counter(c.lower() for c in mensaje if c.isalpha())
print(frecuencias.most_common())
# Salida: [('o', 2), ('h', 1), ('l', 1), ('a', 1), ('m', 1), ('u', 1),
    ('n', 1), ('d', 1)]
```

6.9. Generación de un Hash Simple (Checksum)

La criptografía no solo protege la confidencialidad, sino también la **integridad**. Este ejercicio propone implementar una función de hash sencilla (inspirada en algoritmos clásicos como K&R [1] o similares). El programa debe leer una cadena y devolver un valor numérico único (hash). Si se cambia un solo carácter de la cadena, el valor del hash debe cambiar, demostrando cómo se detectan modificaciones en los datos.

```
# fichero hash_simple.py
# Implementación de una función Hash simple (Estilo K&R)
# Transforma una cadena en un valor numérico único.
def generar_hash_kr(texto):
    hashval = 0
    for char in texto:
        # Algoritmo K&R: hash = hash * 31 + c
        hashval = ord(char) + 31 * hashval
        # Python maneja enteros infinitos. Para que esto se comporte
            como un hash
        # de tamaño fijo y haga "overflow", aplicamos una máscara de
            64 bits.
        hashval = hashval & 0xFFFFFFFFFFFFFFFF

    return hashval

if __name__ == "__main__":
    try:
        entrada = input("Introduce un texto para generar su Hash: ")
        if not entrada:
            print("Advertencia: Has introducido una cadena vacía.")

        hash_value = generar_hash_kr(entrada)
        print(f"Entrada: {entrada}")
        print(f"Hash calculado (Hex): {hash_value:X}")
        print(f"Hash calculado (Dec): {hash_value}")

    except Exception as e:
        print(f"Error: {e}")
```

6.10. Generador de Contraseñas Aleatorias Seguras

Este ejercicio propone crear un programa que genere contraseñas seguras de una longitud dada por el usuario. La contraseña se generará seleccionando aleatoriamente elementos de un conjunto de caracteres (*charset*) que combina letras mayúsculas, minúsculas, números y símbolos. Para asegurar la calidad y la imprevisibilidad de la secuencia, haremos uso del módulo `secrets`. El programa debe garantizar que todos los caracteres tengan la misma probabilidad de ser elegidos, evitando sesgos que debilen la seguridad de la clave generada. En Python, el módulo estándar random es pseudo-aleatorio y determinista (basado en el algoritmo Mersenne Twister). Si alguien logra deducir el estado interno del generador, podría predecir las siguientes contraseñas. Para criptografía y contraseñas, Python ofrece el módulo secrets, que actúa como un Generador de Números Pseudoaleatorios Criptográficamente Seguro (CSPRNG).

```
# fichero passgen.py
# Generador de contraseñas aleatorias seguras
# Utiliza el módulo 'secrets' (CSPRNG).
import secrets
import string
# Definimos el conjunto de caracteres permitidos
CHARSET = string.ascii_letters + string.digits + "!@#$%^&*()"
def generar_contrasena(longitud):
    # secrets.choice(CHARSET) elige un carácter de forma segura
    # Repetimos el proceso 'longitud' veces y unimos en una cadena
    return "".join(secrets.choice(CHARSET) for _ in range(longitud))
if __name__ == "__main__":
    try:
        entrada = input("Introduce la longitud deseada: ")
        longitud = int(entrada)
        if longitud <= 0:
            print("Error: La longitud debe ser mayor que 0.")
        else:
            print("Generando contraseña...", end=" ")
            password = generar_contrasena(longitud)
            print(f"\n{password}")

    except ValueError:
        print("\nError: Debes introducir un número entero.")
```

Capítulo 7

Procesamiento básico de información visual con OpenCV

En este capítulo vamos a presentar una serie de ejercicios básicos sobre procesamiento de imágenes utilizando la biblioteca OpenCV. El objetivo principal es que el alumnado pueda afianzar de forma progresiva los contenidos vistos en las clases teóricas mediante la resolución de problemas prácticos, claramente estructurados y con dificultad creciente.

A través de estos ejercicios, el estudiante aprenderá a realizar operaciones fundamentales como cargar, mostrar y analizar imágenes, así como aplicar transformaciones clásicas de preprocesamiento: conversión a escala de grises, filtrados, recortes, rotaciones, ajustes de brillo y contraste, entre otras. Además, se introducirán tareas esenciales para comprender el funcionamiento interno de las imágenes digitales, tales como la manipulación de canales de color, el trabajo con regiones de interés (ROI), la lectura de metadatos o la conversión entre distintos espacios de color. Por último, veremos algunos de los filtros y detectores de bordes más destacados de la literatura. [4].

7.1. Leer, mostrar e imprimir información sobre una imagen

En este ejercicio se introduce el proceso fundamental de carga y visualización de imágenes utilizando OpenCV. Además, se muestra cómo acceder a información básica como dimensiones, tipo de datos y número de canales, lo cual es esencial para cualquier tarea posterior de procesamiento.

```
#leermostrar.py
import cv2
import matplotlib.pyplot as plt
# Leer la imagen
img = cv2.imread('imagen.jpg')
# Mostrar la imagen
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
# Imprimir información sobre la imagen
print(f'Tamaño de la imagen: {img.shape}')
```

7.2. Convertir una imagen a escala de grises

Aquí se practica la conversión de una imagen a escala de grises, una operación muy común que reduce la complejidad del procesamiento al trabajar con un único canal de intensidad en lugar de tres canales de color.

```
#convertiragrises.py
import cv2
import matplotlib.pyplot as plt

# Leer la imagen original
img = cv2.imread('imagen.jpg')

#  Convertir a escala de grises
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Configurar el lienzo para 2 imágenes una al lado de la otra
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
```

```
# Convertir BGR de OpenCV a RGB de Matplotlib
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original (Color)')
ax1.axis('off')

ax2.imshow(gray_img, cmap='gray')
ax2.set_title('Escala de Grises')
ax2.axis('off')

plt.tight_layout()
plt.show()
```

7.3. Rotar una imagen

Este ejercicio muestra cómo aplicar una rotación simple a una imagen alrededor de su centro, empleando funciones de transformación geométrica de OpenCV.

```
#rotar.py
import cv2
import matplotlib.pyplot as plt

# Leer la imagen
img = cv2.imread('imagen.jpg')
# Obtener las dimensiones de la imagen
(h, w) = img.shape[:2]

# Calcular el centro de la imagen
center = (w // 2, h // 2)

# Generar la matriz de rotación
M = cv2.getRotationMatrix2D(center, 90, 1.0)
# Realizar la rotación
rotated_img = cv2.warpAffine(img, M, (w, h))
# Mostrar la imagen rotada
plt.imshow(cv2.cvtColor(rotated_img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.4. Invertir colores de una imagen

Aquí se realiza la inversión de los valores de color de cada píxel, produciendo un efecto negativo. Este tipo de manipulación permite comprender operaciones punto a punto en imágenes.

```
#invertircolores.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
# La función bitwise_not invierte cada bit de cada píxel.
img_invertida = cv2.bitwise_not(img)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(img_invertida, cv2.COLOR_BGR2RGB))
ax2.set_title('Invertida (Negativo)')
ax2.axis('off')

plt.tight_layout()
plt.show()
```

7.5. Ajustar el brillo y contraste de una imagen

En este ejercicio se modifican los niveles de brillo y contraste aplicando transformaciones lineales sobre los píxeles. Es una operación esencial para mejorar la calidad visual o preparar imágenes para análisis.

```
#ajustarbrillo.py
import cv2
import matplotlib.pyplot as plt
# Cargar la imagen
img = cv2.imread('imagen.jpg')

# Ajustar el brillo (beta) y contraste (alpha)
# Usando la fórmula: g(x) = alpha * f(x) + beta
alpha = 1.5
beta = 50
img_ajustada = cv2.convertScaleAbs(img, alpha=alpha, beta=beta)

# Creamos una figura con 1 fila y 2 columnas
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# Imagen 1: Original
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

# Imagen 2: Ajustada
ax2.imshow(cv2.cvtColor(img_ajustada, cv2.COLOR_BGR2RGB))
ax2.set_title(f'Ajustada (alpha={alpha}, beta={beta})')
ax2.axis('off')

plt.tight_layout()
plt.show()
```

7.6. Recortar una región de interés (ROI) en una imagen

Se muestra cómo definir explícitamente una región de interés mediante coordenadas y extraerla. Este ejercicio refuerza el trabajo con índices y la selección precisa de áreas dentro de la imagen.

```
#recortarregioROI.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
#  Definir la región de interés (ROI)
x, y, w, h = 2000, 1400, 700, 700
# 3. Recortar la ROI
roi = img[y:y+h, x:x+w]

img_con_cuadro = img.copy()
cv2.rectangle(img_con_cuadro, (x, y), (x+w, y+h), (0, 255, 0), 5) #
    Cuadro verde

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img_con_cuadro, cv2.COLOR_BGR2RGB))
ax1.set_title('Imagen Original (Zona marcada)')
ax1.axis('on')

ax2.imshow(cv2.cvtColor(roi, cv2.COLOR_BGR2RGB))
ax2.set_title(f'ROI: {w}x{h} píxeles')
ax2.axis('off')

plt.tight_layout()
plt.show()
```

7.7. Reflejar una imagen horizontal o verticalmente

Este ejercicio enseña a aplicar reflexiones o espejados horizontales y verticales, operaciones útiles en aumentación de datos y en análisis de simetrías.

```
#reflejarimagen.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')

img_reflejada_h = cv2.flip(img, 1)
img_reflejada_v = cv2.flip(img, 0)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(img_reflejada_h, cv2.COLOR_BGR2RGB))
ax2.set_title('Horizontal (flipCode=1)')
ax2.axis('off')

ax3.imshow(cv2.cvtColor(img_reflejada_v, cv2.COLOR_BGR2RGB))
ax3.set_title('Vertical (flipCode=0)')
ax3.axis('off')

plt.tight_layout()
plt.show()
```

7.8. Cargar y mostrar información de metadatos de una imagen

En este apartado se muestra cómo cargar una imagen y extraer información EXIF y otros metadatos relevantes, tales como el modelo de cámara, fecha de captura o parámetros de disparo, cuando están disponibles.

```
from PIL import Image, ExifTags
import cv2
import matplotlib.pyplot as plt
# Usaremos PIL (necesario para metadatos EXIF)
imagen_pil = Image.open(ruta_imagen)
print(f"Formato: {imagen_pil.format}")
print(f"Tamaño (ancho x alto): {imagen_pil.size}")
print(f"Modo de color: {imagen_pil.mode}")
exif_data = imagen_pil._getexif()
print("Información mediante librería PIL ")
if exif_data is not None:
    for etiqueta, valor in exif_data.items():
            nombre = ExifTags.TAGS.get(etiqueta, etiqueta)
            print(f"{nombre}: {valor}")
else:
    print("La imagen no contiene metadatos EXIF.")

imagen_cv = cv2.imread(ruta_imagen)
if imagen_cv is not None:
    alto, ancho, canales = imagen_cv.shape
    print("Información mediante librería OpenCV")
    print(f"Resolución: {ancho} x {alto} píxeles")
    print(f"Número de canales: {canales}")
    print(f"Tipo de datos: {imagen_cv.dtype}")
else:
    print("\nOpenCV no pudo cargar la imagen.")
plt.imshow(cv2.cvtColor(imagen_cv, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

7.9. Aplicar un efecto de desenfoque a una imagen

En este ejercicio se aplican técnicas de desenfoque básico, permitiendo comprender cómo la convolución suaviza la imagen y reduce detalles o ruido.

```
#desenfoque.py
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('imagen.jpg')
kernel_size = (5, 5)
sigma = 1.4
img_desenfocada = cv2.GaussianBlur(img, kernel_size, sigma)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original (Nítida)')
ax1.axis('off')
ax2.imshow(cv2.cvtColor(img_desenfocada, cv2.COLOR_BGR2RGB))
ax2.set_title(f'Desenfoque (Sigma={sigma})')
ax2.axis('off')
plt.tight_layout()
plt.show()
```

7.10. Calcular e imprimir el histograma en escala de grises.

El primer paso para comprender la composición de una imagen es analizar su distribución de luminancia. En esta sección se presenta un script que transforma una imagen a escala de grises y genera un histograma unidimensional. Este gráfico permite observar la frecuencia de aparición de cada nivel de intensidad, desde el negro absoluto (0) hasta el blanco puro (255).

```
#histogramagris.py
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('imagen.jpg')
gris = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#  Calcular el histograma
hist = cv2.calcHist([gris], [0], None, [256], [0, 256])
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
```

```
ax1.imshow(gris, cmap='gray')
ax1.set_title('Imagen en Grises')
ax1.axis('off')
ax2.plot(hist, color='black')
ax2.set_title('Histograma de Intensidad')
ax2.set_xlim([0, 256])
ax2.grid(True)
plt.tight_layout()
plt.show()
```

7.11. Calcular e imprimir los histogramas de cada canal de color

En las imágenes en color, la información se divide habitualmente en tres canales: Rojo, Verde y Azul (RGB). En este apartado, desarrollaremos un código capaz de descomponer la imagen original en sus componentes básicos para visualizar sus respectivos histogramas. Esta técnica es vital para identificar sesgos cromáticos o dominantes de color en una fotografía.

```
#histogramacolores.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagen.jpg')
# Separar en canales
chans = cv2.split(img)

colors = ("b", "g", "r")
plt.figure()
plt.title("Histograma de Color")
plt.xlabel("Bins")
plt.ylabel("# of Pixels")
# Crear un histograma por cada canal
for (chan, color) in zip(chans, colors):
    hist = cv2.calcHist([chan], [0], None, [256], [0, 256])
    plt.plot(hist, color=color)
    plt.xlim([0, 256])

plt.show()
```

7.12. Detección de bordes con el Operador Sobel

El operador Sobel [4] calcula las derivadas de la imagen en los ejes X e Y. Es fundamental para encontrar contornos verticales y horizontales de forma independiente.

```
#filtro_sobel.py
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagenbordes.jpg', 0)

sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)
sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)

sobel_combined = cv2.magnitude(sobelx, sobely)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(sobel_combined, cmap='gray')
ax2.set_title('Detección de Bordes Sobel')
ax2.axis('off')

plt.tight_layout()
plt.show()
```

7.13. Filtro de Relieve (Emboss)

El filtro Emboss o de Relieve [4], resalta las transiciones de intensidad de forma direccional, creando un efecto visual de profundidad que simula el relieve en una superficie física.

```
#filtro_emboss.py
import cv2
import matplotlib.pyplot as plt
import numpy as np

img = cv2.imread('imagen.jpg', 0)

# Definición del kernel de Emboss (Realce/Relieve)
kernel = np.array([[-2,-1,0],
                   [-1, 1,1],
                   [ 0, 1,2]])

emboss = cv2.filter2D(img, -1, kernel) + 128

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(emboss, cmap='gray')
ax2.set_title('Efecto Emboss (Relieve)')
ax2.axis('off')

plt.tight_layout()
plt.show()
```

7.14. Filtro Bilateral: Suavizado con preservación de bordes

A diferencia de otros filtros, el filtradobilateral [4] considera la diferencia de intensidad cromática, permitiendo suavizar zonas planas mientras mantiene los bordes perfectamente nítidos.

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagenbordes.jpg', 0)

bilateral = cv2.bilateralFilter(img, 35, 75, 75)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(img, cmap='gray')
ax1.set_title('Original (Gris)')
ax1.axis('off')

ax2.imshow(bilateral, cmap='gray')
ax2.set_title('Bilateral (Bordes preservados)')
ax2.axis('off')

plt.tight_layout()
plt.show()
```

7.15. Eliminación de ruido "Sal y Pimienta"(Filtro de Mediana)

El filtro de mediana [4] es un filtro no lineal excelente para eliminar ruido impulsivo. Selecciona el valor central de la vecindad tras ordenarlos, eliminando píxeles aberrantes sin emborronar la imagen.

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('imagenbordes.jpg',0)

median = cv2.medianBlur(img, 35)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

ax1.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax1.set_title('Original')
ax1.axis('off')

ax2.imshow(cv2.cvtColor(median, cv2.COLOR_BGR2RGB))
ax2.set_title('Filtro de Mediana (Sal y Pimienta)')
ax2.axis('off')

plt.tight_layout()
plt.show()
```

Capítulo 8

Introducción al Aprendizaje Automático con Scikit-Learn

En este capítulo nos adentraremos en los fundamentos del Aprendizaje Automático (*Machine Learning*) utilizando el ecosistema de librerías de Python más extendido en la industria y la academia: `scikit-learn` para el modelado y `pandas` para la manipulación de datos.

El objetivo es que el alumnado comprenda el flujo de trabajo estándar en un proyecto de datos: carga y exploración, preprocesamiento, división de conjuntos de datos, entrenamiento de modelos (tanto supervisados como no supervisados) y evaluación de resultados. Para ello, nos hemos basado en algunos de los ejemplos presentes en el libro Hans-On Machine Learning with Scikit-learn, Keras, and TensorFlow de Aurélien Géron [3].

A través de estos ejercicios, pasaremos de manejar estructuras de datos tabulares a entrenar nuestros primeros algoritmos predictivos, sentando las bases para problemas más complejos.

8.1. Carga y exploración básica de un Dataset

El siguiente ejercicio nos muestra cómo cargar un dataset para trabajar con él y realizar una exploración básica del mismo. Antes de aplicar cualquier algoritmo, es crucial cargar los datos y entender su estructura. En este ejercicio utilizaremos pandas para cargar un conjunto de datos tabular, visualizar sus primeras filas y obtener una descripción estadística de sus variables.

```
import pandas as pd
from sklearn.datasets import load_iris

# Cargamos un dataset de ejemplo (Iris)
# Scikit-learn nos devuelve un objeto tipo diccionario, pero lo
    convertiremos
# a DataFrame de Pandas para que sea más legible, como un Excel.
datos_iris = load_iris()
df = pd.DataFrame(data=datos_iris.data,
    columns=datos_iris.feature_names)

# Añadimos la columna objetivo (target)
df['especie'] = datos_iris.target

# Mostramos las primeras 5 filas
print("Primeras 5 filas del dataset ")
print(df.head())

# Obtenemos información estadística básica
print("\n Descripción estadística del dataset")
print(df.describe())

# Verificamos tipos de datos y si hay nulos
print("\n Información del DataFrame ")
print(df.info())
```

8.1. Carga y exploración básica de un Dataset

Además de los dataset que proporciona scikit-learn, podemos cargar cualquier dataset desde un csv que dispongamos. Seguidamente veremos un ejemplo.

```
import pandas as pd

# Carga de un archivo CSV
# Supongamos que tenemos un archivo llamado 'viviendas.csv' en la
    misma carpeta
# Si no tienes uno, puedes usar esta URL de un dataset real de
    viviendas:
url = "viviendas.csv"
#"https://raw.githubusercontent.com/ageron/
# handson-ml2/master/datasets/housing/housing.csv"

try:
    # Leemos el archivo CSV
    df = pd.DataFrame()
    df = pd.read_csv(url)  #Puede ser una ruta local o una url

    # Exploración inicial
    print(" Dimensiones del dataset (Filas, Columnas) ")
    print(df.shape)

    print("\n Listado de columnas")
    print(df.columns.tolist())
    #  Mostrar las primeras filas
    print("\nPrimeras filas del dataset de viviendas ")
    print(df.head())
    # Conteo de valores en una variable categórica
    # Esto es útil para saber cuántas casas hay cerca del mar, en el
       interior, etc.
    print("\n Distribución por cercanía al océano ")
    print(df['ocean_proximity'].value_counts())

    # Guardar una copia local si se desea
    df.to_csv("copia_viviendas.csv", index=False)
except Exception as e:
    print(f"Error al cargar el archivo: {e}")
```

8.2. División del conjunto de datos (Train/Test Split)

Este ejercicio propone mostrar cómo realizar la típica división del conjunto de datos para poder trabajar con él en ciencia de datos. Para evaluar correctamente un modelo, nunca debemos probarlo con los mismos datos con los que fue entrenado. Aquí aprenderemos a utilizar la función `train_test_split` para separar nuestros datos en un subconjunto de entrenamiento y otro de prueba.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Cargar datos
iris = load_iris()
X = iris.data  # Características (largo de pétalo, ancho, etc.)
y = iris.target # Etiquetas (0, 1, 2 que representan las especies)

# Dividir en entrenamiento (80%) y prueba (20%)
# random_state=42 asegura que la división sea siempre la misma al
    ejecutar
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print(f"Total de datos: {X.shape[0]}")
print(f"Datos de entrenamiento: {X_train.shape[0]}")
print(f"Datos de prueba: {X_test.shape[0]}")y
```

8.3. Preprocesamiento: Estandarización de datos

Este ejercicio nos muestra un pequeño ejemplo de preprocesamiento de datos, el cual se suele realizar a los dataset para poder trabajar con ello. Muchos algoritmos de Machine Learning no funcionan correctamente si las características numéricas tienen escalas muy diferentes. En este ejercicio aplicaremos el `StandardScaler` para normalizar los datos, haciendo que tengan media 0 y desviación estándar 1.

```
from sklearn.preprocessing import StandardScaler
import numpy as np
# Simulamos datos con escalas muy diferentes
# Ejemplo: [Edad (0-100), Salario (1000-50000)]
datos = np.array([
    [25, 30000],
    [50, 45000],
    [20, 15000],
    [35, 32000]
])
print("Datos originales:\n", datos)
# Inicializar el escalador
scaler = StandardScaler()
# Ajustar (calcular media y desviación) y transformar
datos_escalados = scaler.fit_transform(datos)
print("\nDatos estandarizados (Media~0, Var~1):\n", datos_escalados)
```

8.4. Regresión Lineal Simple

Este ejercicio muestra un ejemplo de cómo entrenar un modelo simple de Regresión Lineal para predecir un valor numérico a partir de una variable de entrada, visualizando la recta de regresión resultante.

```
import numpy as np
from sklearn.linear_model import LinearRegression
# Datos de ejemplo simple: y = 2x + 1
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([3, 5, 7, 9, 11])
# Crear y entrenar el modelo
modelo = LinearRegression()
modelo.fit(X, y)
# Realizar una predicción para un valor nuevo (x = 10)
nuevo_x = np.array([[10]])
prediccion = modelo.predict(nuevo_x)
print(f"Coeficiente (pendiente): {modelo.coef_[0]}")
print(f"Intercepto (ordenada): {modelo.intercept_}")
print(f"Predicción para x=10: {prediccion[0]}")
```

8.5. Clasificación con Árboles de Decisión

En este ejercicio vamos a implementar sobre el dataset iris un modelo de clasificación basado en árboles de decisión. En problemas de clasificación, el objetivo es predecir una etiqueta o categoría [3].

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

# Carga y Split
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=42)

# Crear el modelo (Árbol de decisión)
# max_depth=3 limita la profundidad para evitar sobreajuste simple
arbol = DecisionTreeClassifier(max_depth=3, random_state=42)

# Entrenar
arbol.fit(X_train, y_train)

# Predecir sobre el conjunto de test
predicciones = arbol.predict(X_test)

print("Etiquetas reales (Test):   ", y_test[:10])
print("Etiquetas predichas:       ", predicciones[:10])
```

8.6. Evaluación de modelos: Matriz de Confusión y Accuracy

Este ejercicio nos presenta cómo generar una matriz de confusión y obtener la accuray del modelo, ya que entrenar un modelo no es suficiente; debemos medir qué tan bueno es.

```
from sklearn.metrics import accuracy_score, confusion_matrix
# (Asumimos que las variables y_test y predicciones vienen del
    ejercicio anterior)
# Para este ejemplo, creamos datos simulados:
y_real      = [0, 1, 1, 0, 2, 2, 1, 0]
```

```
y_predichos = [0, 1, 0, 0, 2, 1, 1, 0] # Algunos errores intencionados

# Calcular Accuracy (Exactitud)
acc = accuracy_score(y_real, y_predichos)
print(f"Exactitud del modelo: {acc:.2f} ({acc*100}%)")

# Matriz de confusión
# Muestra: filas (realidad) vs columnas (predicción)
matriz = confusion_matrix(y_real, y_predichos)
print("\nMatriz de Confusión:\n", matriz)
```

8.7. Aprendizaje No Supervisado: Clustering con K-Means

En este ejercicio vamos a aprender a realizar un proceso de aprendizaje no supervisado basado en Clustering (K-Means). A diferencia de los ejercicios anteriores, aquí trabajaremos sin etiquetas predefinidas. Usaremos el algoritmo K-Means para agrupar datos automáticamente basándonos en su similitud geométrica, una técnica fundamental para descubrir patrones ocultos.

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Generar datos sintéticos (nubes de puntos)
X, _ = make_blobs(n_samples=100, centers=3, cluster_std=0.60,
    random_state=0)

# Configurar K-Means para buscar 3 grupos (clusters)
kmeans = KMeans(n_clusters=3, n_init=10)
# Ajustar el modelo (aquí no hay 'y' porque es no supervisado)
kmeans.fit(X)
# Obtener los centros de los grupos y las etiquetas asignadas
centroides = kmeans.cluster_centers_
etiquetas = kmeans.labels_

print("Coordenadas de los centroides hallados:\n", centroides)
print("\nEtiquetas asignadas a los primeros 5 puntos:", etiquetas[:5])
```

8.8. Validación Cruzada (Cross-Validation)

Este ejercicio nos propone realizar un comprobación de mi modelo utilizando Validación Cruzada. Hasta ahora hemos dividido los datos una sola vez en entrenamiento y prueba. Sin embargo, esto puede llevar a resultados engañosos si tenemos suerte (o mala suerte) con la partición elegida. La validación cruzada (K-Fold) divide los datos en K partes y entrena/evalúa el modelo K veces, asegurando una métrica de rendimiento mucho más robusta y fiable.

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score

# Cargar datos
X, y = load_iris(return_X_y=True)

# Definir el modelo
# Usamos KNN (K-Vecinos más cercanos) como ejemplo

knn = KNeighborsClassifier(n_neighbors=5)
# Aplicar Validación Cruzada (Cross Validation) con 5 pliegues (folds)
# Esto divide los datos en 5 partes: entrena con 4 y prueba con 1,
   rotando 5 veces.
scores = cross_val_score(knn, X, y, cv=5, scoring='accuracy')
print("Puntuaciones en cada iteración:", scores)
print(f"Precisión media: {scores.mean():.2f}")
print(f"Desviación estándar: {scores.std():.2f}")
# Una desviación baja indica que el modelo es estable.
```

8.9. Preprocesamiento: Codificación de variables categóricas (One-Hot Encoding)

En este ejercicio vamos a realizar preprocesamiento codificando variables de categorías [3]. La mayoría de algoritmos de Machine Learning operan exclusivamente con números. Cuando tenemos variables cualitativas (como Çolor": Rojo, Verde, Azul), no debemos asignarles simplemente 1, 2 y 3, ya que el algoritmo podría interpretar erróneamente un orden matemático. La técnica *One-Hot Encoding* [3] convierte estas categorías en columnas binarias independientes.

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# Crear datos de ejemplo con una variable categórica
df = pd.DataFrame({
    'ID': [1, 2, 3, 4],
    'Ciudad': ['Madrid', 'Barcelona', 'Madrid', 'Valencia'],
    'Salario': [30000, 32000, 31000, 29000]
})

print("DataFrame Original ")
print(df)

# Inicializar el codificador
# sparse_output=False nos devuelve un array normal (numpy) en vez de
   una matriz dispersa
encoder = OneHotEncoder(sparse_output=False)

# Ajustar y transformar la columna 'Ciudad'
# Necesitamos pasarle un array 2D, por eso usamos las dobles corchetes
   [['Ciudad']]
ciudades_encoded = encoder.fit_transform(df[['Ciudad']])

# Ver el resultado y las categorías encontradas
print("\n Categorías detectadas ")
print(encoder.categories_)

print("\n Matriz One-Hot resultante ")
print(ciudades_encoded)
# Nota: La primera columna será 1 si es Barcelona, la segunda si es
   Madrid, etc.
```

8.10. Persistencia del Modelo: Guardar y Cargar

Este ejercicio nos propone que una vez que hemos entrenado un modelo satisfactorio, el último paso es guardarlo en disco para poder utilizarlo en producción o en otro momento sin necesidad de reentrenarlo. En este ejercicio utilizaremos la librería `joblib`, estándar en el ecosistema de scikit-learn, para serializar nuestro modelo entrenado.

```
import joblib
from sklearn.linear_model import LinearRegression

# Entrenar un modelo simple (ej. Doblar el número)
X = [[1], [2], [3], [4]]
y = [2, 4, 6, 8]

modelo = LinearRegression()
modelo.fit(X, y)

print("Predicción antes de guardar (input 5):", modelo.predict([[5]]))

# Guardar el modelo en un archivo (Persistencia)
nombre_fichero = 'modelo_regresion.pkl'
joblib.dump(modelo, nombre_fichero)
print(f"Modelo guardado en {nombre_fichero}")

#  Simular que estamos en otro script o programa...
# Borramos el modelo de la memoria para probar
del modelo

print("\nCargando modelo desde el disco...")
modelo_cargado = joblib.load(nombre_fichero)

#  Usar el modelo cargado
prediccion = modelo_cargado.predict([[5]])
print("Predicción con el modelo cargado (input 5):", prediccion)
```

Bibliografía

[1] Brian W. Kernighan Dennis M. Ritchie. *El lenguaje de programación C ANSI*. Prentice Hall., 2 edition, 1988.

[2] Merino L. Santos E. *Álgebra Lineal con métodos elementales*. Paraninfo, 1997.

[3] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2nd edition, 2019.

[4] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, fourth edition, 2018.

[5] Stewart James. *Cálculo - Trascendentes Tempranas*. CENGAGE, eight edition, 2018.

[6] Jetbrains. *PyCharm https://www.jetbrains.com/es-es/pycharm/*. 2025.

[7] S. Jiménez Zafra and Montejo Ráez A. *Curso de Programación Python*. Anaya, 2 edition, 2021.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, second edition, 1998.

[9] Mark Lutz. *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media, 5th edition, 2013.

[10] Lucena López Manuel. *Criptografía y Seguridad en computadores*. Universidad de Jaén, fifth edition, 2023.

[11] Eric Matthes. *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*. No Starch Press, 3rd edition, 2023.

[12] Microsoft. *Visual Studio Code https://code.visualstudio.com/*. 2025.

[13] G. Brassard P. Bratley. *Fundamentos de algoritmia*. Prentice Hall., 1998.

[14] Tim Peters. Timsort: An adaptive, stable, natural mergesort. https://github.com/python/cpython/blob/main/Objects/listsort.txt, 2002.

[15] Python Software Foundation. Python version 1.0 release. https://www.python.org/download/releases/1.0/, January 1994.

[16] Python Software Foundation. *Python 3.13.0 Documentation*. Python Software Foundation, 2024.

[17] Python Software Foundation. *Python 3.14.0 Documentation*. Python Software Foundation, 2025.

[18] Luciano Ramalho. *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media, 2nd edition, 2022.

[19] Johnson Richard. *Sorting Algoritthms and Techniques: Definitive Referente for Developers and Engineers*. AV, first edition, 2025.

[20] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 2nd edition, 1996.

[21] Douglas R. Stinson and Maura B. Paterson. *Cryptography: Theory and Practice*. CRC Press, 4th edition, 2018.

[22] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4 edition, 2013.

[23] Guido Van Rossum. Foreword for "programming python"(1st ed.). https://www.python.org/doc/essays/foreword/, 1996.

[24] Guido Van Rossum. The making of python: A conversation with guido van rossum. *Artima Developer*, 2003.

[25] Kahan W. *Miscalculating Area and Angles of Needle-like Triangle*. Berkeley, https://people.eecs.berkeley.edu/ wkahan/triangle.pdf edition, 2014.

[26] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C*. Person, second edition, 2003.