

Fundamentos de Programación y Ciencia de Datos con Python3

Ricardo J. Rodríguez

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

27 de junio de 2026



Distribuido bajo licencia CC BY-NC-SA. (© Ricardo J. Rodríguez)
<https://creativecommons.org/licenses/by-nc-sa/4.0/es/>

1 de 448

*Dímelo y lo olvido, enséñame y lo recuerdo,
involúcrame y lo aprendo.*
(proverbio chino)

*Todos los modelos son erróneos,
pero algunos son útiles.*
(GEORGE E. P. BOX)

Índice general

Índice de figuras	XI
Índice de tablas	XIII
Índice de códigos	XIX
Prefacio	XXI
1 Instalación del entorno de trabajo: Jupyter Notebook (en Docker) y Visual Studio Code	1
1.1 ¿Por qué un entorno aislado en Docker?	1
1.2 Instalación de Docker	2
1.3 Configuración del entorno: Dockerfile y dependencias	2
1.3.1 Dependencias adicionales	4
1.4 Construcción de la imagen	5
1.5 Ejecución del contenedor	5
1.5.1 Desde Docker Desktop	5
1.5.2 Desde la terminal	6
1.6 Acceso a Jupyter desde el navegador	7
1.6.1 Conectar al servidor	8
1.6.2 Crear un notebook y elegir kernel	8
1.6.3 Probar el entorno	9
1.7 Visual Studio Code como entorno alternativo	10
1.7.1 Instalación	10
1.7.2 Conectar VS Code al servidor Jupyter del contenedor	11
2 Introducción a la programación y resolución automática de problemas con Python	15
2.1 Programación y resolución automática de problemas	15
2.2 Propiedades de un algoritmo	17
2.3 Estructura de un programa en Python	17
2.4 Desarrollo y ejecución de un programa en Python	18
2.5 Guía de estilo en la programación	19
2.6 Sintaxis básica de Python: primeros pasos	19
2.6.1 Comentarios y <i>docstrings</i>	20
2.6.2 Variables	20
2.6.3 Tipos de datos básicos (introducción)	21

2.6.4	Entrada y salida de datos	22
2.6.5	Operadores aritméticos	23
2.7	Multiparadigma en Python	25
2.8	Ejercicios	25
3	Tipos de datos y operadores	35
3.1	Tipos de datos	35
3.2	Operaciones	36
3.2.1	Operadores de comparación	36
3.2.2	Operadores aritméticos	37
3.2.3	Operadores lógicos	37
3.2.4	Operadores de asignación	38
3.3	Prioridad de operadores	39
3.4	Conversión de tipos	40
3.4.1	Conversión implícita	41
3.4.2	Conversión explícita	41
3.4.3	Conversiones de tipo más comunes	41
3.5	Cadenas formateadas (<i>f-strings</i>)	42
3.6	Manejo de errores en la conversión	43
3.6.1	Captura específica con <code>try / except</code>	43
3.6.2	Captura genérica con <code>except Exception as e</code>	44
3.6.3	Cláusula <code>else</code> : código sin error	44
3.6.4	Cláusula <code>finally</code> : código que siempre se ejecuta	45
3.6.5	Lanzar errores con <code>raise</code>	45
3.7	Ejercicios	46
4	Instrucciones simples y estructuradas	57
4.1	Instrucciones simples	57
4.1.1	Declaraciones y asignaciones	57
4.1.2	Expresiones	58
4.1.3	Entrada y salida de datos	58
4.2	Instrucciones estructuradas	59
4.2.1	Bloques de código: agrupando instrucciones	60
4.2.2	Instrucciones condicionales: tomando decisiones	60
4.2.3	Instrucciones iterativas: repitiendo acciones	63
4.3	Ejercicios	68
5	Diseño de programas elementales en Python	105
5.1	Funciones	105
5.1.1	Definición de funciones	108
5.1.2	Parámetros en la función	108
5.1.3	Llamar a una función	109
5.1.4	La sentencia <code>return</code>	110
5.1.5	Documentar funciones: <i>docstrings</i>	112

5.1.6	Anotaciones de tipo	113
5.1.7	Ámbito de las variables	114
5.2	Comunicación entre funciones	115
5.3	Ejercicios	117
6	Desarrollo modular y diseño descendente	141
6.1	Diseño descendente	142
6.2	Uso de <code>import</code> : módulos y paquetes	142
6.2.1	Importar un módulo completo	142
6.2.2	Importar nombres concretos	143
6.2.3	Alias al importar	143
6.2.4	Dónde busca Python los módulos: <code>sys.path</code>	143
6.2.5	Paquetes: módulos agrupados en directorios	144
6.3	Caso práctico: programa dirigido por menú	144
6.3.1	Análisis del problema	145
6.3.2	Módulo de funciones de cálculo	145
6.3.3	Módulo principal	148
6.3.4	Niveles de abstracción	150
6.4	Ejercicios	151
7	Enteros	171
7.1	Representación de números enteros en Python	171
7.1.1	Conversiones entre bases	172
7.2	Operadores aritméticos	172
7.3	Desbordamientos	173
7.4	El tipo <code>bool</code> y su relación con los enteros	174
7.5	Bibliotecas numéricas: el módulo <code>math</code>	174
7.6	Manipulación de números enteros	175
7.6.1	Contar el número de cifras de un entero	175
7.6.2	Suma de las cifras de un número entero	175
7.6.3	Obtener la <i>i</i> -ésima cifra menos significativa	176
7.6.4	Invertir un número (imagen especular)	177
7.6.5	Comprobar la primalidad de un número	178
7.6.6	Cálculo del máximo común divisor	180
7.7	Aplicación: representación binaria de enteros	180
7.7.1	Bits, bytes y enteros: <code>bit_length</code> , <code>to_bytes</code> , <code>from_bytes</code>	181
7.8	Ejercicios	181
8	Flotantes (números reales)	193
8.1	Representación de números reales en Python	193
8.1.1	Información sobre la representación: <code>sys.float_info</code>	194
8.2	Comparación de flotantes: <code>math.isclose</code>	194
8.3	Valores especiales: <code>inf</code> y <code>nan</code>	195
8.3.1	Infinito (<code>inf</code>)	195

8.3.2	No es un número (nan)	195
8.4	Operadores aritméticos sobre flotantes	196
8.5	Conversiones y precisión exacta	196
8.5.1	Conversión entre tipos	196
8.5.2	Precisión exacta: Decimal y Fraction	197
8.5.3	Precisión en numpy: una mirada al cap. 12	198
8.6	Cálculos matemáticos: la biblioteca math	198
8.6.1	Funciones trigonométricas	198
8.6.2	Funciones exponenciales y logarítmicas	198
8.6.3	Cálculo de raíces y potencias	199
8.6.4	Redondeo y truncamiento	199
8.6.5	Valor absoluto y resto	200
8.6.6	Constantes matemáticas	200
8.7	Ejemplos	200
8.7.1	Cálculo de propiedades de un polígono regular	200
8.7.2	Resolución de una ecuación de segundo grado	202
8.7.3	Cálculo de series numéricas	203
8.8	Ejercicios	206
9	Booleanos y cadenas	237
9.1	Booleanos	237
9.1.1	Operadores lógicos	237
9.1.2	Valores <i>truthy</i> y <i>falsy</i>	238
9.1.3	Evaluación cortocircuitada	239
9.2	Cadenas	239
9.2.1	Definición de cadenas	240
9.2.2	Inmutabilidad	240
9.2.3	Indexación y rebanado	241
9.2.4	Operadores sobre cadenas	241
9.2.5	Evaluación implícita en condicionales	241
9.3	Métodos integrados de cadenas	242
9.3.1	Cambio de mayúsculas/minúsculas	242
9.3.2	Eliminación de espacios y caracteres en los extremos	242
9.3.3	Búsqueda y conteo	243
9.3.4	Modificación	243
9.3.5	División y unión	243
9.3.6	Predicados sobre el contenido	244
9.4	Formateo de cadenas	244
9.4.1	f-strings	244
9.4.2	El método str.format()	245
9.4.3	Operador % (estilo C)	245
9.5	Bytes y codificación	245
9.5.1	Normalización Unicode	246

9.6	Ejercicios	247
10	Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior	283
10.1	Tipos mutables e inmutables	283
10.2	Listas	284
10.2.1	Creación	285
10.2.2	Indexación y rebanado	285
10.2.3	Operadores y métodos comunes	285
10.2.4	Listas por comprensión	286
10.2.5	Generadores y expresiones generadoras	287
10.2.6	Listas multidimensionales	287
10.2.7	Algoritmos comunes con listas	288
10.2.8	Ejemplos	290
10.3	Tuplas	291
10.3.1	Creación	291
10.3.2	Desempaquetado y tuplas anidadas	291
10.3.3	Operaciones e inmutabilidad	292
10.3.4	Tuplas con nombre: <code>namedtuple</code>	292
10.3.5	Devolver varios valores	293
10.4	Conjuntos	293
10.4.1	Creación	293
10.4.2	Operaciones básicas	294
10.4.3	Operaciones de teoría de conjuntos	294
10.4.4	Conjuntos por comprensión	295
10.4.5	Iteración	295
10.4.6	Conjuntos inmutables: <code>frozenset</code>	295
10.5	Diccionarios	295
10.5.1	Creación y uso básico	295
10.5.2	Iteración y vistas	296
10.5.3	Diccionarios anidados	296
10.5.4	Diccionarios por comprensión	297
10.5.5	Copia y referencias	297
10.5.6	El módulo <code>collections</code>	297
10.6	Funciones de primera clase y de orden superior	298
10.6.1	Funciones lambda	299
10.6.2	<code>map</code> , <code>filter</code> , <code>reduce</code>	299
10.6.3	<code>sorted</code> con clave	300
10.7	Aplicación: estructuras de datos en el análisis de registros	300
10.8	Ejercicios	303
11	Entrada y salida de datos. Ficheros	337
11.1	Entrada y salida estándar	337
11.2	La función <code>eval</code> y la alternativa segura	338

11.3	Apertura de ficheros: <code>open</code> , modos y codificación	339
11.3.1	Codificación: el parámetro <code>encoding</code>	339
11.4	Rutas: <code>pathlib</code> y <code>os.path</code>	339
11.5	Ficheros de texto	340
11.5.1	Lectura	340
11.5.2	Escritura	340
11.5.3	El bloque <code>with</code> y el cierre automático	341
11.5.4	Manejo de errores	341
11.6	Formatos comunes: JSON y CSV	341
11.6.1	JSON	341
11.6.2	CSV y TSV con <code>csv</code>	342
11.6.3	Ejemplo: gestión de contactos	343
11.7	Ficheros binarios	345
11.7.1	Lectura y escritura básicas	345
11.7.2	Empaquetado con <code>struct</code>	346
11.7.3	Hashes y resúmenes con <code>hashlib</code>	346
11.7.4	El módulo <code>pickle</code>	347
11.8	Robustez y buenas prácticas al manejar ficheros	348
11.8.1	Path traversal	348
11.8.2	Escritura atómica	349
11.8.3	Ficheros temporales seguros	349
11.8.4	Permisos al escribir secretos	349
11.9	Ejemplo: almacenamiento binario de usuarios	350
11.10	Ejercicios	353
12	Introducción a numpy y pandas	381
12.1	Biblioteca <code>numpy</code>	381
12.1.1	El objeto <code>ndarray</code>	382
12.1.2	Tipos de datos (<code>dtype</code>)	382
12.1.3	Operaciones y <i>broadcasting</i>	383
12.1.4	Indexación, rebanado y máscaras	383
12.1.5	Reorganización: <code>reshape</code> , <code>transpose</code> , <code>flatten</code>	384
12.1.6	Funciones vectorizadas útiles	384
12.1.7	Eficiencia: vistas vs copias	385
12.1.8	Generación de números pseudoaleatorios	386
12.1.9	Interoperabilidad con bytes y ficheros binarios	386
12.1.10	Otros tipos: complejos, fechas, estructurados	387
12.2	Biblioteca <code>pandas</code>	388
12.2.1	El tipo <code>Series</code>	388
12.2.2	El tipo <code>DataFrame</code>	390
12.3	Aplicación: análisis de registros de actividad	395
12.3.1	Detección de anomalías por z-score	396
12.3.2	Top de IPs en un fichero de logs	396

12.3.3	Eventos por hora con <code>resample</code>	396
12.3.4	Cruzar IPs con una lista negra	397
12.3.5	Análisis de frecuencias para criptoanálisis clásico	397
12.4	Ejercicios	398
13	Visualización de datos con <code>matplotlib</code> y <code>seaborn</code>	401
13.1	Biblioteca <code>matplotlib</code>	401
13.1.1	Anatomía de una figura	402
13.1.2	Primer gráfico: una línea	402
13.1.3	Tipos de gráfico habituales	403
13.1.4	Múltiples subgráficos	404
13.1.5	Personalización: ejes, leyenda, anotaciones	404
13.1.6	Guardar la figura	405
13.2	Biblioteca <code>seaborn</code>	405
13.2.1	Filosofía y datos en formato largo	405
13.2.2	Gráficos estadísticos esenciales	405
13.2.3	Estilos y paletas	406
13.3	Aplicación: visualización de registros de actividad	406
13.3.1	Distribución de tamaños de paquete	407
13.3.2	Mapa de calor de actividad por hora y día	407
13.3.3	Serie temporal con umbrales	407
13.3.4	Matriz de correlación de <i>features</i>	408
13.4	Ejercicios	408
14	Introducción al aprendizaje automático con <code>scikit-learn</code>	421
14.1	Conceptos fundamentales	421
14.1.1	Aprendizaje supervisado y no supervisado	421
14.1.2	Variables, <i>features</i> y vectores de datos	422
14.1.3	Entrenamiento, validación y prueba	422
14.1.4	Métricas de evaluación	422
14.2	La API de <code>scikit-learn</code>	423
14.2.1	Un ejemplo mínimo de clasificación	423
14.2.2	Preprocesado: <code>StandardScaler</code> y <code>LabelEncoder</code>	424
14.2.3	Pipelines	425
14.2.4	Validación cruzada	425
14.2.5	Búsqueda de hiperparámetros: <code>GridSearchCV</code>	425
14.3	Algoritmos clásicos	426
14.3.1	Clasificación	426
14.3.2	Regresión	426
14.3.3	<i>Clustering</i>	427
14.3.4	Detección de anomalías	427
14.3.5	Reducción de dimensionalidad	427
14.4	Aplicación: detección de <i>spam</i>	427
14.5	Aplicación: clasificación de tráfico de red	428

Índice general

14.6 Aplicación: detección de anomalías sin etiquetas	429
14.7 Cierre del libro	430
14.8 Ejercicios	431
Epílogo	447

Índice de figuras

1	Arquitectura del entorno de trabajo. El navegador o Visual Studio Code, instalados en la máquina anfitriona, se comunican con el servidor Jupyter que corre dentro del contenedor Docker mediante el puerto 8888.	2
2	Ventana de inicio de Docker Desktop	3
3	Docker Desktop mostrando la imagen fundamentos-python creada anteriormente.	6
4	Configuración del contenedor de la imagen fundamentos-python en Docker Desktop	7
5	Pantalla de <i>log</i> en Docker Desktop tras arrancar el contenedor: muestra la URL con el <i>token</i> para acceder a Jupyter.	8
6	Interfaz principal de Jupyter al acceder desde la máquina anfitriona.	9
7	Creación de un nuevo <i>notebook</i> en Jupyter.	9
8	Selección de <i>kernel</i> en Jupyter.	10
9	Ejecución del primer programa Python en Jupyter desde el navegador.	11
10	Pestaña Extensions de VS Code en la barra lateral izquierda.	12
11	Instalación de la extensión Jupyter en VS Code.	12
12	Selector de <i>kernel</i> en la parte derecha de la barra superior del fichero « <i>.ipynb</i> » en VS Code.	13
13	Sugerencia de VS Code para instalar la extensión de Python.	13
14	Conexión a un servidor Jupyter existente desde VS Code.	13
15	Selección del <i>kernel</i> en el servidor Jupyter remoto.	13
16	Ejecución del primer programa Python desde un <i>notebook</i> en VS Code conectado al contenedor.	14
1	Interacción con el usuario desde Jupyter Notebook en VS Code.	23

Índice de tablas

1	Tipos de datos más básicos en Python.	35
2	Operadores de comparación en Python.	37
3	Operadores aritméticos en Python.	37
4	Operadores lógicos en Python.	38
5	Operadores de asignación en Python.	38
6	Prioridad de los operadores básicos de Python (de mayor a menor).	40
1	Operadores aritméticos sobre enteros.	172
1	Operadores aritméticos sobre flotantes.	196
1	Operadores lógicos sobre booleanos.	238
2	Valores considerados <i>falsy</i> en Python.	238
3	Operadores y operaciones básicas sobre cadenas.	241
1	Tipos mutables e inmutables más comunes en Python.	284
2	Operaciones de teoría de conjuntos en Python.	294
1	Modos de apertura de ficheros en Python.	339
2	Prefijos de orden de bytes en struct	346
1	Tipos de datos más comunes en numpy	382

Índice de códigos

1.1	Fichero « <code>Dockerfile</code> » para crear el entorno de trabajo del libro.	3
1.2	Fichero « <code>requirements.txt</code> » con las dependencias usadas en el libro. . .	5
2.1	Solución al Ejercicio 2.1.	26
2.2	Solución al Ejercicio 2.2.	27
2.3	Solución al Ejercicio 2.3.	29
2.4	Solución al Ejercicio 2.4.	31
2.5	Solución al Ejercicio 2.5.	33
3.1	Solución al Ejercicio 3.1.	46
3.2	Solución al Ejercicio 3.2.	47
3.3	Solución al Ejercicio 3.3.	48
3.4	Solución al Ejercicio 3.4.	49
3.5	Solución al Ejercicio 3.5.	50
3.6	Solución al Ejercicio 3.6.	51
3.7	Solución al Ejercicio 3.7.	52
3.8	Solución al Ejercicio 3.8.	53
3.9	Solución al Ejercicio 3.9.	54
3.10	Solución al Ejercicio 3.10.	55
4.1	Solución al Ejercicio 4.1.	68
4.2	Solución al Ejercicio 4.2.	69
4.3	Solución al Ejercicio 4.3.	71
4.4	Solución al Ejercicio 4.4.	72
4.5	Solución al Ejercicio 4.5.	73
4.6	Solución al Ejercicio 4.6.	74
4.7	Solución al Ejercicio 4.7.	75
4.8	Solución al Ejercicio 4.8.	76
4.9	Solución al Ejercicio 4.9.	78
4.10	Solución al Ejercicio 4.10.	80
4.11	Solución al Ejercicio 4.11.	81
4.12	Solución al Ejercicio 4.12.	82
4.13	Solución al Ejercicio 4.13.	83
4.14	Solución al Ejercicio 4.14.	85
4.15	Solución al Ejercicio 4.15.	86
4.16	Solución al Ejercicio 4.16.	87
4.17	Solución al Ejercicio 4.17.	88
4.18	Solución al Ejercicio 4.18.	89

4.19	Solución al Ejercicio 4.19.	91
4.20	Solución al Ejercicio 4.20.	93
4.21	Solución al Ejercicio 4.21.	94
4.22	Solución al Ejercicio 4.22.	95
4.23	Solución al Ejercicio 4.23.	96
4.24	Solución al Ejercicio 4.24.	97
4.25	Solución al Ejercicio 4.25.	98
4.26	Solución al Ejercicio 4.26.	99
4.27	Solución al Ejercicio 4.27.	100
4.28	Solución al Ejercicio 4.28.	101
4.29	Solución al Ejercicio 4.29.	102
4.30	Solución al Ejercicio 4.30.	103
5.1	Solución al Ejercicio 5.1.	118
5.2	Solución al Ejercicio 5.2.	119
5.3	Solución al Ejercicio 5.3.	120
5.4	Solución al Ejercicio 5.4.	121
5.5	Solución al Ejercicio 5.5.	122
5.6	Solución al Ejercicio 5.6.	123
5.7	Solución al Ejercicio 5.7.	124
5.8	Solución al Ejercicio 5.8.	126
5.9	Solución al Ejercicio 5.9.	127
5.10	Solución al Ejercicio 5.10.	128
5.11	Solución al Ejercicio 5.11.	129
5.12	Solución al Ejercicio 5.12.	130
5.13	Solución al Ejercicio 5.13.	131
5.14	Solución al Ejercicio 5.14.	132
5.15	Solución al Ejercicio 5.15.	134
5.16	Solución al Ejercicio 5.16.	135
5.17	Solución al Ejercicio 5.17.	136
5.18	Solución al Ejercicio 5.18.	137
5.19	Solución al Ejercicio 5.19.	138
5.20	Solución al Ejercicio 5.20.	139
6.1	Solución al Ejercicio 6.1.	151
6.2	Solución al Ejercicio 6.2.	152
6.3	Solución al Ejercicio 6.3.	153
6.4	Solución al Ejercicio 6.4.	154
6.5	Solución al Ejercicio 6.5.	156
6.6	Solución al Ejercicio 6.6.	158
6.7	Solución al Ejercicio 6.7.	160
6.8	Solución al Ejercicio 6.7 (continuación).	161
6.9	Solución al Ejercicio 6.8 (archivo « <code>formas_geometricas.py</code> »).	162
6.10	Solución al Ejercicio 6.8 (archivo principal).	163

6.11 Solución al Ejercicio 6.9 (fichero «conversion_temperaturas.py»).	165
6.12 Solución al Ejercicio 6.9 (fichero principal).	166
6.13 Solución al Ejercicio 6.10 (fichero «conversion_unidades.py»).	168
6.14 Solución al Ejercicio 6.10 (fichero principal).	169
src/leccion6/conversion-unidades-main.py	170
7.1 Solución al Ejercicio 7.1.	182
7.2 Solución al Ejercicio 7.2.	184
7.3 Solución al Ejercicio 7.3.	185
7.4 Solución al Ejercicio 7.4.	186
7.5 Solución al Ejercicio 7.5.	187
7.6 Solución al Ejercicio 7.6.	188
7.7 Solución al Ejercicio 7.7.	189
7.8 Solución al Ejercicio 7.8.	190
7.9 Solución al Ejercicio 7.9.	191
7.10 Solución al Ejercicio 7.10.	192
8.1 Solución al Ejercicio 8.1.	207
8.2 Solución al Ejercicio 8.2.	209
8.3 Solución al Ejercicio 8.3.	211
8.4 Solución al Ejercicio 8.4.	213
8.5 Solución al Ejercicio 8.5.	215
8.6 Solución al Ejercicio 8.6.	217
8.7 Solución al Ejercicio 8.7.	219
8.8 Solución al Ejercicio 8.8.	221
8.9 Solución al Ejercicio 8.9.	223
8.10 Solución al Ejercicio 8.10.	225
8.11 Solución al Ejercicio 8.11.	227
8.12 Solución al Ejercicio 8.12.	229
8.13 Solución al Ejercicio 8.13.	231
8.14 Solución al Ejercicio 8.14.	233
8.15 Solución al Ejercicio 8.15.	235
9.1 Solución al Ejercicio 9.1.	248
9.2 Solución al Ejercicio 9.2.	249
9.3 Solución al Ejercicio 9.3.	250
9.4 Solución al Ejercicio 9.4.	251
9.5 Solución al Ejercicio 9.5.	252
9.6 Solución al Ejercicio 9.6.	254
9.7 Solución al Ejercicio 9.7.	255
9.8 Solución al Ejercicio 9.8.	257
9.9 Solución al Ejercicio 9.9.	258
9.10 Solución al Ejercicio 9.10.	259
9.11 Solución al Ejercicio 9.11.	260

9.12 Solución al Ejercicio 9.12.	261
9.13 Solución al Ejercicio 9.13.	262
9.14 Solución al Ejercicio 9.14.	263
9.15 Solución al Ejercicio 9.15.	264
9.16 Solución al Ejercicio 9.16.	265
9.17 Solución al Ejercicio 9.17.	266
9.18 Solución al Ejercicio 9.18.	268
9.19 Solución al Ejercicio 9.19.	269
9.20 Solución al Ejercicio 9.20.	270
9.21 Solución al Ejercicio 9.21.	271
9.22 Solución al Ejercicio 9.22.	272
9.23 Solución al Ejercicio 9.23.	273
9.24 Solución al Ejercicio 9.24.	274
9.25 Solución al Ejercicio 9.25.	275
9.26 Solución al Ejercicio 9.26.	276
9.27 Solución al Ejercicio 9.27.	278
9.28 Solución al Ejercicio 9.28.	279
9.29 Solución al Ejercicio 9.29.	280
9.30 Solución al Ejercicio 9.30.	282
10.1 Solución al Ejercicio 10.1.	304
10.2 Solución al Ejercicio 10.2.	305
10.3 Solución al Ejercicio 10.3.	306
10.4 Solución al Ejercicio 10.4.	307
10.5 Solución al Ejercicio 10.5.	308
10.6 Solución al Ejercicio 10.6.	310
10.7 Solución al Ejercicio 10.7.	312
10.8 Solución al Ejercicio 10.8.	314
10.9 Solución al Ejercicio 10.9.	316
10.10 Solución al Ejercicio 10.10.	318
10.11 Solución al Ejercicio 10.11.	319
10.12 Solución al Ejercicio 10.12.	320
10.13 Solución al Ejercicio 10.13.	321
10.14 Solución al Ejercicio 10.14.	322
10.15 Solución al Ejercicio 10.15.	323
10.16 Solución al Ejercicio 10.16.	324
10.17 Solución al Ejercicio 10.17.	325
10.18 Solución al Ejercicio 10.18.	326
10.19 Solución al Ejercicio 10.19.	327
10.20 Solución al Ejercicio 10.20.	329
10.21 Solución al Ejercicio 10.21.	330
10.22 Solución al Ejercicio 10.22.	331
10.23 Solución al Ejercicio 10.23.	332

10.24	Solución al Ejercicio 10.24.	333
10.25	Solución al Ejercicio 10.25.	334
10.26	Solución al Ejercicio 10.26.	335
11.1	Código de ejemplo: gestión de contactos.	343
11.2	Código de ejemplo: almacenamiento y recuperación de datos en un archivo binario.	350
11.3	Solución al Ejercicio 11.1.	354
11.4	Solución al Ejercicio 11.2.	356
11.5	Solución al Ejercicio 11.3.	358
11.6	Solución al Ejercicio 11.3 – <i>continuación</i>	359
11.7	Solución al Ejercicio 11.4.	361
11.8	Solución al Ejercicio 11.5.	363
11.9	Solución al Ejercicio 11.5 – <i>continuación</i>	364
11.10	Solución al Ejercicio 11.6.	366
11.11	Solución al Ejercicio 11.6 – <i>continuación</i>	367
11.12	Solución al Ejercicio 11.7.	369
11.13	Solución al Ejercicio 11.7 – <i>continuación</i>	370
11.14	Solución al Ejercicio 11.8.	372
11.15	Solución al Ejercicio 11.8 – <i>continuación</i>	373
11.16	Solución al Ejercicio 11.9.	375
11.17	Solución al Ejercicio 11.9 – <i>continuación</i>	376
11.18	Solución al Ejercicio 11.10.	378
11.19	Solución al Ejercicio 11.10 – <i>continuación</i>	379
12.1	Solución al Ejercicio 12.1.	399
13.1	Solución al Ejercicio 13.1.	409
13.2	Solución al Ejercicio 13.2.	411
13.3	Solución al Ejercicio 13.3.	413
13.4	Solución al Ejercicio 13.4.	415
13.5	Solución al Ejercicio 13.5.	416
13.6	Solución al Ejercicio 13.6.	418
13.7	Solución al Ejercicio 13.7.	420
14.1	Solución al Ejercicio 14.1.	432
14.2	Solución al Ejercicio 14.2.	434
14.3	Solución al Ejercicio 14.3.	436
14.4	Solución al Ejercicio 14.4.	437
14.5	Solución al Ejercicio 14.5.	439
14.6	Solución al Ejercicio 14.6.	441
14.7	Solución al Ejercicio 14.7.	443
14.8	Solución al Ejercicio 14.8.	445

Prefacio

¿Te has preguntado alguna vez cuánto puede ayudarte aprender Python a programar y a analizar datos? La programación y la ciencia de datos se han convertido en competencias transversales en ciencia e ingeniería: permiten automatizar tareas, modelar problemas y extraer conocimiento a partir de los datos. Sus aplicaciones abarcan dominios muy distintos, desde el análisis experimental hasta la ciberseguridad, donde se utilizan para detectar tráfico de red anómalo, clasificar programas maliciosos o analizar registros de actividad. En todos esos contextos Python es una de las herramientas de referencia, por su sintaxis legible, su amplio ecosistema de bibliotecas y su comunidad activa.

Sin embargo, antes de poder analizar un conjunto de datos o entrenar un modelo, es necesario dominar los fundamentos del lenguaje: cómo se estructura un programa, qué tipos de datos existen, cómo se controla el flujo de ejecución, cómo se modulariza el código y cómo se manipulan ficheros y datos estructurados. Este libro está pensado precisamente para ese objetivo: proporcionar una base sólida de programación en Python y de tratamiento de datos, sobre la que después se pueden abordar tareas de análisis de datos o de aprendizaje automático en cualquier dominio de aplicación.

Objetivo del libro

Este libro es una introducción autocontenida a la programación en Python y a los fundamentos de la ciencia de datos, dirigida a quien quiera adquirir las habilidades necesarias para resolver problemas de cálculo y de tratamiento de datos. El énfasis está puesto en los principios, técnicas y herramientas que permiten convertirse en un programador competente, capaz de comprender y resolver problemas de procesamiento de información mediante la escritura de programas en Python.

El libro recopila una serie de capítulos que cubren desde los fundamentos del lenguaje (tipos de datos, operadores, estructuras de control, funciones, ficheros) hasta las bibliotecas más relevantes para el análisis y manipulación de datos (**numpy** y **pandas**), pasando por el desarrollo modular y el diseño descendente. A través de numerosos ejemplos, ejercicios resueltos y código ejecutable, el lector se familiariza de forma progresiva con las herramientas que más adelante podrá aplicar en proyectos de análisis de datos y de aprendizaje automático.

Este libro también ha sido creado con la intención de servir de apoyo a profesionales o docentes interesados en impartir contenidos de programación y de introducción a la ciencia de datos. Los ejercicios al final de cada capítulo pueden utilizarse para crear material de laboratorio que permita al estudiante asimilar los conceptos a través del enfoque «aprender haciendo» (*learning by doing*).

Espero que el libro te resulte útil. Si encuentras alguna errata o algo que se pueda mejorar, te ruego que me lo comuniques para revisarlo. Eres libre de utilizar este libro como prefieras, recordando siempre citar la fuente.

Cómo leer el libro

El libro se organiza en catorce capítulos que siguen una progresión incremental, agrupados conceptualmente en cinco bloques.

El primer bloque, formado por los Capítulos 1–3, establece los cimientos: la instalación del entorno de trabajo basado en Docker, Jupyter Notebook y Visual Studio Code (Capítulo 1), una introducción a la programación y a la resolución automática de problemas con Python (Capítulo 2), y una primera presentación de los tipos de datos y operadores del lenguaje (Capítulo 3).

El segundo bloque, formado por los Capítulos 4–6, aborda las estructuras de control y el desarrollo modular: instrucciones simples y estructuradas (Capítulo 4), diseño y uso de funciones (Capítulo 5) y desarrollo modular y diseño descendente como metodología de resolución de problemas (Capítulo 6).

El tercer bloque, formado por los Capítulos 7–10, profundiza en los tipos de datos simples y estructurados: enteros (Capítulo 7), flotantes (Capítulo 8), booleanos y cadenas (Capítulo 9) y, finalmente, las estructuras de datos más usadas en Python (listas, diccionarios, tuplas y conjuntos) junto con las funciones de primer orden (Capítulo 10).

El cuarto bloque, formado por los Capítulos 11–12, se centra en la manipulación de datos a mayor escala: la entrada/salida y el manejo de ficheros (Capítulo 11), y una introducción a las bibliotecas `numpy` y `pandas`, fundamentales para el procesamiento numérico y el análisis de datos tabulares (Capítulo 12).

El quinto y último bloque, formado por los Capítulos 13–14, da los primeros pasos en ciencia de datos: la visualización de datos con `matplotlib` y `seaborn` (Capítulo 13) y una introducción al aprendizaje automático con `scikit-learn` (Capítulo 14).

Cada capítulo termina con una sección de ejercicios cuyo objetivo es transformar los conceptos teóricos en práctica concreta. Trabajar los ejercicios es la única forma de que las técnicas se conviertan en *músculo*: leer sin practicar produce una vaga familiaridad que no sobrevive al primer problema real. Las soluciones se ofrecen al final de cada ejercicio, deliberadamente impresas con una rotación de 180° para invitar al lector a intentar resolverlo antes de mirarlo.

Un lector con experiencia previa en programación en cualquier otro lenguaje puede saltarse el Capítulo 2 en una primera lectura y volver a él como referencia. Un lector cuyo interés principal sea el procesamiento numérico y el análisis de datos puede leer los Capítulos 1, 3 y 4 para asentar el vocabulario mínimo y luego saltar directamente a los Capítulos 12 a 14.

Prerrequisitos

El libro asume un conocimiento básico que cualquier estudiante o profesional con formación universitaria suele tener:

- Manejo de un sistema operativo a nivel de usuario: navegación de ficheros, instalación de aplicaciones, uso elemental del terminal.
- Nociones elementales de matemáticas: aritmética, álgebra básica, conceptos de función e iteración.
- Disposición a experimentar con el código. El aprendizaje de la programación requiere ejecutar, equivocarse y corregir; el lector debe estar dispuesto a teclear los ejemplos, modificarlos y observar el comportamiento del programa.

El libro *no* asume experiencia previa con Python ni con ningún otro lenguaje de programación; todo se construye desde cero, partiendo del concepto de algoritmo y avanzando capítulo a capítulo.

Entorno de trabajo

Para reproducir los ejemplos del libro y resolver los ejercicios propuestos, se ha optado por un entorno basado en *Docker* y *Jupyter Notebook*. El Capítulo 1 detalla los pasos necesarios para configurarlo, evitando así los problemas habituales de compatibilidad entre versiones de Python o conflictos entre bibliotecas en el sistema anfitrión. Como editor de código complementario se recomienda Visual Studio Code, también descrito en ese mismo capítulo.

El uso de un contenedor Docker garantiza que todos los lectores trabajen sobre un entorno reproducible e idéntico, independientemente del sistema operativo del que dispongan (Windows, macOS o Linux), con todas las bibliotecas necesarias preinstaladas. El fichero «*Dockerfile*» y el listado de dependencias («*requirements.txt*») se distribuyen junto con el material del libro.

Declaración de uso responsable de inteligencia artificial generativa

Resulta pertinente, en un libro sobre programación y ciencia de datos, ser transparente sobre el uso que se ha hecho de herramientas de inteligencia artificial generativa durante su elaboración. Durante la preparación de este libro, el autor ha utilizado herramientas de inteligencia artificial generativa (en particular, modelos de lenguaje de gran tamaño) como apoyo para tareas concretas como la armonización del texto, la generación de versiones alternativas de explicaciones, la revisión de erratas o la sugerencia de ejemplos adicionales. En todos los casos, el contenido producido por estas herramientas

ha sido cuidadosamente revisado, validado y editado por el autor, quien **asume la plena responsabilidad del contenido publicado**, incluyendo cualquier error, omisión o imprecisión que pudiera persistir.

Esta forma de trabajar es coherente con el espíritu pedagógico del propio libro: la inteligencia artificial es una herramienta poderosa, pero no sustituye al criterio humano. Es responsabilidad del lector (y, en su caso, del docente que se apoye en este libro) mantener una actitud crítica frente a cualquier salida producida por una IA generativa, contrastar la información, comprender los conceptos subyacentes y no delegar en la herramienta el aprendizaje. Programar bien y analizar datos con rigor exige entender lo que se hace, no únicamente aceptar lo que un modelo sugiere.

Sin más, te invito a abrir el primer capítulo, instalar el entorno y empezar a escribir tu primer programa. *¡Buen viaje!*

1

Instalación del entorno de trabajo: Jupyter Notebook (en Docker) y Visual Studio Code

En este capítulo se explica cómo instalar y configurar el entorno de desarrollo que usaremos a lo largo del libro. La estrategia consiste en **desplegar Python y Jupyter Notebook dentro de un contenedor Docker**, de modo que el lector trabaje sobre un entorno reproducible, idéntico al del autor e independiente del sistema operativo de su máquina (Windows, macOS o Linux). Adicionalmente, se describe cómo emplear Visual Studio Code como editor frente al servidor Jupyter en ejecución dentro del contenedor, lo que aporta una experiencia de desarrollo más rica que la del navegador para proyectos con varios ficheros.

1.1. ¿Por qué un entorno aislado en Docker?

Aprender a programar implica equivocarse: instalar bibliotecas que rompen otras versiones, mezclar entornos, terminar con una instalación de Python desordenada en el sistema. Para evitarlo, este libro adopta un enfoque de **entorno reproducible**: empaquetar Python, Jupyter y todas las dependencias dentro de un contenedor Docker. Las ventajas son tres:

- *Aislamiento*: lo que se instale dentro del contenedor no afecta al resto del sistema. Si algo falla, basta con borrar el contenedor y crearlo de nuevo.
- *Reproducibilidad*: el mismo «**Dockerfile**» produce el mismo entorno en cualquier máquina, hoy y dentro de tres años. Los ejemplos del libro funcionan exactamente

1. Instalación del entorno de trabajo: Jupyter Notebook (en Docker) y Visual Studio Code

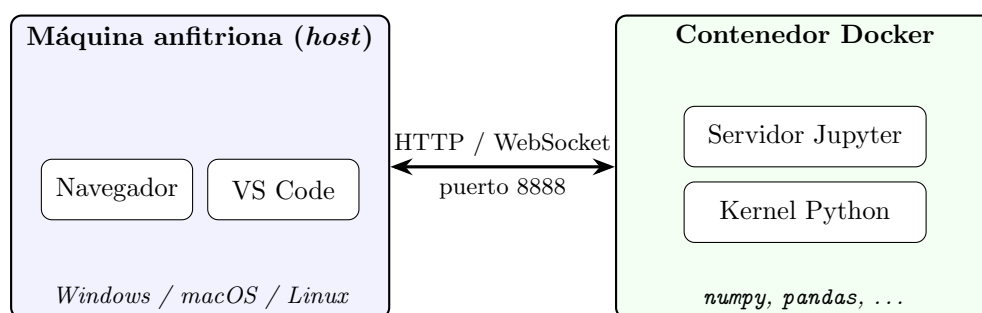


Figura 1: Arquitectura del entorno de trabajo. El navegador o Visual Studio Code, instalados en la máquina anfitriona, se comunican con el servidor Jupyter que corre dentro del contenedor Docker mediante el puerto 8888.

igual en tu equipo que en el del autor.

- *Portabilidad:* el contenedor se ejecuta de la misma forma en Windows, macOS y Linux.

La Figura 1 resume la arquitectura del entorno: la máquina anfitriona (*host*) ejecuta Docker, dentro del cual corre el servidor Jupyter; el lector accede al servidor desde un navegador o desde Visual Studio Code mediante el puerto 8888.

1.2. Instalación de Docker

Antes de comenzar, asegúrate de tener Docker instalado en tu máquina. Si aún no lo tienes, descárgalo desde el sitio oficial: <https://www.docker.com/get-started/>. En este libro recomendamos instalar Docker Desktop (<https://docs.docker.com/desktop/>), ya que ofrece una interfaz sencilla para gestionar contenedores y simplifica la configuración del entorno tanto en Windows como en macOS o Linux.

Tras instalarlo, ejecuta la aplicación. Aparecerá una ventana similar a la Figura 2. No es necesario crear ninguna cuenta: puedes saltar este paso (opción `Skip` en la esquina superior derecha).

1.3. Configuración del entorno: Dockerfile y dependencias

La forma más cómoda de definir un entorno Docker es escribir un fichero llamado «Dockerfile» que describe, paso a paso, cómo construirlo: imagen base, dependencias a instalar, puerto a exponer y comando que se ejecuta al arrancar. A continuación se muestra el «Dockerfile» que usaremos a lo largo del libro (Código 1.1). Crea este fichero, con ese nombre exacto, en el directorio que vayas a usar como espacio de trabajo.

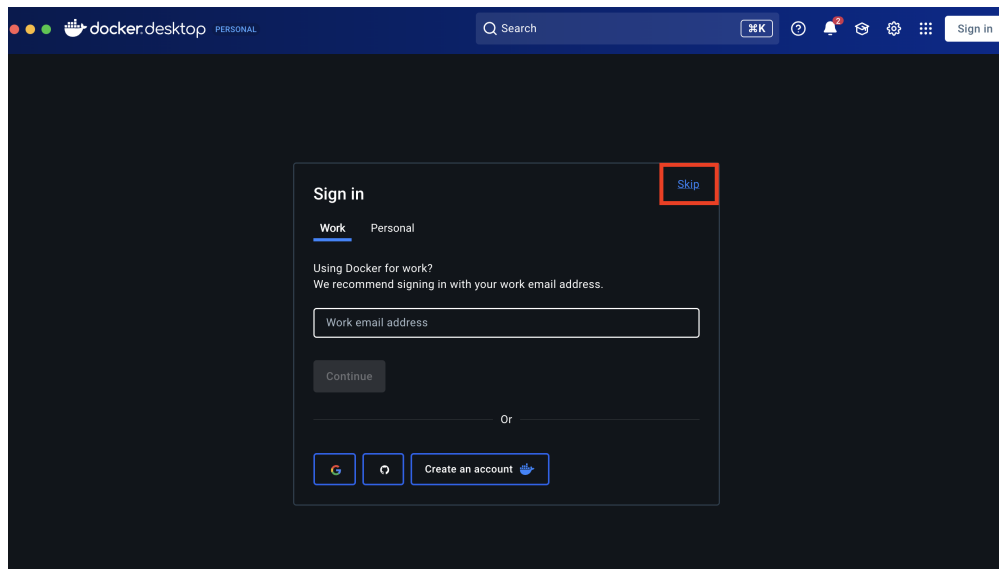


Figura 2: Ventana de inicio de Docker Desktop.

Código 1.1: Fichero «Dockerfile» para crear el entorno de trabajo del libro.

```
1  # Imagen oficial de Python como base
2  FROM python:3.12
3
4  # Variables de entorno recomendadas para Python en contenedor
5  ENV PYTHONDONTWRITEBYTECODE=1
6  ENV PYTHONUNBUFFERED=1
7
8  # Directorio de trabajo dentro del contenedor
9  WORKDIR /workspace
10
11 # Copia el listado de dependencias al contenedor
12 COPY requirements.txt .
13
14 # Instala JupyterLab y las bibliotecas del libro
15 RUN pip install --no-cache-dir jupyterlab && \
16     pip install --no-cache-dir -r requirements.txt
17
18 # Expone el puerto 8888 para Jupyter
19 EXPOSE 8888
20
```

```
21 # Arranca Jupyter al iniciar el contenedor
22 ENTRYPOINT ["jupyter", "lab", "--ip=0.0.0.0", "--port=8888",
↪ \
23     "--no-browser", "--allow-root"]
```

Cada bloque cumple una función concreta:

- `FROM python:3.12` parte de una imagen oficial de Python en su versión 3.12, que ya incluye el intérprete y `pip` preinstalados.
- Las dos variables `PYTHONDONTWRITEBYTECODE` y `PYTHONUNBUFFERED` son recomendaciones habituales para Python dentro de un contenedor: evitan generar ficheros `.pyc` y aseguran que las salidas (`print`, errores) aparezcan al instante en el log.
- `WORKDIR /workspace` fija `«/workspace»` como directorio de trabajo dentro del contenedor.
- `COPY requirements.txt .` introduce dentro del contenedor el listado de dependencias (que veremos enseguida).
- `RUN pip install ...` instala JupyterLab (que incluye Jupyter Notebook clásico) y las bibliotecas adicionales declaradas en `«requirements.txt»`.
- `EXPOSE 8888` declara que el contenedor publicará el servicio en el puerto 8888 (el puerto por defecto de Jupyter).
- `ENTRYPOINT [...]` indica el comando que se ejecuta al iniciar el contenedor: arrancar JupyterLab escuchando en todas las interfaces, sin abrir un navegador interno y permitiendo el acceso como `root` dentro del contenedor.

1.3.1. Dependencias adicionales

Para los ejemplos y ejercicios del libro será necesario instalar algunas bibliotecas adicionales en el entorno de trabajo. Se trata de bibliotecas para el manejo eficiente de datos numéricos y estructurados, así como para su visualización:

- **Manipulación de datos:**
 - `numpy`: cálculo numérico vectorizado y manipulación de vectores y matrices multidimensionales.
 - `pandas`: análisis y manipulación de datos tabulares (lectura de CSV, agrupaciones, filtrados, etc.).
- **Visualización de datos:**
 - `matplotlib`: representación gráfica de datos (líneas, barras, dispersión, etc.).
 - `seaborn`: gráficos estadísticos de alto nivel contruidos sobre `matplotlib`.

Si más adelante quieres trabajar con técnicas concretas de aprendizaje automático o de análisis de datos, podrás añadir bibliotecas adicionales: `scipy` (operaciones matemáticas avanzadas), `statsmodels` (modelos estadísticos), `plotly` (visualización interactiva), `requests` (acceso a servicios web) o `Pillow` (procesamiento de imágenes), entre muchas otras. Sin embargo, este libro sólo necesita las cinco listadas anteriormente.

Crea, en el mismo directorio donde guardaste el «`Dockerfile`», un fichero llamado «`requirements.txt`» con el contenido del Código 1.2. Las versiones se fijan a un mínimo razonable para garantizar la reproducibilidad de los ejemplos del libro a lo largo del tiempo.

Código 1.2: Fichero «`requirements.txt`» con las dependencias usadas en el libro.

```
1 numpy>=2.0,<3
2 pandas>=2.2,<3
3 matplotlib>=3.9,<4
4 seaborn>=0.13,<1
5 scikit-learn>=1.4,<2
```

1.4. Construcción de la imagen

A partir del «`Dockerfile`» y del «`requirements.txt`» es necesario construir la imagen de Docker, es decir, materializar la receta en una plantilla ejecutable. Para ello, abre una terminal en el directorio donde se encuentran ambos ficheros y ejecuta:

```
docker build -t fundamentos-python .
```

Este comando crea una imagen llamada `fundamentos-python` con todas las dependencias del libro. El proceso puede tardar unos minutos la primera vez, ya que descargará la imagen base de Python 3.12, JupyterLab y las bibliotecas del «`requirements.txt`». Una vez finalizado, la imagen quedará lista para ejecutar el contenedor.

1.5. Ejecución del contenedor

Una vez construida la imagen, hay dos formas de levantar el contenedor: a través de la interfaz gráfica de Docker Desktop o desde la línea de comandos.

1.5.1. Desde Docker Desktop

Abre Docker Desktop y dirígete a la sección `Images`. Como se muestra en la Figura 3, la imagen recién creada aparece en la lista con el nombre `fundamentos-python` y la etiqueta `latest`.

Para iniciar el contenedor desde la propia aplicación, pulsa el botón ► (parte derecha de la Figura 3). Se abrirá una ventana de configuración opcional (Figura 4) donde conviene rellenar:

1. Instalación del entorno de trabajo: Jupyter Notebook (en Docker) y Visual Studio Code

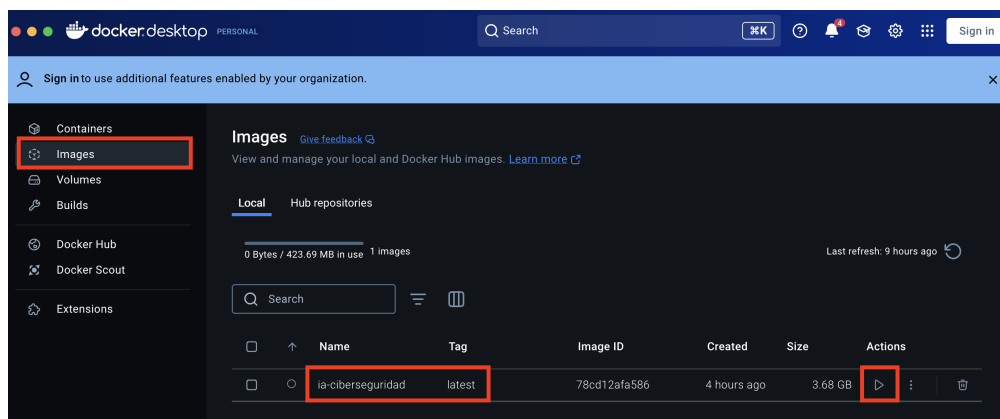


Figura 3: Docker Desktop mostrando la imagen `fundamentos-python` creada anteriormente.

- *Nombre del contenedor*: puedes dejar `fundamentos-python` o elegir otro nombre.
- *Puertos*: asegúrate de que el campo *Host port* sea 8888. Si ese puerto está ocupado por otro servicio en tu máquina, puedes usar otro distinto. Esta configuración mapea el puerto interno del contenedor (donde escucha Jupyter) al puerto correspondiente de la máquina *host*, permitiendo el acceso desde el navegador o desde VS Code.

Al pulsar `Run` se inicia el contenedor y aparece la pantalla de log (Figura 5). Anota la URL que aparece en el log: la necesitarás para conectarte al servidor Jupyter desde el navegador.

1.5.2. Desde la terminal

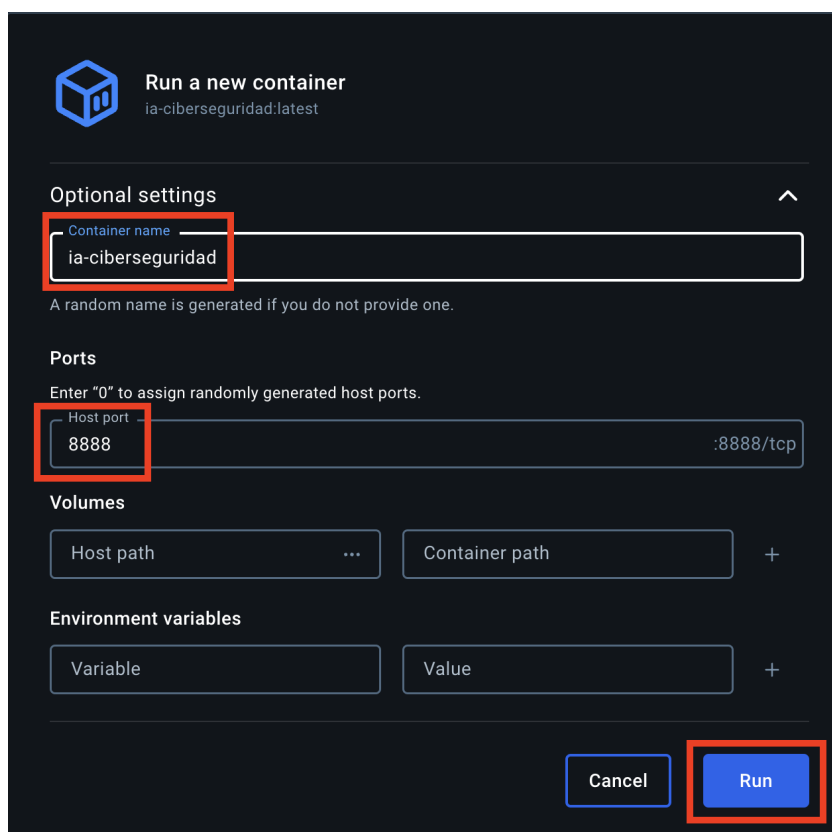
Como alternativa, en macOS o Linux abre una terminal y ejecuta:

```
docker run --rm -p 8888:8888 --name fundamentos-python \
-v "$(pwd)":/workspace fundamentos-python
```

En Windows, si usas *PowerShell*, sustituye `$(pwd)` por `${PWD}`; si usas la consola `cmd`, por `%cd%`.

El comando `docker run` usa varios parámetros:

- `--rm`: borra el contenedor automáticamente al pararse, evitando dejar contenedores huérfanos.
- `-p 8888:8888`: mapea el puerto 8888 de la máquina anfitriona al puerto 8888 del contenedor (formato `HOST:CONTENEDOR`).
- `--name fundamentos-python`: asigna un nombre identificativo al contenedor.



The screenshot shows the 'Run a new container' window in Docker Desktop. At the top, it says 'Run a new container' and 'ia-ciberseguridad:latest'. Below this is an 'Optional settings' section. The 'Container name' field is highlighted with a red box and contains the text 'ia-ciberseguridad'. Below this, it says 'A random name is generated if you do not provide one.' The 'Ports' section is next, with the instruction 'Enter "0" to assign randomly generated host ports.' The 'Host port' field is highlighted with a red box and contains the text '8888'. To the right of this field, it says ':8888/tcp'. Below the ports section is the 'Volumes' section, which has two input fields: 'Host path' and 'Container path', with a plus sign to the right. The 'Environment variables' section is at the bottom, with two input fields: 'Variable' and 'Value', with a plus sign to the right. At the bottom right of the window, there are two buttons: 'Cancel' and 'Run'. The 'Run' button is highlighted with a red box.

Figura 4: Configuración del contenedor de la imagen `fundamentos-python` en Docker Desktop.

- `-v "$(pwd)":/workspace`: monta el directorio de trabajo actual de la máquina anfitriona dentro de `/workspace` en el contenedor, de modo que los notebooks que crees se guarden en tu disco y no se pierdan al borrar el contenedor.
- El último `fundamentos-python` es el nombre de la imagen a ejecutar.

Tras el arranque, la terminal muestra el log del servidor Jupyter, incluida la URL completa con el *token* de autenticación.

1.6. Acceso a Jupyter desde el navegador

Jupyter es una herramienta interactiva que permite ejecutar código en celdas, ver resultados de forma inmediata y combinar texto, gráficos y código en un mismo documento. Su unidad básica de trabajo es el *notebook*: un fichero con extensión `.ipynb` que contiene una secuencia de celdas en las que se puede escribir y ejecutar código Python, añadir explicaciones en texto enriquecido (formato Markdown) o insertar imágenes, tablas y gráficos.

1. Instalación del entorno de trabajo: Jupyter Notebook (en Docker) y Visual Studio Code

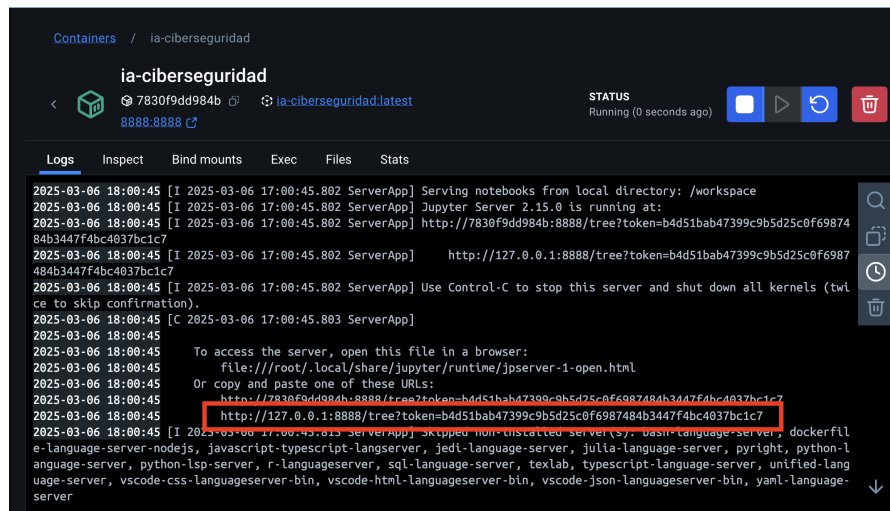


Figura 5: Pantalla de *log* en Docker Desktop tras arrancar el contenedor: muestra la URL con el *token* para acceder a Jupyter.

1.6.1. Conectar al servidor

Con el contenedor en ejecución, abre un navegador web y accede a la URL que aparece en el log. Tendrá la forma:

`http://127.0.0.1:8888/lab?token=...`

Si cambiaste el puerto del *host* en el paso anterior, actualízalo en la URL. Es imprescindible incluir el `?token=...` que ofrece el log: es el mecanismo de autenticación de Jupyter.

Al acceder, se abre la interfaz principal de JupyterLab, donde se ven los ficheros del directorio de trabajo (entre ellos, el `requirements.txt` si montaste el directorio con `-v`); véase la Figura 6.

1.6.2. Crear un notebook y elegir kernel

Para crear un *notebook*, dirígete a `File` `>` `New` `>` `Notebook` (Figura 7). Aparecerá una ventana de selección de *kernel* (Figura 8).

El *kernel* es el motor que ejecuta el código del *notebook*. Jupyter soporta múltiples lenguajes, aunque el más habitual es Python con el *kernel* `ipykernel`. En la ventana de selección, elige `Python 3 (ipykernel)` y pulsa `Select`. Si en algún momento el *kernel* deja de responder, puedes reiniciarlo desde `Kernel` `>` `Restart`.

Un *notebook* se compone de celdas de varios tipos:

- *Celdas de código*: contienen código Python que se ejecuta al pulsar `⇧` + `⇩` o el botón `▶`; el resultado aparece debajo de la celda.

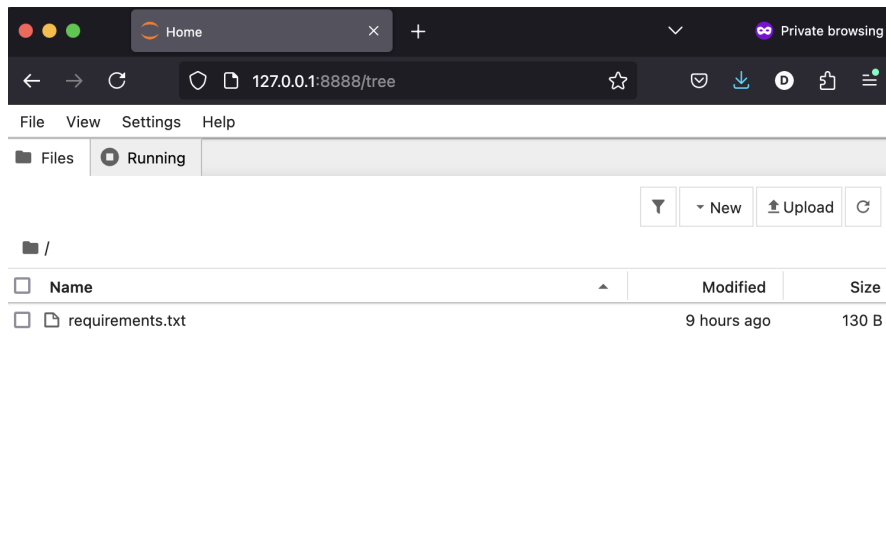


Figura 6: Interfaz principal de Jupyter al acceder desde la máquina anfitriona.

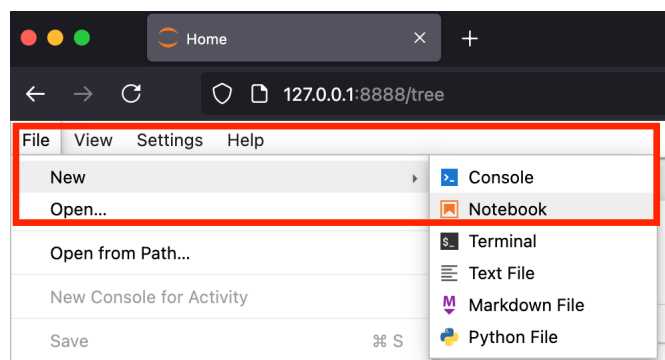


Figura 7: Creación de un nuevo *notebook* en Jupyter.

- *Celdas Markdown*: permiten escribir texto formateado con Markdown, incluyendo títulos, listas, imágenes y ecuaciones matemáticas en $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$.
- *Celdas de salida*: se generan automáticamente bajo cada celda de código y muestran su salida (texto, gráficos, datos tabulares).

1.6.3. Probar el entorno

En la primera celda del *notebook* recién creado, escribe el siguiente código de prueba:

```
1 | print("Hola, mundo desde Python")
```

Al ejecutar la celda con `⇧ + ↵`, el mensaje `Hola, mundo desde Python` aparece justo debajo (Figura 9), lo que **confirma que el entorno está funcionando correctamente**. A partir de aquí, ya puedes empezar a programar.

1. Instalación del entorno de trabajo: Jupyter Notebook (en Docker) y Visual Studio Code

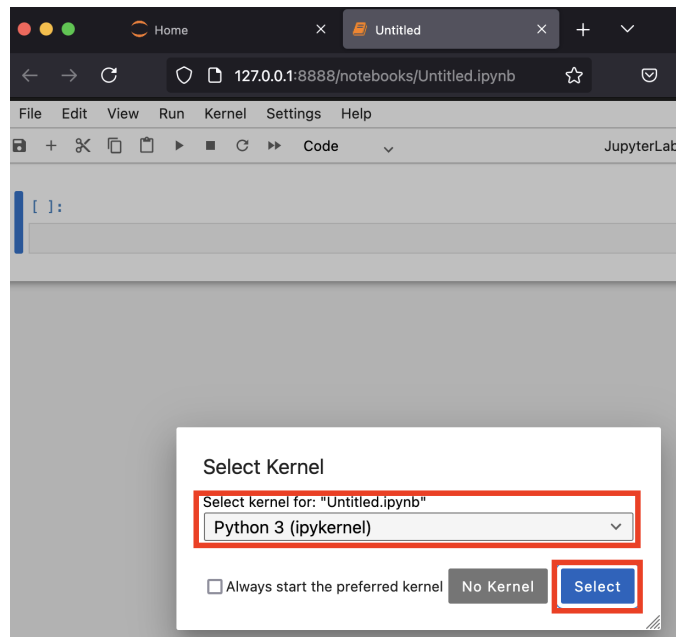


Figura 8: Selección de *kernel* en Jupyter.

1.7. Visual Studio Code como entorno alternativo

Visual Studio Code (VS Code) es un *entorno de desarrollo integrado* (en inglés, *Integrated Development Environment*, o IDE) ligero, gratuito y altamente configurable, que permite programar en múltiples lenguajes (incluido Python) y que puede usarse para todo el contenido de este libro. A diferencia de Jupyter, que se centra en la ejecución interactiva por celdas y el análisis exploratorio, VS Code ofrece una experiencia de desarrollo más estructurada con depurador, control de versiones integrado con Git, resaltado de sintaxis, autocompletado inteligente y soporte para entornos virtuales.

Mientras Jupyter resulta ideal para pruebas rápidas y experimentación con datos, VS Code permite organizar proyectos con varios ficheros y trabajar con ellos de manera eficiente. Además, gracias a la extensión de Jupyter, es posible combinar lo mejor de ambos mundos: ejecutar *notebooks* dentro de VS Code, sin necesidad de abrir el navegador.

1.7.1. Instalación

Descarga VS Code desde su página oficial e instálalo: <https://code.visualstudio.com/download>.

Una vez instalado, abre la aplicación. Para poder ejecutar *notebooks* dentro de VS Code es necesario instalar la extensión de Jupyter. En la barra lateral izquierda, haz clic en el icono de **Extensions** (Figura 10), busca **Jupyter** e instala la extensión oficial de Microsoft (Figura 11).

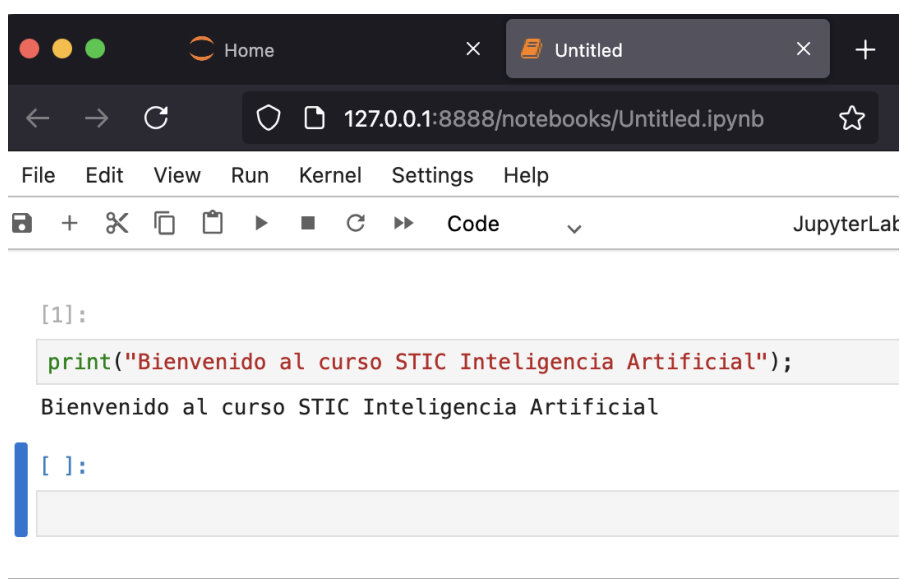


Figura 9: Ejecución del primer programa Python en Jupyter desde el navegador.

1.7.2. Conectar VS Code al servidor Jupyter del contenedor

Una vez instalada la extensión, crea un nuevo fichero llamado «prueba.ipynb» (la extensión `.ipynb` corresponde a los *notebooks* de Jupyter). Para ejecutarlo, antes hay que seleccionar un *kernel*: la opción se encuentra en la parte derecha de la barra superior del fichero recién creado (Figura 12).

Si VS Code detecta que no tienes la extensión de Python instalada (Figura 13), instálala: la necesita para gestionar los *kernels*.

Como Jupyter está corriendo dentro del contenedor Docker, en lugar de usar un *kernel* local nos conectamos al servidor remoto. Selecciona la opción `Existing Jupyter Server` (Figura 14) e introduce la URL del servidor que se está ejecutando en el contenedor (por ejemplo, `http://127.0.0.1:8888/lab?token=tu_token`). Pulsa `↵` para confirmar.

Tras conectar (VS Code preguntará por un nombre identificativo del servidor; puedes poner el que prefieras), selecciona el *kernel* Python 3 (`ipykernel`) que se ejecutará dentro del contenedor (Figura 15).

A partir de aquí ya puedes programar y ejecutar código Python desde VS Code conectado al contenedor. En la primera celda escribe el código de prueba:

```
1 | print("Hola, mundo desde Python")
```

Pulsa `⇧+↵` (ejecuta y crea una nueva celda debajo) o el botón de ejecutar celda (sólo ejecuta la actual). El otro botón de la barra superior del fichero ejecuta **todas** las celdas del *notebook* de una vez.

Si todo está correctamente configurado, el mensaje `Hola, mundo desde Python` se imprime justo debajo de la celda (Figura 16). Llegado este punto, dispones de VS Code

1. Instalación del entorno de trabajo: Jupyter Notebook (en Docker) y Visual Studio Code

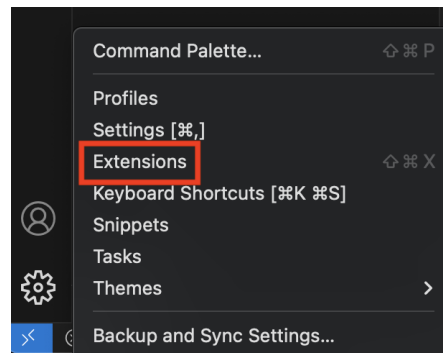


Figura 10: Pestaña `Extensions` de VS Code en la barra lateral izquierda.

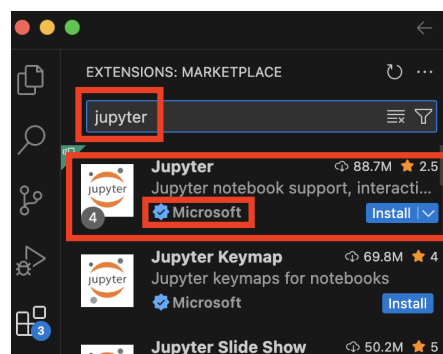


Figura 11: Instalación de la extensión `Jupyter` en VS Code.

conectado a un servidor Jupyter en Docker: puedes aprovechar el editor y el depurador de VS Code mientras trabajas con *notebooks* dentro de un contenedor aislado.

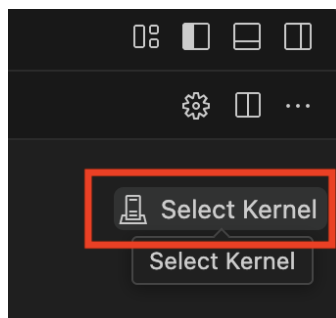


Figura 12: Selector de *kernel* en la parte derecha de la barra superior del fichero «.ipynb» en VS Code.

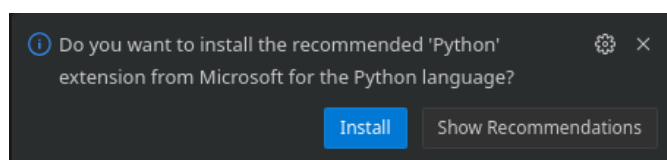


Figura 13: Sugerencia de VS Code para instalar la extensión de Python.

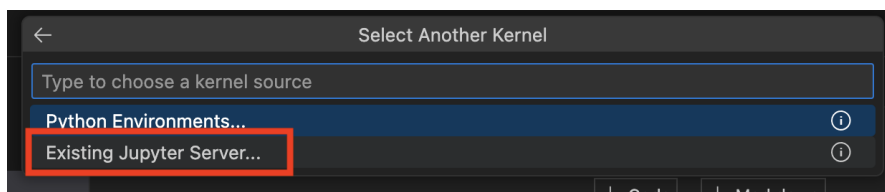


Figura 14: Conexión a un servidor Jupyter existente desde VS Code.

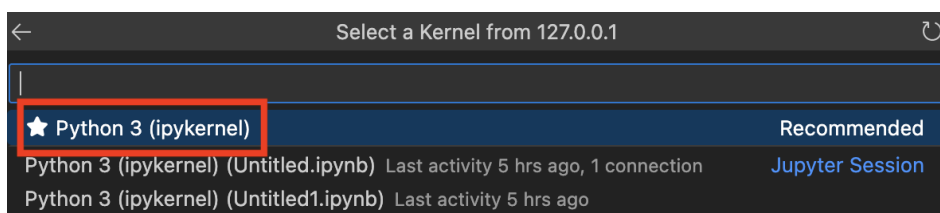


Figura 15: Selección del *kernel* en el servidor Jupyter remoto.

1. Instalación del entorno de trabajo: Jupyter Notebook (en Docker) y Visual Studio Code

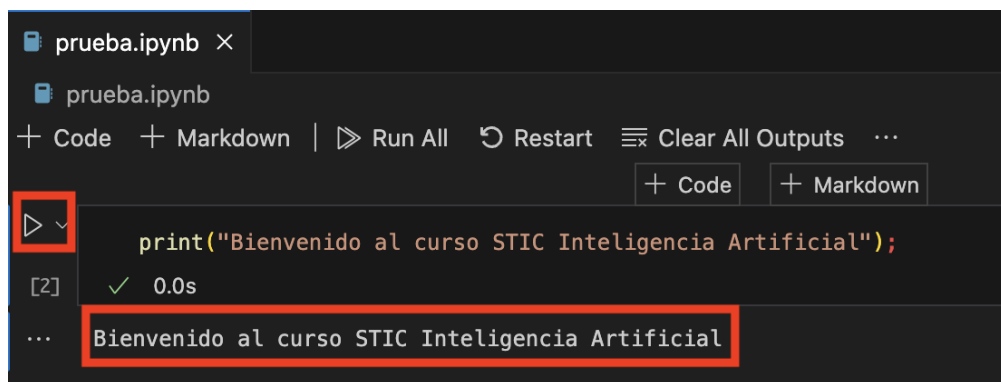


Figura 16: Ejecución del primer programa Python desde un *notebook* en VS Code conectado al contenedor.

2

Introducción a la programación y resolución automática de problemas con Python

En este capítulo exploraremos *cómo los programas pueden utilizarse para resolver problemas relacionados con el procesamiento de información cuando son ejecutados en un computador*. Con el entorno de trabajo ya configurado en el Capítulo 1, el siguiente paso es entender qué es un algoritmo, qué es un programa, y cómo se relacionan ambos conceptos. Veremos también los primeros elementos de la sintaxis de Python para que el lector pueda escribir y ejecutar sus primeros programas.

2.1. Programación y resolución automática de problemas

La programación es una disciplina que permite automatizar la resolución de problemas en ámbitos muy diversos: cálculo numérico, gestión de datos, procesamiento de imágenes, reconocimiento y traducción de lenguaje natural, control de sistemas industriales, robótica, conducción autónoma, análisis de señales biomédicas, ciberseguridad, entre muchos otros.

Los computadores (término que se usará en este libro para referirse a los equipos electrónicos programables, sean ordenadores personales, servidores o sistemas embebidos) están diseñados para procesar información de forma automática y eficiente. Su funcionamiento depende de programas que, al ejecutarse, deben proporcionar soluciones a los problemas planteados.

La pregunta central que aborda este libro es: *¿cómo escribir un programa para resolver un problema de procesamiento de información?*

Para resolver un problema, el primer paso es **identificar uno o varios métodos que puedan utilizarse para encontrar una solución**. Luego, es necesario **seleccionar el más adecuado** considerando aspectos como su eficiencia y su simplicidad. Posteriormente, **el método elegido debe formalizarse por escrito**, describiendo la información relevante y la forma en que debe ser procesada para obtener un resultado. Esa formalización se conoce como *algoritmo*.

Algoritmo: secuencia ordenada y finita de pasos, definida sin ambigüedades, que permite encontrar la solución a un problema de procesamiento de información.

Los algoritmos pueden expresarse en distintos lenguajes: lenguajes de programación como Python, C, C++, Java, JavaScript, Rust o Go, o lenguas naturales como español, inglés o francés. Los lenguajes de programación tienen la ventaja de ser comprensibles directamente por los computadores, algo que aún no es posible con los lenguajes naturales sin una capa de interpretación adicional.

Cuando un algoritmo se implementa en un lenguaje de programación, se convierte en un *programa*. Existen cientos de lenguajes de programación, surgen nuevos constantemente, y otros caen en desuso con el tiempo.

En este libro utilizaremos Python como lenguaje de referencia para expresar algoritmos. Sin embargo, el enfoque principal no será el dominio exclusivo de su sintaxis, sino la comprensión de los principios y técnicas de programación, y la aplicación de buenas prácticas de diseño de software. Al finalizar este libro, el lector dispondrá de los conocimientos y habilidades necesarios para analizar y resolver problemas de procesamiento de información, y será capaz de elegir el lenguaje más adecuado a cada contexto.

Un mini-ejemplo: del problema al algoritmo. Supongamos el siguiente problema: dada una lista de notas de un examen, calcular la nota media. Sin haber escrito todavía una sola línea de Python, podemos plantear el algoritmo en lenguaje natural:

1. Empezar con un acumulador con valor cero y un contador con valor cero.
2. Para cada nota de la lista:
 - a) sumar la nota al acumulador;
 - b) incrementar el contador en uno.
3. Si el contador es cero, no hay notas y la media no está definida.
4. En caso contrario, la media es el acumulador dividido entre el contador.

Esta secuencia ordenada y sin ambigüedades es un algoritmo. La traducción a Python (es decir, su conversión en programa) la veremos a lo largo de los próximos capítulos.

2.2. Propiedades de un algoritmo

Un algoritmo debe cumplir ciertas propiedades para garantizar su correcto funcionamiento y facilitar su comprensión y reutilización. Estas propiedades pueden agruparse en *imprescindibles* y *deseables*.

Las propiedades *imprescindibles* son:

- **Corrección:** un algoritmo debe proporcionar una solución correcta al problema planteado. Sus resultados deben ajustarse a la especificación dada y resolver el problema de manera adecuada.
- **Legibilidad:** la descripción de un algoritmo debe ser clara y fácil de entender, no solo para quien lo diseñó, sino también para cualquier persona con conocimientos de programación. Esto facilita su revisión, mantenimiento y posible modificación.

Las propiedades *deseables*, aunque no estrictamente necesarias, mejoran la calidad y utilidad de un algoritmo:

- **Generalidad:** un algoritmo no debe restringirse a resolver un problema demasiado específico si, con un esfuerzo adicional asumible, puede aplicarse a una gama más amplia de problemas.
- **Reusabilidad:** un buen algoritmo debe poder reutilizarse en otros programas sin modificaciones o con ajustes mínimos. Esto mejora la eficiencia en el desarrollo y evita la redundancia de código.
- **Eficiencia:** un buen algoritmo debe utilizar los recursos de manera óptima, considerando tanto el tiempo de ejecución como el consumo de memoria. La eficiencia es crucial para resolver problemas de manera rápida y sin sobrecargar el sistema.
- **Independencia del lenguaje y del sistema:** un algoritmo debe ser lo suficientemente abstracto como para implementarse en distintos lenguajes de programación y ejecutarse en distintos entornos sin dificultades.

2.3. Estructura de un programa en Python

Python es un lenguaje de programación ampliamente utilizado por su sencillez y legibilidad. A continuación se presenta un programa muy simple que imprime un mensaje en pantalla:

```
1 | # Pre: ---
2 | # Post: Escribe en pantalla el mensaje "Hola, mundo desde
↪ | Python"
3 |
4 | print("Hola, mundo desde Python")
```

2. Introducción a la programación y resolución automática de problemas con Python

A pesar de su brevedad, este programa ya muestra los elementos básicos de un programa Python:

- **Comentarios:** las dos primeras líneas comienzan por el símbolo `#`, lo que las marca como comentarios. Python ignora todo lo que aparece tras un `#` hasta el final de la línea. En este caso, los comentarios documentan la *precondición* y la *postcondición* del programa: qué se asume al arrancar (**Pre:** `---`, es decir, ninguna), y qué garantiza al terminar (**Post:** `...`). Documentar pre y postcondiciones es una práctica recomendada al diseñar algoritmos.
- **Instrucción:** la última línea contiene una llamada a la función `print`, que muestra el mensaje recibido como argumento entre paréntesis.

En Jupyter Notebook, al ejecutar la celda, la salida aparece justo debajo de la celda de código, sin necesidad de invocaciones adicionales.

2.4. Desarrollo y ejecución de un programa en Python

El desarrollo de un programa en Python consta de tres pasos principales: (1) escribir el código en un archivo o celda; (2) ejecutarlo; e (3) interpretar los resultados.

La gestión de estos pasos depende del entorno en el que se trabaje. En un entorno interactivo como Jupyter Notebook, la ejecución se realiza directamente desde una celda de código. En un editor de texto o en un IDE, el flujo habitual es escribir el código en un archivo con extensión `.py` y ejecutarlo desde una terminal con:

```
python nombre_del_archivo.py
```

En un entorno interactivo, basta con ejecutar la celda donde se encuentra el código. Por ejemplo, ejecutar la celda:

```
1 | print("Hola, mundo desde Python")
```

mostrará en pantalla el mensaje `Hola, mundo desde Python`.

El desarrollo y la ejecución también pueden gestionarse desde un IDE. Un IDE es un software que facilita la edición de código, la ejecución y, cuando aparecen errores, su depuración. Existen numerosos IDEs para Python; algunos populares son PyCharm, Visual Studio Code, Spyder y Thonny. En este libro recomendamos VS Code (descrito en el Capítulo 1) por su ligereza y por su buena integración con Jupyter.

Es importante que cualquier código esté debidamente documentado mediante comentarios. Los comentarios ayudan a explicar la funcionalidad del programa, el propósito de las variables y las acciones realizadas. En Python, los comentarios se introducen con el símbolo `#` y duran hasta el final de la línea. Una buena documentación facilita el mantenimiento del código y permite que otros programadores lo entiendan y modifiquen en el futuro. Como regla general, los comentarios deben ser claros y concisos, evitando explicaciones innecesarias o redundantes.

2.5. Guía de estilo en la programación

Al escribir algoritmos y programas conviene seguir una serie de reglas de estilo cuyo propósito es mejorar la calidad del código y su legibilidad. Mantener un estilo coherente facilita el trabajo del propio programador y la colaboración con otros desarrolladores.

En Python, la guía de estilo de referencia es **PEP 8**¹, escrita por Guido van Rossum (creador del lenguaje) y mantenida por la comunidad. Algunas reglas básicas de PEP 8 que conviene conocer desde el principio son:

- **Indentación:** usar 4 espacios por nivel de indentación. No mezclar tabuladores y espacios.
- **Longitud de línea:** limitar las líneas a 79 caracteres como máximo (en docstrings y comentarios, a 72).
- **Nombres de variables y funciones:** en `snake_case` (todo en minúsculas con guiones bajos para separar palabras), por ejemplo, `edad_alumno` o `calcular_media`.
- **Nombres de constantes:** en mayúsculas con guiones bajos, por ejemplo, `PI = 3.141592`.
- **Espacios alrededor de operadores:** `a = b + c` y no `a=b+c`.
- **Líneas en blanco:** separar funciones y clases con dos líneas en blanco; bloques lógicos dentro de funciones, con una línea en blanco.

Los ejemplos de este libro siguen estas reglas, y se anima al lector a adoptarlas desde el primer programa. Existen herramientas que comprueban automáticamente el cumplimiento de PEP 8 (por ejemplo, `flake8` o `pycodestyle`); otras como `black` reformatean el código directamente.

2.6. Sintaxis básica de Python: primeros pasos

La sintaxis de Python es sencilla y fácil de aprender, lo que lo hace un lenguaje excelente para principiantes. Vamos a comenzar con los conceptos básicos. Todos los ejemplos pueden probarse desde una celda de Jupyter Notebook o desde una terminal de Python.

En Python, las *sentencias* son instrucciones que indican al computador qué hacer. Cada línea de código es, en general, una sentencia. Python utiliza la *indentación* para estructurar los bloques de código y organizar el flujo del programa. A diferencia de otros lenguajes (C, C++, Java, JavaScript), Python *no* usa llaves `{ }` para delimitar bloques: la indentación es semánticamente significativa.

¹PEP, del inglés *Python Enhancement Proposal*, número 8: <https://peps.python.org/pep-0008/>.

```
1 | print("Hola, mundo desde Python")
```

El ejemplo anterior es una sentencia que imprime el mensaje "Hola, mundo desde Python" en la consola. Veamos cómo se usa la indentación para definir un bloque:

```
1 | if 5 > 3:
2 |     print("5 es mayor que 3")
```

En este caso, la sentencia `print` está dentro del bloque del `if` porque está indentada con 4 espacios. Sin la indentación, Python no sabría que la llamada forma parte del cuerpo de la condición. Las sentencias condicionales se estudian en detalle en capítulos posteriores.

2.6.1. Comentarios y docstrings

Los comentarios sirven para añadir notas explicativas dentro del código. Python los ignora durante la ejecución. En Python, los comentarios se introducen con el símbolo `#` y se extienden hasta el final de la línea:

```
1 | # Esta línea es un comentario
2 | print("Hola, mundo") # Otro comentario al final de una línea
```

Para comentar varias líneas, lo idiomático en Python es prefijar cada una con `#`:

```
1 | # Este programa imprime un saludo
2 | # y termina su ejecución
3 | print("Hola, mundo")
```

A veces verás cadenas de varias líneas delimitadas por comillas triples (`""" ... """` o `''' ... '''`) usadas como si fueran comentarios:

```
1 | """
2 | Esto NO es un comentario en sentido estricto:
3 | es una cadena literal que el intérprete descarta.
4 | """
5 | print("Hola, mundo")
```

En realidad, lo que aparece entre comillas triples es un *literal de cadena de varias líneas*: si no se asigna a ninguna variable ni se imprime, simplemente se evalúa y se descarta. Su uso idiomático no es como comentario, sino como **docstring** (cadena de documentación) en la primera línea de un módulo, función o clase, donde sí tiene un significado especial: las herramientas de Python (como `help()`) la leen para describir el objeto. Los *docstrings* se estudiarán con mayor detalle al introducir funciones y módulos.

2.6.2. Variables

En Python, una *variable* es un nombre que apunta a un valor almacenado en memoria. Ese valor puede cambiar a lo largo de la ejecución, según el programa lo manipule. Las variables son, por tanto, contenedores que almacenan datos que pueden ser utilizados y modificados durante la ejecución.

A diferencia de lenguajes como C o Java, en Python no es necesario declarar el tipo de una variable: el tipo lo determina el valor que se le asigna. Python es un *lenguaje dinámicamente tipado*, lo que significa que el tipo de una variable puede cambiar a lo largo de la ejecución.

Para usar una variable basta con asignarle un valor mediante el operador `=`. El nombre de la variable, llamado *identificador*, debe cumplir algunas reglas:

- Debe comenzar por una letra o un guion bajo (`_`).
- Puede continuar con letras, dígitos o guiones bajos.
- Es sensible a mayúsculas y minúsculas: `edad` y `Edad` son variables distintas.
- No puede coincidir con una *palabra reservada* del lenguaje (por ejemplo, `if`, `for`, `class`, `return`).

Veamos un ejemplo. Tras la sentencia:

```
1 | edad = 25
```

Python infiere automáticamente, a partir del valor asignado, que la variable `edad` contiene un entero. Si más adelante se asigna a la misma variable un valor de otro tipo:

```
1 | edad = "Veinticinco"
```

la variable pasa a contener una cadena de texto. Esta flexibilidad facilita el desarrollo rápido pero también **puede ser fuente de errores si no se tiene cuidado con los tipos de los datos que se manipulan**.

2.6.3. Tipos de datos básicos (introducción)

Los tipos de datos más básicos en Python son los enteros, los flotantes, las cadenas y los booleanos. Estos tipos son esenciales para realizar cálculos y manipular información. A continuación se presentan brevemente, junto con un ejemplo de literal y de asignación. Un *literal* es la representación directa de un valor en el código fuente, sin tener que calcularlo ni invocar funciones.

Enteros (`int`)

Los enteros son números sin decimales. Pueden ser positivos o negativos y, en Python, no tienen un tamaño fijo: pueden crecer hasta el límite que permita la memoria disponible.

```
1 | numero_entero = 42
```

En el ejemplo, se define una variable `numero_entero` y se le asigna el literal entero 42.

Flotantes (float)

Los flotantes son números con parte decimal. Python utiliza una precisión doble de punto flotante para representarlos, lo que le permite manejar una amplia gama de valores reales.

```
1 | numero_flotante = 3.141592
```

Cadenas (str)

Las cadenas son secuencias de caracteres, es decir, textos. Se pueden delimitar con comillas simples (' ') o con comillas dobles (" "), pero no se pueden mezclar (las comillas que abren la cadena deben coincidir con las que la cierran). Ambas formas son equivalentes en Python.

```
1 | mensaje = "Hola, ¿cómo estás?"
```

Booleanos (bool)

Los valores booleanos sólo pueden tomar dos valores: **True** (verdadero) o **False** (falso). Son muy útiles en el control de flujo del programa, especialmente en condiciones y bucles, que se estudiarán en capítulos posteriores.

```
1 | es_mayor = True
```

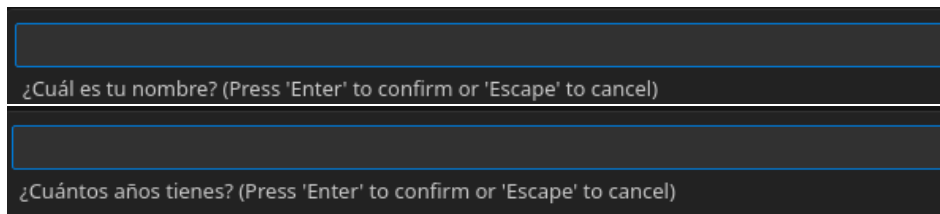
2.6.4. Entrada y salida de datos

Una de las primeras necesidades de un programa es interactuar con el usuario. En Python, la función `input()` se usa para leer información de la consola. Esta función **siempre devuelve una cadena de texto**, así que si lo que se necesita es un número, hay que realizar una *conversión explícita* usando funciones como `int()` (para enteros) o `float()` (para reales).

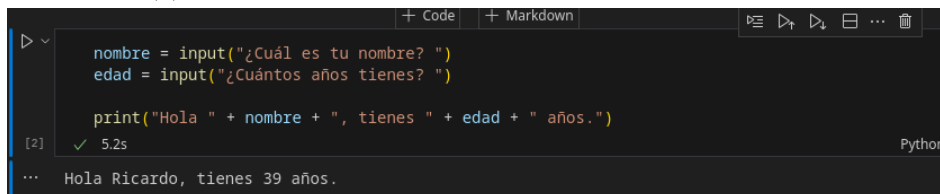
A continuación se muestra un ejemplo de lectura de datos del usuario y muestra de un saludo:

```
1 | nombre = input("¿Cuál es tu nombre? ")
2 | edad = input("¿Cuántos años tienes? ")
3 |
4 | print("Hola " + nombre + ", tienes " + edad + " años.")
```

La función `print()` muestra información al usuario. Conviene destacar que en el ejemplo la variable `edad` **no** es un entero, sino una cadena, ya que los datos leídos con `input()` siempre son cadenas. Si se necesitara como entero, habría que convertirlo explícitamente con `edad = int(edad)`. La diferencia entre la *conversión implícita* (la que Python hace automáticamente, como al sumar un `int` con un `float`) y la *conversión explícita* (la que el programador realiza llamando a `int()`, `float()`, etc.) se estudiará en detalle en el siguiente capítulo. Para concatenar cadenas y valores de variables se puede



(a) Ventanas emergentes para petición de nombre y edad



(b) Ejecución de la celda tras la introducción de datos

Figura 1: Interacción con el usuario desde Jupyter Notebook en VS Code.

usar el operador `+`, o emplear *f-strings* (cadenas formateadas), que se verán en detalle también en el siguiente capítulo.

Si se trabaja con VS Code y Jupyter Notebook, al ejecutar una celda con interacción, los datos se solicitan mediante una ventana flotante en la parte central superior de la ventana (Figura 1).

2.6.5. Operadores aritméticos

Python soporta un conjunto básico de operadores aritméticos para realizar operaciones matemáticas simples. Estos operadores son esenciales en cualquier programa que realice cálculos. A continuación se describe cada uno con un breve ejemplo.

Suma (+)

Suma dos valores.

```
1 resultado = 5 + 3
2 print(resultado) # Imprime 8
```

Resta (-)

Resta el segundo valor del primero.

```
1 resultado = 5 - 3
2 print(resultado) # Imprime 2
```

Multiplicación (*)

Multiplica dos valores.

```
1 resultado = 5 * 3
2 print(resultado) # Imprime 15
```

División (/)

Realiza la división de dos valores. Es importante recordar que **en Python 3, la división con / (división verdadera) siempre devuelve un valor de tipo float, incluso si los dos operandos son enteros**. Esto difiere del comportamiento de otros lenguajes de programación.

```
1 resultado = 5 / 2
2 print(resultado) # Imprime 2.5; resultado es de tipo float
```

División entera (//)

Realiza la división eliminando la parte decimal del cociente. El tipo del resultado depende de los operandos: si ambos son enteros, el resultado es entero; si alguno de los operandos es flotante, el resultado es flotante.

```
1 print(5 // 2) # Imprime 2 (int)
2 print(5.0 // 2) # Imprime 2.0 (float)
```

Módulo (%)

Calcula el resto de una división. Es muy útil para verificar, por ejemplo, si un número es par o impar.

```
1 resultado = 5 % 2
2 print(resultado) # Imprime 1 (resto de la división)
```

Exponenciación (**)

Eleva un número a una potencia.

```
1 resultado = 5 ** 2
2 print(resultado) # Imprime 25
```

Estos operadores permiten escribir programas que realicen cálculos numéricos básicos. En programación más avanzada, se combinan para resolver problemas matemáticos más complejos, desde la física hasta las ciencias computacionales y la ciencia de datos.

2.7. Multiparadigma en Python

Python es un lenguaje **multiparadigma**: permite programar bajo distintos enfoques. En estos primeros capítulos trabajaremos bajo el **paradigma imperativo**, en el que se define una secuencia de instrucciones que el intérprete ejecuta de arriba a abajo, modificando el estado de las variables.

A medida que avance el aprendizaje, se introducirán otros paradigmas. Python soporta también la **programación orientada a objetos**, que organiza el código en clases y objetos, favoreciendo la modularidad y la reutilización; y la **programación funcional**, que trata las funciones como valores de primera clase y permite aplicar técnicas como las funciones de orden superior y construcciones como `map`, `filter` o `reduce`. Posteriormente, se trabajará también bajo los enfoques de **programación estructurada** y **programación modular**, que se explican en capítulos posteriores.

2.8. Ejercicios

EJERCICIO 2.1

Escribe un programa en Python que solicite al usuario su nombre y su ciudad de origen, y luego lo salude de manera personalizada mencionando ambos datos.

Ejemplo de ejecución esperada:

```
¿Cuál es tu nombre? Alberto
¿De qué ciudad eres? Teruel
Hola, Alberto de Teruel.
¡Bienvenido a la programación con Python!
```

Modifica ahora el programa para que también pregunte por el país de origen y lo incluya en el mensaje de bienvenida.

Ejemplo de ejecución esperada:

```
¿Cuál es tu nombre? Ana
¿De qué ciudad eres? Buenos Aires
¿De qué país eres? Argentina
Hola, Ana de Buenos Aires, Argentina.
¡Bienvenido a la programación con Python!
```

Solución (girar para leer)

```
1 # Pre: El usuario debe ingresar su nombre y su ciudad
2 # Post: Se muestra un mensaje personalizado de
3 #       bienvenida con la ciudad de origen.
4 # Solicita el nombre y la ciudad del usuario
5 nombre = input("¿Cuál es tu nombre? ")
6 ciudad = input("¿De qué ciudad eres? ")
7 # Solicita el país de origen
8 país = input("¿De qué país eres? ")
9
10 # Muestra el mensaje de bienvenida
11 print("Hola, " + nombre + " de " + ciudad + ", " + país
12 print("!Bienvenido a la programación con Python!")
```

Código 2.1: Solución al Ejercicio 2.1.

EJERCICIO 2.2

Escribe un programa en Python que solicite el nombre y la edad del usuario, y luego lo salude indicando cuántos años tendrá dentro de 5 años.

Ejemplo de ejecución esperada:

```
¿Cuál es tu nombre? Sofía
¿Cuántos años tienes? 30
Hola, Sofía. Dentro de 5 años tendrás 35 años.
```

Modifica ahora el programa para que pregunte cuántos años en el futuro quiere calcular el usuario y muestre el resultado basado en su elección.

Ejemplo de ejecución esperada:

```
¿Cuál es tu nombre? Marcos
¿Cuántos años tienes? 25
¿Dentro de cuántos años quieres saber tu edad? 10
Hola, Marcos. Dentro de 10 años tendrás 35 años.
```

Solución (girar para leer)

```

1 # Pre: El usuario debe ingresar su nombre y edad.
2 # Post: Se muestra un mensaje con su edad dentro de 5 a
3     ños.
4 # Solicita el nombre y la edad del usuario
5 nombre = input("¿Cuál es tu nombre? ")
6 edad = int(input("¿Cuántos años tienes? "))
7
8 # Solicita los años que el usuario quiere calcular en
9     el futuro
10 años_futuro = int(input("¿Dentro de cuántos años
11     quieres saber tu edad? "))
12 # Calcula la edad en el futuro especificado
13 edad_futura = edad + años_futuro
14 # Muestra el mensaje con la edad futura
15 print("Hola, " + nombre + " . Dentro de " + str(
16     edad_futura) + " años tendrás " + str(edad_futura)
17     + " años.")

```

Código 2.2: Solución al Ejercicio 2.2.

EJERCICIO 2.3

Escribe un programa en Python que solicite al usuario una temperatura en grados Celsius y la convierta a grados Fahrenheit, usando la fórmula:

$$F = \left(C \cdot \frac{9}{5} \right) + 32$$

Muestra el resultado en pantalla.

Ejemplo de ejecución esperada:

```
Introduce la temperatura en grados Celsius: 25
25 grados Celsius son 77.0 grados Fahrenheit.
```

Modifica ahora el programa para que también convierta la temperatura a Kelvin utilizando la fórmula:

$$K = C + 273.15$$

Ejemplo de ejecución esperada:

```
Introduce la temperatura en grados Celsius: 30
30 grados Celsius son 86.0 grados Fahrenheit y 303.15 grados Kelvin.
```

Solución (girar para leer)

```

1 # Pre: El usuario debe ingresar una temperatura en
  grados Celsius.
2 # Post: Se muestra la conversión a grados Fahrenheit.
3
4 # Solicita la temperatura en Celsius
5 Celsius = float(input("Introduce la temperatura en
  grados Celsius: "))
6
7 # Convierte a Fahrenheit
8 Fahrenheit = (Celsius * 9/5) + 32
9 # Convierte a Kelvin
10 Kelvin = Celsius + 273.15
11
12 # Muestra los resultados en ambas unidades
13 print(f"{int(Celsius)} grados Celsius son {Fahrenheit}
  grados Fahrenheit y {Kelvin} grados Kelvin.")

```

Código 2.3: Solución al Ejercicio 2.3.

EJERCICIO 2.4

Escribe un programa en Python que solicite al usuario la base y la altura de un rectángulo y calcule su área usando la fórmula:

$$A = b \cdot h$$

donde b es la base y h es la altura. Muestra el resultado en pantalla.

Ejemplo de ejecución esperada:

```
Introduce la base del rectángulo: 10
Introduce la altura del rectángulo: 5
El área del rectángulo es 50.
```

Modifica ahora el programa para que también calcule el perímetro del rectángulo usando la fórmula:

$$P = 2 \cdot (b + h)$$

Ejemplo de ejecución esperada:

```
Introduce la base del rectángulo: 8
Introduce la altura del rectángulo: 4
El área del rectángulo es 32 y el perímetro es 24.
```


Solución (girar para leer)

```

1 # Pre: El usuario debe ingresar la base y la altura de
  ↩ un rectángulo.
2 # Post: Se muestra el área calculada.
3
4 # Solicita la base y la altura del rectángulo
5 base = float(input("Introduce la base del rectángulo: "))
  ↩
6 altura = float(input("Introduce la altura del rectá
  ↩ ngulo: "))
7
8 # Calcula el área
9 area = base * altura
10
11 # Calcula el perímetro
12 perimetro = 2 * (base + altura)
13
14 # Muestra el resultado con el perímetro incluido
15 print(f"El área del rectángulo es {int(area)} y el perí
  ↩ metro es {int(perimetro)}.")

```

Código 2.4: Solución al Ejercicio 2.4.

EJERCICIO 2.5

Escribe un programa en Python que calcule el área de un círculo. El programa debe solicitar al usuario el radio del círculo e imprimir el resultado del área utilizando la fórmula:

$$A = \pi \cdot r^2$$

donde A es el área y r es el radio del círculo. Usa el valor de π como 3.141592.

Ejemplo de ejecución esperada:

```
Introduce el radio del círculo: 5
El área del círculo es 78.5398
```

Modifica ahora el programa para que también calcule el perímetro del círculo utilizando la fórmula:

$$P = 2 \cdot \pi \cdot r$$

Ejemplo de ejecución esperada:

```
Introduce el radio del círculo: 5
El área del círculo es 78.5398 y el perímetro es 31.41592
```

Solución (girar para leer)

```

1 # Pre: El usuario debe ingresar el radio del círculo.
2 # Post: Se muestra el área del círculo calculada con la
3     fórmula  $A = \pi r^2$ .
4 # Solicita el radio del círculo
5 radio = float(input("Introduce el radio del círculo: "))
6
7 # Define el valor de pi
8 pi = 3.141592
9
10 # Calcula el área del círculo
11 area = pi * radio ** 2
12
13 # Calcula el perímetro del círculo
14 perimetro = 2 * pi * radio
15
16 # Muestra el resultado con el área y el perímetro
17 print("El área del círculo es " + str(area) + " y el
    perímetro es " + str(perimetro))

```

Código 2.5: Solución al Ejercicio 2.5.

3

Tipos de datos y operadores

Los programas informáticos gestionan información en forma de datos. En este capítulo aprenderemos *cómo definir y manipular datos en Python, así como los operadores y conversiones de tipos* disponibles en el lenguaje.

3.1. Tipos de datos

En Python, cada dato tiene un tipo asociado. Los tipos de datos más básicos del lenguaje se muestran en la Tabla 1.

Tipo	Representa	Ejemplo (literales del dato)
int	Números enteros	-10972, -13, 0, 5, 17, 209
float	Números reales	-1.0E34, -32.17, 0.0, 1.0E-12, 0.47
str	Cadenas de caracteres	"Hola", 'Python', "" (cadena vacía)
bool	Valores lógicos	True, False

Tabla 1: Tipos de datos más básicos en Python.

Para conocer el tipo de un dato en Python se utiliza la función `type()`:

```
1 print(type(42))      # <class 'int'>
2 print(type(3.1416))  # <class 'float'>
3 print(type("A"))     # <class 'str'>
4 print(type(True))    # <class 'bool'>
```

Además de estos tipos básicos, Python incluye un valor especial llamado `None`, que se utiliza para representar la *ausencia de valor*. Su tipo es `NoneType` (no aparece en la Tabla 1 porque no es un tipo en el sentido de los anteriores: `None` es el *único* valor que existe de `NoneType`). Es habitual usar `None` para inicializar variables que aún no tienen un valor significativo, o como valor de retorno de una función que no devuelve nada

explícitamente.

```
1 resultado = None
2 print(type(resultado)) # <class 'NoneType'>
```

Una nota sobre el tamaño en memoria. En el módulo estándar `sys` existe la función `getsizeof()`, que devuelve el número de bytes que ocupa un objeto en memoria. Conviene saber, sin embargo, que el valor obtenido no se corresponde con la representación matemática del dato: por ejemplo, en CPython, un entero pequeño como 0 ocupa 28 bytes, no 4 u 8, porque cada objeto de Python lleva consigo metadatos (tipo, contador de referencias, etc.). Aun así, es útil para hacerse una idea del coste relativo de los distintos tipos:

```
1 import sys
2
3 print(sys.getsizeof(42))      # tamaño de un entero
4 print(sys.getsizeof(3.1416)) # tamaño de un flotante
5 print(sys.getsizeof('A'))    # tamaño de una cadena de un
↪   carácter
6 print(sys.getsizeof(True))   # tamaño de un booleano
```

Python incluye muchas funcionalidades en módulos estándar para mantener el lenguaje ligero. Cuando se necesita acceder a herramientas adicionales, como en el ejemplo anterior, se usa `import` para traer módulos al espacio de nombres del programa.

3.2. Operaciones

Para trabajar con datos, Python proporciona operadores que permiten hacer cálculos y comparaciones. Existen distintos tipos de operadores, que se describen a continuación.

3.2.1. Operadores de comparación

Los operadores de comparación se usan para comparar dos valores y devuelven un resultado de tipo `bool` (`True` o `False`). En la Tabla 2 se muestran todos estos operadores, su significado, un ejemplo y el resultado de evaluar dicha operación.

Veamos un ejemplo:

```
1 a, b = 10, 20 # define a=10 y b=20 simultaneamente
2 print(a < b)  # True
3 print(a == b) # False
4 print(a >= b) # False
```

Operador	Significado	Ejemplo	Resultado
<	Menor que	3 < 5	True
<=	Menor o igual que	5 <= 5	True
>	Mayor que	10 > 3	True
>=	Mayor o igual que	4 >= 8	False
==	Igual a	10 == 10	True
!=	Distinto de	7 != 3	True

Tabla 2: Operadores de comparación en Python.

Operador	Nombre	Ejemplo	Resultado
+	Suma	3 + 4	7
-	Resta	10 - 5	5
*	Multiplicación	4 * 6	24
/	División (real)	9 / 2	4.5
//	División entera	9 // 2	4
%	Módulo (resto)	9 % 2	1
**	Exponente	2 ** 3	8

Tabla 3: Operadores aritméticos en Python.

3.2.2. Operadores aritméticos

Los operadores aritméticos permiten realizar operaciones matemáticas básicas. En la Tabla 3 se muestran los operadores aritméticos disponibles, con un ejemplo y el resultado.

Un ejemplo de uso:

```

1 a, b = 27, 5
2 print(a / b)    # 5.4 (división real)
3 print(a // b)   # 5 (división entera)
4 print(a % b)    # 2 (resto de la división)
5 print(3 ** 2)   # 9 (3 elevado a 2)

```

3.2.3. Operadores lógicos

Los operadores lógicos trabajan con valores booleanos para construir expresiones lógicas. En la Tabla 4 se muestran estos operadores, su significado, un ejemplo y el resultado.

El operador **and** devuelve **True** si *ambos* operandos son verdaderos, y **False** en cualquier otro caso. En cambio, **or** devuelve **True** si *al menos uno* de los operandos es verdadero, y **False** solo cuando ambos son falsos. Las tablas de verdad de **and** y **or** son:

3. Tipos de datos y operadores

Operador	Significado	Ejemplo	Resultado
not	Negación ($\neg$)	not True	False
and	Conjunción ($\wedge$)	True and False	False
or	Disyunción ($\vee$)	True or False	True

Tabla 4: Operadores lógicos en Python.

<i>a</i>	<i>b</i>	<i>a and b</i>	<i>a</i>	<i>b</i>	<i>a or b</i>
True	True	True	True	True	True
True	False	False	True	False	True
False	True	False	False	True	True
False	False	False	False	False	False

Un ejemplo de uso:

```
1 a, b = True, False
2 print(not a)      # False
3 print(a and b)    # False
4 print(a or b)     # True
```

3.2.4. Operadores de asignación

Los operadores de asignación permiten modificar el valor de una variable. La Tabla 5 muestra los más habituales.

Operador	Ejemplo	Equivalente a
=	x = 5	Asigna 5 a x
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 2	x = x / 2

Tabla 5: Operadores de asignación en Python.

Un ejemplo de uso:

```
1 x = 10
2 x += 5
3 print(x)    # 15
```


3.3. Prioridad de operadores

Cuando una expresión contiene varios operadores, es importante saber en qué orden se evalúan. Python sigue un conjunto de reglas de *prioridad* (también llamada *precedencia*) que determinan qué operación se ejecuta primero.

A continuación se describe el orden de evaluación, de mayor a menor prioridad:

- (1) Paréntesis (): tienen la mayor prioridad. Cualquier operación dentro de paréntesis se evalúa primero, sin importar qué otros operadores estén presentes. Permiten cambiar el orden de evaluación cuando se desee:

```
1 resultado = (3 + 5) * 2
2 # Primero se evalúa (3 + 5), y luego el resultado se
↪   multiplica por 2.
3 print(resultado) # 16
```

- (2) Exponente **: se evalúa antes que multiplicaciones, divisiones, sumas o restas:

```
1 resultado = 2 ** 3 + 4
2 # Primero se evalúa 2 ** 3 (que es 8), y luego se le suma
↪   4.
3 print(resultado) # 12
```

- (3) Multiplicación *, división /, división entera //, módulo %: tienen la misma prioridad y se evalúan de izquierda a derecha:

```
1 resultado = 4 * 3 // 2
2 # Primero se evalúa 4 * 3 (que es 12), y luego 12 // 2 (
↪   que es 6).
3 print(resultado) # 6
```

- (4) Suma + y resta -: prioridad menor que las multiplicativas; se evalúan de izquierda a derecha:

```
1 resultado = 5 + 2 * 3
2 # Primero se evalúa 2 * 3 (que es 6), y luego 5 + 6.
3 print(resultado) # 11
```

- (5) Operadores de comparación (<, <=, >, >=, ==, !=): se evalúan después de las operaciones aritméticas y devuelven un valor booleano:

```
1 resultado = 5 + 2 * 3 < 20
2 # Primero se evalúan las operaciones aritmeticas
3 # y al final la comparación.
4 print(resultado) # True
```

3. Tipos de datos y operadores

- (6) Negación lógica **not**: tiene mayor prioridad que **and** y **or**, pero menor que las operaciones aritméticas y de comparación.
- (7) Conjunción lógica **and**: prioridad menor que **not**.
- (8) Disyunción lógica **or**: la menor prioridad de los operadores lógicos:

```
1 resultado = 5 + 2 * 3 < 20 and 3 > 2
2 # Primero las operaciones aritmeticas, luego las
  ↪ comparaciones,
3 # y finalmente la conjuncion lógica.
4 print(resultado) # True
```

Cuando varios operadores tienen la misma prioridad, Python evalúa la expresión de izquierda a derecha:

```
1 resultado = 5 + 3 - 2
2 # Primero se evalua 5 + 3, y luego se resta 2.
3 print(resultado) # 6
```

Si en algún momento tienes dudas sobre el orden de evaluación, lo más sensato es usar paréntesis explícitamente para garantizar el orden deseado:

```
1 resultado = (5 + 3) * 2
2 # Aquí, primero se evalua 5 + 3, y luego se multiplica por 2.
3 print(resultado) # 16
```

La Tabla 6 resume las reglas anteriores.

Prioridad	Operador	Descripción
1	()	Paréntesis (más alta prioridad)
2	**	Exponente
3	*, /, //, %	Multiplicación, división, división entera, módulo
4	+, -	Suma y resta
5	<, <=, >, >=, ==, !=	Operadores de comparación
6	not	Negación lógica
7	and	Conjunción lógica
8	or	Disyunción lógica (más baja prioridad)

Tabla 6: Prioridad de los operadores básicos de Python (de mayor a menor).

3.4. Conversión de tipos

En Python los tipos de las variables son dinámicos, lo que significa que no es necesario declararlos. Sin embargo, en muchos casos hay que convertir un dato de un tipo a otro:

bien para que las operaciones tengan sentido, bien para garantizar que los valores se ajusten a un formato concreto. Existen dos tipos de conversión: *implícita* y *explícita*.

3.4.1. Conversión implícita

Python realiza automáticamente algunas conversiones de tipo cuando los operandos involucrados son compatibles, especialmente entre tipos numéricos. Este proceso se conoce como *conversión implícita*. Por ejemplo, al sumar un entero con un flotante, Python convierte automáticamente el entero a flotante para que la operación sea coherente:

```
1 x = 5          # entero
2 y = 2.3        # flotante
3 resultado = x + y  # Python convierte x a flotante
↪      automaticamente
4 print(resultado)   # 7.3
```

En este caso, `x` es un entero e `y` es un flotante, pero Python convierte `x` a flotante antes de realizar la suma, lo que evita errores y garantiza que el resultado sea correcto.

3.4.2. Conversión explícita

En otros casos, el programador debe indicar explícitamente cómo realizar la conversión. Esto se hace usando funciones de conversión como `int()`, `float()` o `str()`. La *conversión explícita* es necesaria cuando el tipo de origen no es compatible de forma implícita con el tipo de destino.

Por ejemplo, para convertir un flotante a entero (con truncamiento de la parte decimal):

```
1 x = 5.9
2 y = int(x)    # convierte x a entero, truncando la parte
↪      decimal
3 print(y)      # 5
```

O para convertir un entero a cadena de texto:

```
1 x = 10
2 y = str(x)    # convierte x a cadena de texto
3 print(y)      # '10'
```

3.4.3. Conversiones de tipo más comunes

Las conversiones más habituales al trabajar con datos en Python son:

- **De int a float:** con `float()`. Por ejemplo:

```
1 entero = 7
2 flotante = float(entero)
3 print(flotante)  # 7.0
```

- **De float a int:** con `int()`, que trunca (no redondea) la parte decimal:

```
1 | flotante = 3.75
2 | entero = int(flotante)
3 | print(entero)    # 3
```

- **De cadena a número:** con `int()` o `float()`, siempre que la cadena tenga el formato adecuado:

```
1 | texto_entero = "123"
2 | texto_flotante = "45.67"
3 |
4 | numero_entero = int(texto_entero)
5 | numero_flotante = float(texto_flotante)
6 |
7 | print(numero_entero)    # 123
8 | print(numero_flotante) # 45.67
```

- **De número a cadena:** con `str()`. Útil cuando se quiere mostrar un número en un mensaje o concatenarlo con otro texto:

```
1 | numero = 25
2 | cadena = str(numero)
3 | print(cadena)    # '25'
```

3.5. Cadenas formateadas (f-strings)

Concatenar cadenas con valores de variables usando el operador `+` requiere que todos los operandos sean cadenas: si se intenta concatenar una cadena con un entero, Python lanza un `TypeError`. Por ejemplo:

```
1 | edad = 25
2 | print("Tienes " + edad + " años")    # TypeError: can only
↪ concatenate str
```

Para resolver esto, hay que convertir explícitamente el entero a cadena con `str()`:

```
1 | print("Tienes " + str(edad) + " años")
```

Sin embargo, Python ofrece una forma más cómoda y legible: las cadenas formateadas o *f-strings* (de *formatted string literals*). Una f-string se identifica por la `f` delante de las comillas y permite incrustar expresiones entre llaves `{ }`:

```
1 | nombre = "Alicia"
2 | edad = 25
3 | print(f"Hola, {nombre}, tienes {edad} años")
4 | # Salida: Hola, Alicia, tienes 25 años
```

Dentro de las llaves se puede escribir cualquier expresión válida de Python, no solo nombres de variables:

```
1 print(f"En 5 años, {nombre} tendrá {edad + 5} años")
```

Las f-strings también permiten especificar el formato de presentación del valor mediante el carácter `:` seguido de una especificación. Por ejemplo, para mostrar un número con 2 decimales:

```
1 PI = 3.1415926535
2 print(f"Pi con 2 decimales es {PI:.2f}")
3 # Salida: Pi con 2 decimales es 3.14
```

A lo largo del libro usaremos f-strings de forma sistemática, ya que producen un código más claro que la concatenación con `+` y `str()`. En el Capítulo 9 se profundizará en otras opciones de formato.

3.6. Manejo de errores en la conversión

Las conversiones entre tipos pueden fallar en tiempo de ejecución, lo que generaría una *excepción* que detendría el programa si no se trata adecuadamente. Para que el programa pueda recuperarse y dar una respuesta útil al usuario en lugar de detenerse abruptamente, Python proporciona el mecanismo de *manejo de excepciones* mediante los bloques `try`, `except`, `else`, `finally` y la sentencia `raise`.

Los errores más comunes al convertir datos son:

Error de tipo (TypeError). Se produce cuando se intenta convertir un valor a un tipo de dato incompatible. Por ejemplo, convertir el objeto nulo `None` a entero:

```
1 num = int(None) # TypeError: int() argument must be a string
↪ ...
```

Error de valor (ValueError). Se produce cuando el valor a convertir no tiene un formato válido para el tipo de destino. Por ejemplo, `int("12.5")` y `float("abc")` generan ambos un `ValueError`, ya que ninguna de las dos cadenas representa un número válido para el tipo solicitado:

```
1 num = int("12.5") # ValueError: invalid literal for int()
↪ with base 10
```

3.6.1. Captura específica con `try / except`

El bloque `try` contiene la operación que puede fallar; el bloque `except` se ejecuta *solo* si dentro del `try` se ha producido el error indicado. La práctica recomendada es indicar siempre el tipo de excepción que se quiere capturar, para no enmascarar errores no esperados.

```
1 entrada = input("Introduce un número entero: ")
2
3 try:
4     numero = int(entrada)
5     print(f"El número introducido es {numero}")
6 except ValueError:
7     print(f"Error: '{entrada}' no es un número válido.")
```

Si el usuario escribe 42, la conversión funciona y se imprime `El número introducido es 42`. Si escribe hola, el bloque `except ValueError` atrapa la excepción y se imprime el mensaje de error sin que el programa se detenga.

3.6.2. Captura genérica con `except Exception as e`

A veces interesa capturar un conjunto amplio de errores. La forma recomendada es `except Exception as e`: se atrapan todas las excepciones derivadas de `Exception` y, además, se obtiene una referencia `e` al objeto de error, lo que permite mostrar información útil sobre lo ocurrido:

```
1 entrada = input("Introduce un número entero: ")
2
3 try:
4     numero = int(entrada)
5     print(f"El número introducido es {numero}")
6 except Exception as e:
7     print(f"Se ha producido un error: {e}")
```

Es preferible `except Exception` a `except:` a secas. Esta última forma captura *absolutamente todo*, incluyendo `KeyboardInterrupt` (Ctrl+C) y `SystemExit`, lo que generalmente es indeseable: el usuario no podría siquiera abortar un programa colgado.

3.6.3. Cláusula `else`: código sin error

La cláusula `else` se ejecuta *solo si* el bloque `try` termina sin excepciones. Es útil para separar el código que puede fallar (en `try`) del que solo tiene sentido cuando todo ha ido bien (en `else`):

```
1 entrada = input("Introduce un número entero: ")
2
3 try:
4     numero = int(entrada)
5 except ValueError:
6     print(f"Error: '{entrada}' no es un número válido.")
7 else:
8     print(f"La conversión fue exitosa: {numero}")
```

3.6.4. Cláusula finally: código que siempre se ejecuta

La cláusula `finally` se ejecuta *siempre*, haya o no haya habido excepción. Es ideal para liberar recursos (cerrar ficheros, conexiones de red, bloqueos) sin importar si la operación tuvo éxito:

```
1 entrada = input("Introduce un número entero: ")
2
3 try:
4     numero = int(entrada)
5     print(f"Doble: {numero * 2}")
6 except ValueError:
7     print(f"Error: '{entrada}' no es un número válido.")
8 finally:
9     print("Operación finalizada (con o sin error).")
```

3.6.5. Lanzar errores con raise

A veces el programa detecta una situación errónea y debe avisar de ella. La sentencia `raise` permite *lanzar* explícitamente una excepción del tipo deseado, opcionalmente con un mensaje:

```
1 edad = int(input("Edad: "))
2 if edad < 0:
3     raise ValueError(f"La edad no puede ser negativa: {edad}")
4 print(f"Edad válida: {edad}")
```

Esta excepción puede capturarse después con un `try/except` en cualquier punto del código que llame a esta porción de programa, lo que permite separar la *detección* del problema de su *tratamiento*.

3.7. Ejercicios

EJERCICIO 3.1

Escribe un programa en Python que calcule el resultado de la siguiente expresión:

$$\text{resultado} = (a + b) \cdot c - \frac{a}{b}$$

El programa debe solicitar al usuario los valores de a , b y c y mostrar el resultado. Controla el error de división entre cero si $b = 0$.

Ejemplo de ejecución esperada

```
Introduce el valor de a: 10
Introduce el valor de b: 5
Introduce el valor de c: 2
El resultado de la expresión es 28.0
```

Solución (girar para leer)

```
1  try:
2      a = float(input("Introduce el valor de a: "))
3      b = float(input("Introduce el valor de b: "))
4      c = float(input("Introduce el valor de c: "))
5      # Calculamos la expresión
6      resultado = (a + b) * c - (a / b)
7
8      # Mostramos el resultado
9      print("El resultado de la expresión es", resultado)
10
11
12 except ZeroDivisionError:
13     print("Error: No se puede dividir por cero.")
14 except ValueError:
15     print("Error: Por favor, introduce valores numé-
16         ricos válidos.")
```

Código 3.1: Solución al Ejercicio 3.1.

EJERCICIO 3.2

Escribe un programa en Python que convierta una cantidad en kilómetros a metros, centímetros y milímetros. El programa debe solicitar al usuario la cantidad en kilómetros y mostrar los resultados.

Ejemplo de ejecución esperada

```
Introduce la cantidad en kilómetros: 3.5
3.5 km equivale a:
3500.0 metros
350000.0 centímetros
3500000.0 milímetros
```

Solución (girar para leer)

```

1 kilómetros = float(input("Introduce la cantidad en kiló
2 metros = kilómetros * 1000
3 centímetros = kilómetros * 100000
4 milímetros = kilómetros * 1000000
5 print(kilómetros, "km equivale a:")
6 print(metros, "metros")
7 print(centímetros, "centímetros")
8 print(milímetros, "milímetros")

Código 3.2: Solución al Ejercicio 3.2.
```

3. Tipos de datos y operadores

EJERCICIO 3.3

Escribe un programa en Python que realice las siguientes operaciones entre un número entero y un número flotante:

- Suma
- Multiplicación
- División
- División entera
- Módulo

El programa debe solicitar al usuario los dos números, uno entero y otro flotante, y mostrar los resultados.

Ejemplo de ejecución esperada

```
Introduce un número entero: 7
Introduce un número flotante: 2.3
Suma: 9.3
Multiplicación: 16.1
División: 3.0434782608695654
División entera: 3.0
Módulo: 0.1
```

Solución (girar para leer)

```
1 entero = int(input("Introduce un número entero: "))
2 flotante = float(input("Introduce un número flotante: "))
3 suma = entero + flotante
4 multiplicacion = entero * flotante
5 division = entero / flotante
6 division-entera = entero // flotante
7 modulo = entero % flotante
8 print("Suma:", suma)
9 print(f"Multiplicación: {multiplicacion:.1f}")
10 print("División:", division)
11 print("División entera:", division-entera)
12 print(f"Módulo: {modulo:.1f}")

Código 3.3: Solución al Ejercicio 3.3.
```

EJERCICIO 3.4

Escribe un programa en Python que evalúe las siguientes expresiones booleanas:

- $a \wedge b$
- $a \vee c$
- $\neg b$
- $a \wedge (b \vee c)$

El programa debe solicitar al usuario los valores de las variables a , b y c (como True o False), y mostrar los resultados de las expresiones.

Ejemplo de ejecución esperada

```
Introduce True o False para a: True
Introduce True o False para b: False
Introduce True o False para c: True
a and b: False
a or c: True
not b: True
a and (b or c): True
```

Solución (girar para leer)

```
1 # Solicitar los valores booleanos
2 a = input("Introduce True o False para a: ")
3 b = input("Introduce True o False para b: ")
4 c = input("Introduce True o False para c: ")
5
6 # Convertir las entradas en valores booleanos
7 a == "True"
8 b == "True"
9 c == "True"
10
11 # Evaluar las expresiones booleanas
12 print("a and b:", a and b)
13 print("a or c:", a or c)
14 print("not b:", not b)
15 print("a and (b or c):", a and (b or c))
```

Código 3.4: Solución al Ejercicio 3.4.

3. Tipos de datos y operadores

EJERCICIO 3.5

Escribe un programa en Python que utilice los operadores de asignación `+=`, `-=`, `*=`, `/=`, `**=` para modificar el valor de una variable. El programa debe solicitar al usuario un número inicial y luego aplicar los operadores mencionados, mostrando el valor de la variable después de cada operación.

Ejemplo de ejecución esperada

```
Introduce un número inicial: 10
Después de += 5: 15
Después de -= 3: 12
Después de *= 2: 24
Después de /= 4: 6.0
Después de **= 2: 36.0
```

Solución (gíralo para leer)

```
1 x = int(input("Introduce un número inicial: "))
2 x += 5
3 print("Después de += 5:", x)
4 x -= 3
5 print("Después de -= 3:", x)
6 x *= 2
7 print("Después de *= 2:", x)
8 x /= 4
9 print("Después de /= 4:", x)
10 x **= 2
11 print("Después de **= 2:", x)
```

Código 3.5: Solución al Ejercicio 3.5.

EJERCICIO 3.6

Escribe un programa en Python que calcule la suma de tres números. El programa debe solicitar al usuario los tres números y luego mostrar el resultado de la suma. Incorpora un control de errores ante entradas incorrectas por parte del usuario.

Ejemplo de ejecución esperada

```
Introduce el primer número: 3
Introduce el segundo número: 5
Introduce el tercer número: 7
La suma de los tres números es 15
```

Solución (girar para leer)

```

1  try:
2      num1 = int(input("Introduce el primer número: "))
3      num2 = int(input("Introduce el segundo número: "))
4      num3 = int(input("Introduce el tercer número: "))
5      suma = num1 + num2 + num3
6      print("La suma de los tres números es", suma)
7
8
9  except ValueError:
10     print("Error: Por favor, ingresa valores numéricos válidos.")

```

Código 3.6: Solución al Ejercicio 3.6.

3. Tipos de datos y operadores

EJERCICIO 3.7

Escribe un programa en Python que reciba tres números y determine cuál es el mayor de los tres. El programa debe solicitar al usuario los tres números y mostrar el número mayor. Incorpora un control de errores ante entradas incorrectas por parte del usuario.

Ejemplo de ejecución esperada

```
Introduce el primer número: 8
Introduce el segundo número: 12
Introduce el tercer número: 5
El número mayor es 12
```

Solución (girar para leer)

```
1 try:
2     num1 = int(input("Introduce el primer número: "))
3     num2 = int(input("Introduce el segundo número: "))
4     num3 = int(input("Introduce el tercer número: "))
5     # calculamos el número mayor
6     mayor = max(num1, num2, num3)
7     print("El número mayor es", mayor)
8 except ValueError:
9     print("Error: Por favor, ingresa valores numéricos válidos.")
```

Código 3.7: Solución al Ejercicio 3.7.

EJERCICIO 3.8

Escribe un programa en Python que cuente el número de caracteres en una cadena introducida por el usuario. El programa debe solicitar al usuario una cadena de texto y luego mostrar el número de caracteres que contiene. Incorpora un control de errores ante entradas incorrectas por parte del usuario.

Ejemplo de ejecución esperada

Introduce una cadena de texto: Hola Mundo
El número de caracteres es 10

Solución (girar para leer)

```

1  try:
2      cadena = input("Introduce una cadena de texto: ")
3
4      # Contamos el número de caracteres
5      longitud = len(cadena)
6      print("El número de caracteres es", longitud)
7
8      except ValueError:
9          print("Error: Por favor, ingresa una cadena de
          texto válida.")

```

Código 3.8: Solución al Ejercicio 3.8.

3. Tipos de datos y operadores

EJERCICIO 3.9

Escribe un programa en Python que convierta una cantidad de segundos a horas, minutos y segundos. El programa debe solicitar al usuario el número de segundos e imprimir el resultado de la conversión en el formato:

horas : minutos : segundos

Para realizar la conversión, utiliza las siguientes fórmulas:

- Las horas son el cociente de la división entera de los segundos entre 3600 (número de segundos en una hora).
- Los minutos son el cociente de la división entera del resto anterior entre 60 (número de segundos en un minuto).
- Los segundos resultantes son el resto de esa última división.

Ejemplo de ejecución esperada

Introduce la cantidad de segundos: 3665

La conversión es 1:1:5

Solución (girar para leer)

```
1 segundos = int(input("Introduce la cantidad de segundos\n"))  
2  
3 # Cálculo de horas, minutos y segundos  
4 horas = segundos // 3600  
5 minutos = (segundos % 3600) // 60  
6 segundos = segundos % 3600  
7  
8 # Mostrar el resultado  
9 print("La conversión es", horas, ":", minutos, ":", segundos)  
10
```

Código 3.9: Solución al Ejercicio 3.9.

EJERCICIO 3.10

Escribe un programa en Python que calcule la distancia entre dos puntos en el plano cartesiano, dadas sus coordenadas (x_1, y_1) y (x_2, y_2) . El programa debe solicitar al usuario las coordenadas de ambos puntos y luego calcular la distancia entre ellos utilizando la fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

donde d es la distancia entre los puntos.

Ejemplo de ejecución esperada

Introduce las coordenadas del primer punto (x1, y1): 1 2
 Introduce las coordenadas del segundo punto (x2, y2): 4 6
 La distancia entre los dos puntos es 5.0

Solución (girar para leer)

```

1 # Solicitar las coordenadas de los dos puntos
2 x1, y1 = map(float, input("Introduce las coordenadas
3 del primer punto (x1, y1): ").split())
4 x2, y2 = map(float, input("Introduce las coordenadas
5 del segundo punto (x2, y2): ").split())
6 # split() se usa para separar las coordenadas, permite
7 dividir la cadena en dos partes,
8 basándose en el espacio en blanco entre los valores
9 dados por el usuario
10 # Calcular la distancia utilizando la fórmula de
11 distancia Euclidiana
12 distancia = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
13 # Mostrar el resultado
14 print("La distancia entre los puntos es", distancia)

```

Código 3.10: Solución al Ejercicio 3.10.

4

Instrucciones simples y estructuradas

En este capítulo exploraremos los distintos tipos de instrucciones que Python pone a disposición del programador para manejar datos y estructurar programas de manera eficiente.

Python es un lenguaje de alto nivel que facilita la escritura de código legible y conciso. A diferencia de otros lenguajes como C, C++ o Java, Python no requiere la especificación explícita de tipos de datos ni el uso obligatorio de ; al final de cada línea.

4.1. Instrucciones simples

Las instrucciones simples son las acciones básicas que realiza un programa en Python. Estas instrucciones incluyen la declaración y asignación de variables, la entrada y salida de datos, y las invocaciones de funciones y retornos de valores (estas dos últimas se estudiarán en un capítulo posterior). Son los componentes básicos que permiten manipular datos, hacer cálculos y controlar el flujo del programa.

4.1.1. Declaraciones y asignaciones

En Python, la declaración de variables no requiere un tipo específico, ya que Python utiliza un sistema de tipado dinámico. Esto significa que no necesitas especificar si una variable es un número entero, un flotante, una cadena de texto, etc. Python lo infiere automáticamente cuando se asigna un valor a la variable.

Por ejemplo:

```
1 | nombre = "Juan"    # Asigna una cadena de texto a la variable '
  ↪   nombre '
2 | edad = 30          # Asigna un número entero a la variable '
```

4. Instrucciones simples y estructuradas

```
↪      edad '  
3 altura = 1.75      # Asigna un número flotante a la variable '  
↪      altura '  
4 activo = True      # Asigna un valor booleano a la variable '  
↪      activo '
```

En Python, se usa el signo igual = para asignar un valor a una variable. Por ejemplo, en `nombre = "Juan"`, el valor "Juan" se asigna a la variable `nombre`. Recuerda que en Python puedes cambiar el valor de una variable en cualquier momento. Además, si cambia el tipo del nuevo valor asignado a la variable, dicha variable cambiará automáticamente de tipo. Por ejemplo:

```
1 nombre = "Pedro"  # Ahora 'nombre' tiene el valor "Pedro"  
2 nombre = 3        # Ahora 'nombre' es de tipo entero
```

En Python puedes asignar valores a varias variables en una sola línea. Por ejemplo:

```
1 a, b, c = 5, 10, 15 # Asigna 5 a 'a', 10 a 'b', y 15 a 'c'
```

Respecto a las constantes, en Python no existe el concepto de *constante* como tal (a diferencia de una variable, una constante es algo que no varía durante la ejecución del programa). Por tanto, cuando se quiere especificar que se está declarando una constante, por convención, se escribe en mayúsculas. Por ejemplo:

```
1 # Declaración de constantes  
2 PI = 3.141592
```

4.1.2. Expresiones

En Python, las *expresiones* son combinaciones de valores, operadores y funciones que resultan en un único valor. Las expresiones pueden incluir operaciones matemáticas, concatenación de texto, y mucho más.

Por ejemplo, la siguiente línea de código muestra una expresión matemática:

```
1 x = 5 + 3 # La expresión 5 + 3 se evalúa y se asigna el  
↪      valor 8 a la variable 'x'
```

Además de las operaciones matemáticas simples, Python también permite realizar operaciones de asignación combinadas que modifican el valor de una variable.

```
1 x = 10  
2 x += 5 # Equivalente a x = x + 5, ahora x es 15  
3 x *= 2 # Equivalente a x = x * 2, ahora x es 30
```

4.1.3. Entrada y salida de datos

Como ya hemos visto, las funciones `input()` y `print()` permiten interactuar con el usuario, ya sea para obtener información o mostrar resultados.

La función `input()` pide al usuario que ingrese información a través del teclado. El valor ingresado siempre se recibe como una cadena de texto, por lo que si se necesita otro tipo de dato, como un número entero o flotante, hay que convertirlo explícitamente (véase el Capítulo 3).

```
1 nombre = input("¿Cómo te llamas? ")
2 edad = int(input("¿Cuántos años tienes? ")) # Conversión
↪      explícita a entero
```

Respecto a la salida, la función `print()` se utiliza para mostrar información en la pantalla. Puede imprimir cualquier tipo de dato (texto, números, listas, etc.). En el Capítulo 3 se introdujeron las *f-strings* como forma idiomática de incluir variables dentro de cadenas:

```
1 print(f"Hola, {nombre}. Tienes {edad} años.")
```

A continuación se presentan dos opciones de formato útiles que se aplican a menudo durante la entrada/salida de datos: el formato en distintas bases numéricas y la alineación de texto.

Formato en distintas bases numéricas. Mediante un especificador de formato detrás del carácter `:` se puede mostrar un entero en distintas bases (decimal por defecto, octal con `o`, hexadecimal con `x` o `X`, binario con `b`):

```
1 num = 255
2 print(f"Decimal:      {num}")      # 255
3 print(f"Octal:       {num:o}")     # 377
4 print(f"Hexadecimal: {num:x}")     # ff
5 print(f"Binario:     {num:b}")     # 11111111
```

Alineación de texto. Para mejorar la presentación de datos en columnas, las *f-strings* permiten alinear texto a la izquierda (`<`), centrado (`^`) o a la derecha (`>`), indicando un ancho mínimo:

```
1 nombre = "Alice"
2 print(f"|{nombre:<10}|") # |Alice      |
3 print(f"|{nombre:^10}|") # |  Alice  |
4 print(f"|{nombre:>10}|") # |      Alice|
```

Los mismos especificadores se pueden aplicar también a enteros y flotantes, lo que es útil al mostrar tablas de resultados.

4.2. Instrucciones estructuradas

Las instrucciones estructuradas son esenciales para organizar y controlar el flujo de un programa. A través de ellas, podemos tomar decisiones (*instrucciones condicionales*),

repetir acciones (*instrucciones iterativas*) o agrupar varias instrucciones dentro de un mismo bloque (*bloques de código*). Estas estructuras permiten que un programa sea más flexible, eficiente y legible, al evitar la ejecución de código redundante y al tomar decisiones basadas en condiciones dinámicas (tomadas en tiempo de ejecución del programa).

En términos generales, las instrucciones estructuradas permiten crear programas que pueden tomar decisiones y repetir acciones de manera controlada, lo que las hace fundamentales para cualquier tipo de software. Además, el uso adecuado de estas estructuras favorece la modularidad, la claridad y la mantenibilidad del código, que son características deseadas en nuestros programas. A continuación se discuten estas instrucciones estructuradas en más detalle.

4.2.1. Bloques de código: agrupando instrucciones

En Python, los bloques de código se agrupan usando indentación (espacios o tabuladores al inicio de la línea). Un *bloque de código* es un conjunto de instrucciones que se ejecutan de manera secuencial, sin interrupciones, dentro de un mismo nivel de ejecución. Todo el código dentro de un bloque debe estar alineado con el mismo número de espacios o tabulaciones.

La indentación es uno de los elementos distintivos de Python. A diferencia de otros lenguajes que utilizan llaves (como C++ o Java, por ejemplo) para denotar bloques de código, Python usa la indentación para mejorar la legibilidad del código. La indentación obligatoria asegura que el código sea visualmente claro y menos propenso a errores de sintaxis.

Por ejemplo:

```
1 if x > 0:
2     print("El número es positivo") # Esta instrucción se
↪     ejecuta si x > 0
3     y = x * 2 # Esta también se ejecuta si x > 0
```

En este ejemplo, las dos líneas de código dentro del bloque de `if` sólo se ejecutan si la condición `x > 0` es verdadera. La indentación de 4 espacios (por convención) indica que ambas líneas forman parte del mismo bloque.

Los bloques de código también se pueden utilizar en otras estructuras como funciones, bucles, y condiciones para agrupar instrucciones que se deben ejecutar de manera conjunta.

4.2.2. Instrucciones condicionales: tomando decisiones

Las instrucciones condicionales permiten que un programa tome decisiones, es decir, ejecute diferentes bloques de código **según el valor de una condición o expresión booleana**. En Python, las estructuras condicionales más comunes son `if`, `elif` y `else`. Estas instrucciones permiten evaluar una condición y ejecutar un bloque de código si esa condición es verdadera, o ejecutar otro bloque de código si la condición es falsa.

Instrucción if

La instrucción `if` permite evaluar una condición y ejecutar un bloque de código si la condición es verdadera. Si la condición no es verdadera (es decir, si es falsa), el bloque de código bajo el `if` se salta y no se ejecuta.

Su sintaxis es la siguiente:

```
if condicion:
    # Código a ejecutar si condición es verdadera
```

Vemos ahora cómo usarla mediante un ejemplo:

```
1 edad = int(input("¿Cuántos años tienes?"))
2 if edad >= 18:
3     print("Eres mayor de edad.")
```

En este ejemplo, si `edad` es mayor o igual a 18, se ejecuta el bloque de código dentro del `if`, mostrando el mensaje "Eres mayor de edad". Si la condición no se cumple (por ejemplo, si `edad` es 17), entonces el bloque dentro del `if` no se ejecuta y el programa continúa. Las condiciones pueden ser más complejas, como expresiones aritméticas, comparaciones entre variables, o incluso la evaluación de resultados booleanos.

Instrucción elif

La instrucción `elif` (abreviatura de “*else if*”) se utiliza cuando tienes más de una condición que verificar. Después de un `if`, si la condición del `if` es falsa, Python evalúa las condiciones de los bloques `elif` existentes. Puedes tener varios `elif` seguidos, cada uno con su propia condición. De hecho, puedes tener tantos bloques `elif` como necesites, lo que permite comprobar muchas condiciones diferentes. Después de un `if`, puedes añadir tantos `elif` como quieras. Puedes incluso no tener ninguna instrucción `elif`, dependerá del problema particular que estés resolviendo. Eso sí, una instrucción `elif` siempre tendrá una instrucción `if` previa. A este tipo de estructuras condicionales también se les conoce como *condicionales múltiples*.

Su sintaxis es la siguiente:

```
if condicion1:
    # Código si condición1 es verdadera
elif condicion2:
    # Código si condición2 es verdadera
```

Vemos ahora cómo usarla mediante un ejemplo:

```
1 edad = int(input("¿Cuántos años tienes?"))
2 if edad >= 18:
3     print("Eres mayor de edad.")
4 elif edad >= 13:
5     print("Eres un adolescente.")
```

4. Instrucciones simples y estructuradas

En este caso, si la primera condición (`edad >= 18`) no es verdadera, Python evalúa la condición del `elif` (`edad >= 13`). Supón que el usuario introduce el valor 16. Entonces, como `edad` es 16, se ejecutará el bloque de código dentro del `elif`, mostrando "Eres un adolescente".

Instrucción `else`: bloque de código alternativo

El bloque `else` es opcional y se ejecuta si todas las condiciones anteriores (de `if` y `elif` que haya) son falsas. El bloque `else` no lleva condición, ya que es un bloque de código por defecto que se ejecuta si no se ha ejecutado ningún bloque de código anterior.

Su sintaxis es la siguiente:

```
if condicion1:
    # Código si condición1 es verdadera
elif condicion2:
    # Código si condición2 es verdadera
else:
    # Código si ninguna de las condiciones anteriores es
    verdadera
```

Vemos ahora cómo usarla mediante un ejemplo:

```
1 edad = int(input("¿Cuántos años tienes?"))
2 if edad >= 18:
3     print("Eres mayor de edad.")
4 elif edad >= 13:
5     print("Eres un adolescente.")
6 else:
7     print("Eres un infante.")
```

Supón que el usuario introduce el valor 10. Entonces, como `edad` es 10, no cumple con ninguna de las condiciones del `if` ni del `elif`, por lo que el bloque de código dentro del `else` se ejecuta, mostrando "Eres un infante".

Combinando condiciones

Puedes combinar varias condiciones utilizando operadores lógicos como `and`, `or`, y `not` para crear condiciones más complejas dentro de un `if`, `elif` o `else`.

Recuerda que estos operadores se vieron en el Capítulo 3 y tienen tablas de verdad distintas. El operador `and` se utiliza cuando todas las condiciones deben ser verdaderas, el operador `or` se utiliza cuando al menos una de las condiciones debe ser verdadera, y el operador `not` invierte el resultado de la condición (es decir, si la condición es verdadera, `not` la convierte en falsa, y viceversa).

Por ejemplo:


```
1 edad = 20
2 tiene_permiso = False
3
4 if edad >= 18 and tiene_permiso:
5     print("Puedes entrar.")
6 else:
7     print("No puedes entrar.")
```

En este caso, la persona debe ser mayor de edad y tener permiso. Si cualquiera de estas condiciones no se cumple, se ejecutará el bloque `else`.

Condicionales anidados: condiciones dentro de condiciones

Las condiciones anidadas son cuando tienes una estructura condicional dentro de otra. Esto te permite realizar decisiones más complejas. Los bloques `if`, `elif` y `else` pueden contener otras estructuras condicionales.

Por ejemplo:

```
1 edad = 15
2 tiene_permiso = True
3
4 if edad >= 18:
5     print("Eres mayor de edad.")
6 else:
7     if tiene_permiso:
8         print("Eres menor de edad, pero tienes permiso.")
9     else:
10        print("Eres menor de edad y no tienes permiso.")
```

En este ejemplo, si la condición `edad >= 18` no se cumple, el bloque `else` se ejecuta y dentro de él, hay un `if` que evalúa si la persona tiene permiso (representado en la variable `tiene_permiso`). Dependiendo de la respuesta, se imprime un mensaje distinto.

4.2.3. Instrucciones iterativas: repitiendo acciones

Las estructuras iterativas son una parte fundamental en la programación, ya que permiten que un programa ejecute un bloque de código de manera repetitiva, basándose en una condición o un conjunto de datos. La capacidad de repetir acciones es imprescindible para cualquier programa que necesite realizar tareas de manera sistemática y eficiente, como recorrer listas, procesar datos o realizar cálculos repetitivos. En Python, existen dos tipos de estructuras iterativas principales: los bucles `while` y `for`.

Bucle `while`: ejecutar mientras se cumpla una condición

El bucle `while` es una estructura iterativa que repite un bloque de código mientras se cumpla una condición especificada. La condición se evalúa antes de cada iteración, y

4. Instrucciones simples y estructuradas

si es `True` (verdadera), se ejecuta el bloque de código dentro del bucle. Si la condición es `False` (falsa), el bucle se detiene y el programa continúa con la siguiente instrucción.

Su sintaxis es la siguiente:

```
↪ while condicion:
    # Código que se ejecuta mientras la condición sea
    verdadera
```

Lo que acompaña a `while`, `condicion`, es una expresión booleana que se evalúa en cada iteración del bucle. Mientras esta condición siga siendo verdadera, el bucle continuará ejecutándose. Si la condición se vuelve falsa, el bucle terminará. Se denomina *código dentro del bucle* al bloque de instrucciones que se ejecuta cada vez que la condición se evalúa como verdadera (que está indentando debajo de `while`).

Veamos cómo se usa con un ejemplo sencillo:

```
1 contador = 1
2 while contador <= 5:
3     print(contador)
4     contador += 1 # Incrementa el contador en 1
```

En la línea 1 se está declarando una variable entera denominada `contador` e inicializada a valor 1. Después, aparece un bloque `while`. La condición `contador <= 5` se evalúa antes de cada iteración. Mientras `contador` sea menor o igual a 5, el programa imprimirá el valor de `contador` y lo incrementará en cada iteración. Una vez que `contador` llegue a 6, la condición `contador <= 5` será `False`, y el bucle se detendrá.

Este tipo de bucles es útil cuando no sabemos cuántas veces necesitamos repetir una acción y la ejecución depende de una condición dinámica que cambia durante la ejecución del programa.

Un error común al usar el bucle `while` es no actualizar adecuadamente la condición en cada iteración, en el código dentro del bucle. Si la condición nunca se vuelve falsa, el bucle continuará ejecutándose indefinidamente, creando un bucle infinito. Por ejemplo:

```
1 while True:
2     print("Esto se repite para siempre")
```

Este bucle nunca terminará a menos que intervengas manualmente (por ejemplo, pulsando `Ctrl`+`C` desde la terminal en Windows, Linux o macOS).

Bucle `for`: repetir un bloque de código un número determinado de veces

El bucle `for` en Python es una de las estructuras iterativas más utilizadas. A diferencia del `while`, el `for` se utiliza cuando sabemos cuántas veces queremos repetir un bloque de código, o cuando queremos iterar sobre una secuencia de elementos, como una lista, una cadena de texto, un rango de números, o cualquier otro tipo de colección.

Su sintaxis es la siguiente:

```
for variable in secuencia:
    # Código a ejecutar para cada elemento de la secuencia
```

En este bucle, `variable` es una variable que toma el valor de cada elemento de la `secuencia` en cada iteración del bucle. El elemento `secuencia` puede ser una lista, una tupla, un rango, una cadena de texto o cualquier otro objeto iterable. El bucle ejecuta el bloque de código para cada elemento de la secuencia, asignando ese elemento a `variable` en cada iteración.

Veamos un ejemplo sencillo:

```
1 frutas = ["manzana", "banana", "cereza", "pera"]
2 for fruta in frutas:
3     print(f"Me gusta la {fruta}")
```

En la primera línea de código se define una lista de cadenas (el tipo de dato lista se estudiará en capítulos posteriores). El bucle `for` recorre la lista `frutas` y, en cada iteración, asigna a la variable `fruta` uno de los elementos de la lista (primero `"manzana"`, luego `"banana"`, después `"cereza"` y por último `"pera"`). En cada iteración se ejecuta el `print`, mostrando sucesivamente los mensajes `"Me gusta la manzana"`, `"Me gusta la banana"`, `"Me gusta la cereza"` y `"Me gusta la pera"`.

Una de las formas más comunes de usar un bucle `for` en Python es iterando sobre un rango de números. Para ello, es interesante que conozcas la función `range()`, que genera una secuencia de números y puedes usarla para controlar el número de iteraciones.

Su sintaxis concretamente es la siguiente:

```
range(start, stop, step)
```

Donde `start` es el número desde el que comienza el rango (por defecto es 0), `stop` es el número en el que termina el rango (este valor no se incluye en el rango) y `step` es el tamaño del paso entre cada número del rango (por defecto es 1).

Veamos un ejemplo sencillo:

```
1 for i in range(1, 6):
2     print(i)
```

Este bucle imprimirá los números del 1 al 5, ya que tamaño del paso por defecto (es decir, si no se explicita) es 1. El número 6 no se incluye porque, tal y como se ha descrito anteriormente, `range()` genera una secuencia hasta, pero sin incluir, el valor de `stop`.

Veamos otro ejemplo sencillo:

```
1 for i in range(0, 10, 2):
2     print(i)
```

Este bucle imprimirá los números 0, 2, 4, 6, 8, porque el `step` es 2, lo que significa que el bucle saltará de 2 en 2.

Control del flujo de ejecución en bucles: `break` y `continue`

Dentro de los bucles, puedes utilizar las instrucciones `break` y `continue` para controlar el flujo de ejecución.

4. Instrucciones simples y estructuradas

Concretamente, `break` se usa para salir inmediatamente de un bucle, independientemente de si la condición sigue siendo verdadera o si quedan elementos por recorrer. Por ejemplo:

```
1 for i in range(1, 6):
2     if i == 3:
3         break # Sale del bucle cuando i es 3
4     print(i)
```

Este código imprimirá 1, 2 y luego saldrá del bucle al evaluar la condición `i == 3`.

Por otro lado, `continue` se usa para saltar a la siguiente iteración del bucle, omitiendo el resto del código en el bloque de la iteración actual. Por ejemplo:

```
1 for i in range(1, 6):
2     if i == 3:
3         continue # Salta esta iteración cuando i es 3
4     print(i)
```

Este código imprimirá 1, 2, 4, 5. Cuando la variable `i` tiene el valor 3, se cumple la condición que acompaña al `if`, con lo que se ejecuta el `continue`, que hace que el bucle pase a la siguiente iteración sin ejecutar nada del resto de código dentro del bucle (en este ejemplo, el `print()`).

Cláusula `else` en bucles

A diferencia de muchos otros lenguajes, en Python tanto `for` como `while` admiten una cláusula `else` que se ejecuta **solo si el bucle termina de forma natural**, es decir, sin haber ejecutado un `break`. Esto resulta útil para distinguir entre los dos motivos por los que un bucle de búsqueda puede haber terminado: porque encontró lo que buscaba (en cuyo caso se interrumpe con `break`), o porque agotó la secuencia sin encontrarlo. Veamos un ejemplo:

```
1 numeros = [4, 8, 12, 16, 20]
2 buscado = 11
3
4 for n in numeros:
5     if n == buscado:
6         print(f"{buscado} encontrado")
7         break
8 else:
9     print(f"{buscado} no está en la lista")
```

Si `buscado` aparece en `numeros`, el `break` interrumpe el bucle y el bloque `else` no se ejecuta. Si no aparece, el bucle termina de forma natural y se ejecuta el `else`, mostrando que el valor no estaba en la lista.

Iteraciones idiomáticas: enumerate y zip

Cuando se itera sobre una secuencia, a veces se necesita conocer también la *posición* de cada elemento. Aunque podría hacerse manteniendo un contador manual, lo idiomático en Python es usar la función `enumerate()`, que devuelve pares (índice, elemento):

```
1 frutas = ["manzana", "banana", "cereza"]
2 for i, fruta in enumerate(frutas):
3     print(f"{i}: {fruta}")
4 # Salida:
5 # 0: manzana
6 # 1: banana
7 # 2: cereza
```

Si se quiere comenzar el índice en otro valor (por ejemplo, 1), se pasa con el parámetro `start`:

```
1 for i, fruta in enumerate(frutas, start=1):
2     print(f"{i}: {fruta}") # 1: manzana, 2: banana, ...
```

Cuando se necesita iterar sobre dos (o más) secuencias en paralelo, la función `zip()` las combina elemento a elemento:

```
1 nombres = ["Alice", "Bob", "Charlie"]
2 edades = [30, 25, 35]
3 for nombre, edad in zip(nombres, edades):
4     print(f"{nombre} tiene {edad} años")
```

Si las secuencias tienen distinta longitud, `zip` se detiene en la más corta sin lanzar error.

Combinación de bucles y condicionales

Los bucles `while` y `for` y las estructuras condicionales `if` se combinan habitualmente para crear ciclos de procesamiento más complejos. Es común que dentro de un bucle se evalúen condiciones para decidir qué hacer en cada iteración. Por ejemplo, el siguiente código combina `for` con `if` para recorrer una lista de números y mostrar si cada uno es par o impar:

```
1 numeros = [1, 2, 3, 4, 5]
2 for numero in numeros:
3     if numero % 2 == 0:
4         print(f"{numero} es par")
5     else:
6         print(f"{numero} es impar")
```

Combinando los recursos vistos en las subsecciones anteriores se pueden expresar tareas más ricas. Por ejemplo, el siguiente fragmento numera una lista de palabras y, mediante un `if` con `continue`, omite las palabras vacías:

4. Instrucciones simples y estructuradas

```
1 palabras = ["hola", "", "python", "", "mundo"]
2 for i, palabra in enumerate(palabras, start=1):
3     if palabra == "":
4         continue
5     print(f"Palabra {i}: {palabra}")
```

4.3. Ejercicios

EJERCICIO 4.1

Escribe un programa en Python que pida al usuario su edad y determine a qué categoría pertenece:

- Menor de edad (0-17 años)
- Adulto (18-64 años)
- Adulto mayor (65 años o más)

Ejemplo de ejecución esperada

Introduce tu edad: 20
Eres un adulto.

Solución (girar para leer)

```
1 edad = int(input("Introduce tu edad: "))
2
3 if edad > 18:
4     print("Eres un menor de edad.")
5 elif edad > 65:
6     print("Eres un adulto.")
7 else:
8     print("Eres un adulto mayor.")
```

Código 4.1: Solución al Ejercicio 4.1.

EJERCICIO 4.2

Escribe un programa en Python que pida un número entero al usuario y determine si el número es par o impar.

Ejemplo de ejecución esperada

Introduce un número: 7
El número es impar.

Solución (girar para leer)

Código 4.2: Solución al Ejercicio 4.2.

```

1 numero = int(input("Introduce un número: "))
2
3 if numero % 2 == 0:
4     print("El número es par.")
5 else:
6     print("El número es impar.")

```

EJERCICIO 4.3

Escribe un programa en Python que calcule el impuesto a pagar en función del salario. Se aplicarán las siguientes reglas:

- Si el salario es menor de 2000, no se paga impuesto.
- Si el salario es mayor o igual a 2000 pero menor de 5000, el impuesto será del 10 %.
- Si el salario es mayor o igual a 5000 pero menor de 10000, el impuesto será del 15 %.
- Si el salario es mayor o igual a 10000 pero menor de 20000, el impuesto será del 20 %.
- Si el salario es mayor o igual a 20000, el impuesto será del 30 %.

El programa debe solicitar al usuario el valor de su salario y calcular el impuesto a pagar según las reglas mencionadas. Además, debe mostrar el monto del impuesto y el salario final después de la deducción. Todos los números reales se mostrarán con precisión de dos decimales.

Ejemplo de ejecución esperada

```
Introduce tu salario: 4500
El impuesto a pagar es 450.0
Tu salario final es 4050.0
```


Solución (girar para leer)

```

1 salario = float(input("Introduce tu salario: "))
2
3 if salario < 2000:
4     impuesto = 0
5     elif salario < 5000:
6         impuesto = salario * 0.10
7     elif salario < 10000:
8         impuesto = salario * 0.15
9     elif salario < 20000:
10        impuesto = salario * 0.20
11    else:
12        impuesto = salario * 0.30
13
14 salario_final = salario - impuesto
15
16 print(f"El impuesto a pagar es {impuesto:.2f}")
17 print(f"Tu salario final es {salario_final:.2f}")

```

Código 4.3: Solución al Ejercicio 4.3.

4. Instrucciones simples y estructuradas

EJERCICIO 4.4

Escribe un programa en Python que determine si una persona puede obtener la licencia de conducir. La persona debe ser mayor de 18 años. El programa debe solicitar la edad del usuario y mostrar si puede o no obtener la licencia.

Ejemplo de ejecución esperada

Introduce tu edad: 17

No puedes obtener la licencia de conducir.

Solución (girar para leer)

```
1 edad = int(input("Introduce tu edad: "))
2
3 if edad >= 18:
4     print("Puedes", end=' ');
5     # con end especificamos el carácter con el que
6     acabar la línea
7     # por defecto, el print inserta un salto de línea
8     al final
9     print("obtener la licencia de conducir.")
10 else:
11     print("No puedes", end=' ');
```

Código 4.4: Solución al Ejercicio 4.4.

EJERCICIO 4.5

Escribe un programa en Python que pida la calificación de un examen (un número entre 0 y 10) y determine si el estudiante ha aprobado o no. La calificación mínima para aprobar es 5.

Ejemplo de ejecución esperada

Introduce tu calificación: 4
Has suspendido el examen.

Solución (girar para leer)

```

1 calificacion = float(input("Introduce tu calificación:
2
3 print("Has ", end='')
4 if calificacion >= 5:
5     print("aprobado", end='')
6 else:
7     print("suspendido", end='')
8 print(" el examen.")

```

Código 4.5: Solución al Ejercicio 4.5.

4. Instrucciones simples y estructuradas

EJERCICIO 4.6

Escribe un programa en Python que pida un número al usuario y determine si el número está en el rango entre 10 y 100 (inclusive).

Ejemplo de ejecución esperada

```
Introduce un número: 50
El número está en el rango.
```

Solución (girar para leer)

```
1 numero = int(input("Introduce un número: "))
2
3 print("El número", end=' ')
4 if not (10 <= numero <= 100):
5     print("no", end=' ')
6     print("está en el rango.")
```

Código 4.6: Solución al Ejercicio 4.6.

EJERCICIO 4.7

Escribe un programa en Python que pida tres longitudes de lados y determine si pueden formar un triángulo válido. Un triángulo es válido si la suma de las longitudes de dos lados siempre es mayor que la longitud del tercer lado.

Ejemplo de ejecución esperada

```
Introduce el primer lado: 3
Introduce el segundo lado: 4
Introduce el tercer lado: 5
Los lados forman un triángulo válido.
```

Solución (girar para leer)

```

1 lado1 = float(input("Introduce el primer lado: "))
2 lado2 = float(input("Introduce el segundo lado: "))
3 lado3 = float(input("Introduce el tercer lado: "))
4
5 if lado1 + lado2 < lado3 and \
6    lado1 + lado3 < lado2 and \
7    lado2 + lado3 < lado1:
8     # el carácter \ lo usamos para poder partir la
9     # expresión en
10    # varias líneas y que no se rompa la condición al
11    leerla
12 else:
13     print("Los lados forman un triángulo válido.")
14     print("Los lados no forman un triángulo válido.")
```

Código 4.7: Solución al Ejercicio 4.7.

EJERCICIO 4.8

Escribe un programa en Python que pida un número entre 1 y 12, correspondiente a un mes del año, y muestre el nombre del mes correspondiente (por ejemplo, 1 = enero, 2 = febrero, etcétera).

Ejemplo de ejecución esperada

Introduce el número del mes: 3

El mes es marzo.

Solución (girar para leer)

```

1 mes = int(input("Introduce el número del mes: "))
2
3 if mes == 1:
4     print("El mes es enero.")
5     if mes == 2:
6         print("El mes es febrero.")
7         if mes == 3:
8             print("El mes es marzo.")
9             if mes == 4:
10                print("El mes es abril.")
11                if mes == 5:
12                    print("El mes es mayo.")
13                    if mes == 6:
14                        print("El mes es junio.")
15                        if mes == 7:
16                            print("El mes es julio.")
17                            if mes == 8:
18                                print("El mes es agosto.")
19                                if mes == 9:
20                                    print("El mes es septiembre.")
21                                    if mes == 10:
22                                        print("El mes es octubre.")
23                                        if mes == 11:
24                                            print("El mes es noviembre.")
25                                            if mes == 12:
26                                                print("El mes es diciembre.")
27 else:
28     print("Número de mes inválido.")

```

Código 4.8: Solución al Ejercicio 4.8.

EJERCICIO 4.9

Escribe un programa en Python que calcule la calificación final de un estudiante en una asignatura. El estudiante tiene una calificación de examen y una nota de asistencia. La calificación final se determina con la siguiente lógica:

- Si la nota de examen es menor de 5, el estudiante aprueba o suspende automáticamente dependiendo de la nota de asistencia:
 - Si la nota de asistencia es mayor o igual a 7, el estudiante aprueba el curso con una calificación de 5.
 - Si la nota de asistencia es menor a 7, el estudiante suspende automáticamente con una calificación de 0, independientemente del examen.
- Si la nota de examen es mayor o igual a 5, el estudiante aprueba el curso si su nota de asistencia es mayor o igual a 5. En este caso, la calificación final será el promedio entre la nota de examen y la nota de asistencia.
- Si la nota de asistencia es menor a 5 y el examen es mayor o igual a 5, el estudiante aprobará pero con una calificación final mínima de 5.

El programa debe pedir al usuario las notas de examen y de asistencia y mostrar la calificación final.

Ejemplo de ejecución esperada

```
Introduce la nota del examen: 6
Introduce la nota de asistencia: 4
La calificación final es 5.0.
```

Solución (girar para leer)

```

1 examen = float(input("Introduce la nota del examen: "))
2 asistencia = float(input("Introduce la nota de
3 asistencia: "))
4 # Definir la calificación final
5 calificacion_final = 0
6
7 # Comprobar la calificación en función de las reglas
8 if examen > 5:
9     if asistencia >= 7:
10        calificacion_final = 5
11    else:
12        calificacion_final = 0
13    elif examen >= 5:
14        if asistencia >= 5:
15            calificacion_final = (examen + asistencia) / 2
16        else:
17            calificacion_final = 5
18
19 print(f"La calificación final es {calificacion_final:.1
20 f}")

```

Código 4.9: Solución al Ejercicio 4.9.

EJERCICIO 4.10

Escribe un programa en Python que calcule la bonificación de un empleado en función de su años de experiencia y su desempeño. Las reglas son las siguientes:

- Si el empleado tiene menos de 1 año de experiencia, no recibe bonificación.
- Si el empleado tiene 1 o más años de experiencia:
 - Si su desempeño es excelente, la bonificación será el 20 % de su salario.
 - Si su desempeño es bueno, la bonificación será el 10 % de su salario.
 - Si su desempeño es regular, la bonificación será el 5 % de su salario.
- Si el empleado tiene más de 10 años de experiencia y su desempeño es bueno o excelente, se le dará una bonificación extra de 500 EUR.
- Si el salario es menor a 2000 EUR, la bonificación será ajustada a un máximo de 200 EUR, independientemente de los cálculos anteriores.

El programa debe solicitar al usuario los siguientes datos:

- Años de experiencia.
- Desempeño (como una cadena de texto: “excelente”, “bueno”, “regular”).
- Salario.

El programa debe calcular la bonificación y mostrarla.

Ejemplo de ejecución esperada

```
Introduce tus años de experiencia: 12
Introduce tu desempeño (excelente, bueno, regular): bueno
Introduce tu salario: 3500
La bonificación es 850.0 EUR.
```

Solución (girar para leer)

```

1 # Solicitar los datos al usuario
2 anos-experiencia = int(input("Introduce tus años de
3 experiencia: "))
4 salario = float(input("Introduce tu salario: "))
5 # Inicializar bonificación
6 bonificacion = 0
7 # Calcular bonificación basada en años de experiencia
8 if anos-experiencia < 1:
9     bonificacion = 0
10 # No bonificación si tiene menos
11     de 1 año de experiencia
12 else:
13     if desempeño == "excelente":
14         bonificacion = salario * 0.20 # 20% de
15         bonificación por desempeño excelente
16     elif desempeño == "bueno":
17         bonificacion = salario * 0.10 # 10% de
18         bonificación por desempeño bueno
19     elif desempeño == "regular":
20         bonificacion = salario * 0.05 # 5% de
21         bonificación por desempeño regular
22 # Si el empleado tiene más de 10 años de
23     experiencia y un buen desempeño, bonificación
24     extra
25     if anos-experiencia > 10 and (desempeno == "bueno"
26     or desempeño == "excelente"):
27         bonificacion += 500 # Bonificación extra por
28         muchos años de experiencia
29 # Ajustar bonificación máxima si el salario es menor a
30     2000
31 if salario > 2000:
32     bonificacion = min(bonificacion, 200) # Limitar
33     bonificación a 200 EUR si el salario es bajo
34 # Mostrar la bonificación final
35 print(f"La bonificación es {bonificacion} EUR.")

```

Código 4.10: Solución al Ejercicio 4.10.

EJERCICIO 4.11

Escribe un programa en Python que utilice un bucle `while` para contar desde 1 hasta 10, imprimiendo cada número en cada iteración por pantalla, separados por el carácter coma .

Ejemplo de ejecución esperada

1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Solución (girar para leer)

```
1 contador = 1
2 while contador <= 10:
3     print(contador, end=', ', )
4     contador += 1
```

Código 4.11: Solución al Ejercicio 4.11.

EJERCICIO 4.12

Escribe un programa que pida números al usuario de forma repetida hasta que el usuario introduzca el número 0. El programa debe mostrar la suma total de los números introducidos (sin contar el 0).

Ejemplo de ejecución esperada

```
Introduce un número: 5
Introduce un número: 3
Introduce un número: 2
Introduce un número: 0
La suma total es 10.
```

Solución (girar para leer)

```
1 suma_total = 0
2 while True:
3     numero = int(input("Introduce un número: "))
4     if numero == 0:
5         break
6     suma_total += numero
7
8 print(f"La suma total es {suma_total}.")
```

Código 4.12: Solución al Ejercicio 4.12.

EJERCICIO 4.13

Escribe un programa en Python que permita al usuario introducir varios números. El programa debe calcular y mostrar el promedio de los números introducidos. El usuario deberá escribir 0 para terminar de introducir números.

Ejemplo de ejecución esperada

```
Introduce un número: 5
Introduce un número: 3
Introduce un número: 10
Introduce un número: 0
El promedio es 6.0.
```

Solución (girar para leer)

```

1 suma = 0
2 contador = 0
3
4 while True:
5     numero = float(input("Introduce un número: "))
6     if numero == 0:
7         break
8     suma += numero
9     contador += 1
10
11 if contador > 0:
12     promedio = suma / contador
13     print(f"El promedio es {promedio}.")
14 else:
15     print("No se introdujeron números.")

```

Código 4.13: Solución al Ejercicio 4.13.

EJERCICIO 4.14

Escribe un programa que le pida al usuario que adivine un número secreto entre 1 y 100. El programa debe seguir pidiendo al usuario que ingrese un número hasta que adivine correctamente el número secreto o hasta que se acaben el número máximo de intentos permitidos (valor 5). El programa debe informar si el número ingresado es mayor o menor que el número secreto.

Para generar un número de manera pseudoaleatoria dentro de un rango definido en Python, puedes usar el módulo `random`:

```
1 import random
2 numero_secreto = random.randint(1, 100)
3 # Genera un número pseudoaleatorio entre 1 y 100
```

Ejemplo de ejecución esperada

```
Adivina el número entre 1 y 100 (1/5): 50
Demasiado alto.
Adivina el número entre 1 y 100 (2/5): 25
Demasiado bajo.
Adivina el número entre 1 y 100 (3/5): 40
¡Correcto! El número era 40.
```

Solución (girar para leer)

```

1 import random
2
3 # Número secreto aleatorio entre 1 y 100
4 numero_secreto = random.randint(1, 100)
5 intentos = 0
6 max_intentos = 5
7
8 while intentos < max_intentos:
9     intento = int(input("Adivina el número entre 1 y
10         intentos + 1}/{max_intentos} : "))
11     intentos += 1
12     if intento == numero_secreto:
13         print("¡Correcto! El número era {
14             numero_secreto}.")
15         break # Termina el bucle si adivinó el número
16     elif intento > numero_secreto:
17         print("Demasiado bajo.")
18         continue # Continúa al siguiente intento si el
19             número es bajo
20     else:
21         print("Demasiado alto.")
22         continue # Continúa al siguiente intento si el
23             número es alto
24     {numero_secreto}.")

```

Código 4.14: Solución al Ejercicio 4.14.

EJERCICIO 4.15

Escribe un programa que pida un número entero al usuario y cuente cuántos dígitos tiene ese número, utilizando un bucle `while`.

Ejemplo de ejecución esperada

Introduce un número: 12345

El número tiene 5 dígitos.

Solución (girar para leer)

```
1 numero = int(input("Introduce un número: "))
2 contador_dígitos = 0
3
4 while numero > 0:
5     numero = numero // 10
6     contador_dígitos += 1
7
8 print(f"El número tiene {contador_dígitos} dígitos.")
```

Código 4.15: Solución al Ejercicio 4.15.

EJERCICIO 4.16

Escribe un programa que calcule el factorial de un número introducido por el usuario utilizando un bucle `while`. El factorial de un número n es el producto de todos los números enteros positivos desde 1 hasta n ($n! = n(n-1)(n-2)(n-3)\dots \cdot 3 \cdot 2 \cdot 1 = n \cdot (n-1)!$).

Ejemplo de ejecución esperada

Introduce un número: 5
El factorial de 5 es 120.

Solución (girar para leer)

Código 4.16: Solución al Ejercicio 4.16.

```

1 numero = int(input("Introduce un número: "))
2 factorial = 1
3
4 while numero > 1:
5     factorial *= numero
6     numero -= 1
7
8 print(f"El factorial es {factorial}.")
```

EJERCICIO 4.17

Escribe un programa que imprima los primeros n números de la serie Fibonacci. El usuario debe ingresar el valor de n , y el programa debe mostrar los primeros n números de la serie. Recuerda que los primeros dos números de Fibonacci son 0 y 1, y cada número siguiente es la suma de los dos anteriores (es decir, el tercer número de Fibonacci es 1, el cuarto 2, el quinto 3, el sexto 5, y así sucesivamente).

Ejemplo de ejecución esperada

```
Introduce un número: 6
Los primeros 6 números de Fibonacci son:
0, 1, 1, 2, 3, 5,
```

Solución (girar para leer)

```
1 n = int(input("Introduce un número: "))
2 a, b = 0, 1
3 contador = 0
4
5 print("Los primeros", n, "números de Fibonacci son:")
6 while contador < n:
7     print(a, end=" ")
8     a, b = b, a + b # Actualiza los valores de a y b
9     contador += 1
```

Código 4.17: Solución al Ejercicio 4.17.

EJERCICIO 4.18

Escribe un programa en Python que pida una frase al usuario y cuente cuántas palabras contiene. El programa debe dividir la frase en palabras (usando el espacio como delimitador) y contar cuántas palabras hay.

Ejemplo de ejecución esperada

Introduce una frase: Esto es un ejercicio de programación
La frase tiene 6 palabras.

Solución (girar para leer)

```

1 frase = input("Introduce una frase: ")
2 palabras = frase.split() # Divide la frase en palabras
3 contador_palabras = 0
4
5 while contador_palabras < len(palabras):
6     contador_palabras += 1
7
8 print(f"La frase tiene {contador_palabras} palabras.")

```

Código 4.18: Solución al Ejercicio 4.18.

EJERCICIO 4.19

Escribe un programa que haga un juego de adivinanza, en el que el usuario debe adivinar un número secreto entre 1 y 100. El programa debe permitir un máximo de 5 intentos. Si el usuario no adivina el número dentro de esos 5 intentos, el programa debe mostrar un mensaje diciendo que se ha agotado el número de intentos. Además, si el usuario adivina el número, el programa debe felicitarlo y terminar el juego.

Si el número ingresado es mayor o menor que el número secreto, el programa debe decirle al usuario si el número es mayor o menor, y permitirle seguir intentando.

Condiciones:

- Si el usuario adivina el número, el juego termina de inmediato usando `break`.
- Si el número ingresado es mayor o menor que el número secreto, se muestra el mensaje correspondiente y el programa continúa pidiendo un nuevo intento usando `continue`.

Ejemplo de ejecución esperada

```
Adivina el número entre 1 y 100: 50
Demasiado alto.
Adivina el número entre 1 y 100: 25
Demasiado bajo.
Adivina el número entre 1 y 100: 30
¡Correcto! El número era 30.
```

Solución (girar para leer)

```

1 import random
2
3 # Número secreto aleatorio entre 1 y 100
4 numero_secreto = random.randint(1, 100)
5 intentos = 0
6 max_intentos = 5
7
8 while intentos < max_intentos:
9     intento = int(input("Adivina el número entre 1 y
10     intentos += 1
11
12     if intento == numero_secreto:
13         print("¡Correcto! El número era {
14             numero_secreto}")
15         break # Termina el bucle si adivinó el número
16     elif intento > numero_secreto:
17         print("Demasiado bajo.")
18         continue # Continúa al siguiente intento si el
19         número es bajo
20     else:
21         print("Demasiado alto.")
22         continue # Continúa al siguiente intento si el
23         número es alto
24     {numero_secreto}")

```

Código 4.19: Solución al Ejercicio 4.19.

EJERCICIO 4.20

Escribe un programa que pida una letra al usuario y una frase al usuario, y luego cuente cuántas palabras comienzan con una letra específica. El programa debe seguir pidiendo palabras hasta que el usuario escriba "fin" para terminar. Si una palabra comienza con la letra que el usuario ha especificado, debe aumentar el contador.

Condiciones:

- El bucle debe continuar hasta que el usuario ingrese "fin" (sin contarla como una palabra).
- Si el usuario ingresa "fin", el programa termina con el número de palabras que comienzan con la letra especificada.

Ejemplo de ejecución esperada

```
Introduce la letra con la que deben comenzar las palabras: b
Introduce una palabra (o 'fin' para terminar): manzana
Introduce una palabra (o 'fin' para terminar): banana
Introduce una palabra (o 'fin' para terminar): flor
Introduce una palabra (o 'fin' para terminar): fin
Las palabras que comienzan con 'b' son: 1.
```

Solución (girar para leer)

```

1  letra_especifica = input("Introduce la letra con la que
2  deben comenzar las palabras: ").lower()
3  contador = 0
4  while True:
5      palabra = input("Introduce una palabra (o 'fin'
6      para terminar): ").lower()
7      if palabra == "fin":
8          break # Termina el bucle si el usuario escribe
9              'fin'
10     if palabra[0] == letra_especifica:
11         contador += 1 # Aumenta el contador si la
12         palabra empieza con la letra especificada
13     print(f"Las palabras que comienzan con '{
14     letra_especifica}' son: {contador}.")

```

Código 4.20: Solución al Ejercicio 4.20.

4. Instrucciones simples y estructuradas

EJERCICIO 4.21

Escribe un programa en Python que calcule la suma de los primeros 10 números enteros positivos utilizando un bucle `for`.

Ejemplo de ejecución esperada

La suma de los primeros 10 números es 55.

Solución (girar para leer)

```
1 suma = 0
2 for i in range(1, 11):
3     suma += i
5 print(f"La suma de los primeros 10 números es {suma}.")
```

Código 4.21: Solución al Ejercicio 4.21.

EJERCICIO 4.22

Escribe un programa que pida un número al usuario e imprima la tabla de multiplicar del número dado por el usuario utilizando un bucle `for`.

Ejemplo de ejecución esperada

```
Introduce un número: 5
La tabla de multiplicar de 5 es:
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Solución (girar para leer)

```

5 | print(f"{numero} x {i} = {numero * i}")
4 | for i in range(1, 11):
3 |     print(f"La tabla de multiplicar de {numero} es:")
2 |
1 | numero = int(input("Introduce un número: "))
Código 4.22: Solución al Ejercicio 4.22.
```

4. Instrucciones simples y estructuradas

EJERCICIO 4.23

→ Escribe un programa en Python que imprima todas las palabras de una lista utilizando un bucle `for`. Considera la siguiente lista de palabras: `['manzana', 'banana', 'cereza', 'naranja', 'kiwi']`

Ejemplo de ejecución esperada

```
Palabra 1: manzana
Palabra 2: banana
Palabra 3: cereza
Palabra 4: naranja
Palabra 5: kiwi
```

Solución (girar para leer)

```
1 palabras = ['manzana', 'banana', 'cereza', 'naranja', 'kiwi']
2
3 for i, palabra in enumerate(palabras, 1):
4     print(f"Palabra {i}: {palabra}")
```

Código 4.23: Solución al Ejercicio 4.23.

EJERCICIO 4.24

Escribe un programa que invierta una lista de números utilizando un bucle `for`. El programa debe imprimir la lista original y la lista invertida.

Ejemplo de ejecución esperada

Lista original: [1, 2, 3, 4, 5]

Lista invertida: [5, 4, 3, 2, 1]

Solución (girar para leer)

```

1 numeros = [1, 2, 3, 4, 5]
2 numeros_invertidos = []
3
4 for numero in numeros:
5     numeros_invertidos.insert(0, numero)
6
7 print(f"Lista original: {numeros}")
8 print(f"Lista invertida: {numeros_invertidos}")

```

Código 4.24: Solución al Ejercicio 4.24.

4. Instrucciones simples y estructuradas

EJERCICIO 4.25

Escribe un programa que calcule la suma de todos los números impares entre 1 y un número dado por el usuario, mayor que 1, utilizando un bucle `for`.

Ejemplo de ejecución esperada

Introduce un número: 100

La suma de los números impares entre 1 y 100 es 2500.

Solución (girar para leer)

```
1 numero = int(input("Introduce un número: "))
2 suma_impares = 0
3 for i in range(1, numero + 1, 2):
4     suma_impares += i
5
6 print(f"La suma de los números impares entre 1 y {numero} es {suma_impares}.")
```

Código 4.25: Solución al Ejercicio 4.25.

EJERCICIO 4.26

Escribe un programa que cuente cuántas veces aparece un carácter específico en una cadena de texto utilizando un bucle `for`. Solicita al usuario tanto la cadena de texto como el carácter a contar.

Ejemplo de ejecución esperada

Introduce una cadena de texto: programar

Introduce el carácter a contar: r

El carácter 'r' aparece 3 veces en la cadena.

Solución (girar para leer)

```

1 cadena = input("Introduce una cadena de texto: ")
2 caracter = input("Introduce el carácter a contar: ")
3
4 contador = 0
5 for letra in cadena:
6     if letra == caracter:
7         contador += 1
8
9 print(f"El carácter '{caracter}' aparece {contador}
← veces en la cadena.")

```

Código 4.26: Solución al Ejercicio 4.26.

4. Instrucciones simples y estructuradas

EJERCICIO 4.27

Escribe un programa que calcule el promedio de los números contenidos en una lista de 10 números utilizando un bucle `for`. Considera la siguiente lista de números: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

Ejemplo de ejecución esperada

El promedio es 55.0

Solución (girar para leer)

```
1 numeros = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
2 suma = 0
3 cantidad = 0
4
5 for numero in numeros:
6     suma += numero
7     cantidad += 1
8
9 promedio = suma / cantidad
10 print(f"El promedio es {promedio}.")
```

Código 4.27: Solución al Ejercicio 4.27.

EJERCICIO 4.28

Escribe un programa que cuente cuántas vocales (mayúsculas o minúsculas) contiene una cadena de texto ingresada por el usuario. El programa debe usar un bucle for para iterar sobre cada carácter de la cadena.

Ejemplo de ejecución esperada

Introduce una cadena de texto: Hola Mundo
El número de vocales en la cadena es 4.

Solución (girar para leer)

```

1 cadena = input("Introduce una cadena de texto: ").lower()
2 vocales = "aeiou"
3 contador_vocales = 0
4
5 for letra in cadena:
6     if letra in vocales:
7         contador_vocales += 1
8
9 print(f"El número de vocales en la cadena es {
10     contador_vocales}")

```

Código 4.28: Solución al Ejercicio 4.28.

4. Instrucciones simples y estructuradas

EJERCICIO 4.29

Escribe un programa que encuentre todos los números primos entre 1 y 100, utilizando un bucle `for`. Un número primo es aquel que sólo es divisible por 1 y por sí mismo. El programa debe imprimir los números primos encontrados en este rango.

Condiciones:

- Si se encuentra un número que no es primo, se debe usar `continue` para pasar al siguiente número sin hacer más cálculos.
- Si se encuentra un número primo, se debe imprimir en pantalla.
- El programa debe usar un bucle anidado, donde el bucle exterior recorre los números del 1 al 100, y el bucle interior verifica si cada número es primo.

Ejemplo de ejecución esperada

Números primos entre 1 y 100: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Solución (girar para leer)

```
1 for num in range(2, 101): # Recorre los números del 2
2     es-primo = True # Inicializamos la suposición de
3     for i in range(2, num): # Verificamos si el número
4         es divisible por algún número menor que el
5         if num % i == 0: # Si es divisible, no es
6             es-primo = False
7             break # No necesitamos seguir verificando,
8             salimos del bucle interior
9     if es-primo:
10         print(num, end=" ")
```

Código 4.29: Solución al Ejercicio 4.29.

EJERCICIO 4.30

Escribe un programa que genere una secuencia de números que cumpla las siguientes condiciones:

- La secuencia comienza en 1 y termina en 100.
- La secuencia incluye solo los números que:
 - Son múltiplos de 3, pero no de 5.
 - Son múltiplos de 5, pero no de 3.
 - Si el número es múltiplo tanto de 3 como de 5, se omite.
- El programa debe imprimir la secuencia resultante.

Condiciones:

- Si el número es múltiplo de 3 y 5, usar continue para saltar al siguiente número sin agregarlo a la secuencia.
- Si el número es múltiplo de 3 pero no de 5, agregarlo a la secuencia.
- Si el número es múltiplo de 5 pero no de 3, agregarlo también.

Ejemplo de ejecución esperada

```
3 5 6 9 10 12 18 20 21 24 25 27 33 35 36 39 40 42 48 50 51 54 55 57
63 65 66 69 70 72 78 80 81 84 85 87 93 95 96 99 100
```

Solución (girar para leer)

```
1 for num in range(1, 101):
2     if num % 3 == 0 and num % 5 == 0:
3         continue
4     if num % 3 == 0 or num % 5 == 0:
5         print(num, end=" ")
6     else:
7         print(num, end=" ")
8     print()
```

Código 4.30: Solución al Ejercicio 4.30.

5

Diseño de programas elementales en Python

En este capítulo exploraremos la estructura y funcionamiento de programas básicos en Python, prestando especial atención a la modularidad y la comunicación de datos entre funciones. Para ello, vamos a diseñar e implementar un programa que escriba la tabla de multiplicar de un número dado por el usuario.

El programa permitirá al usuario solicitar la tabla de multiplicar de un número. El programa continuará funcionando hasta que el usuario introduzca el número 0, momento en el que finalizará su ejecución.

Un ejemplo de ejecución de lo que se desea es:

```
¿Qué tabla desea escribir? (0 para acabar): 7
LA TABLA DEL 7
7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
...
7 x 10 = 70
```

```
¿Qué tabla desea escribir? (0 para acabar): 0
```

5.1. Funciones

Podemos resolver este problema directamente dentro de un bloque principal. El flujo del programa sería: (1) pedirle al usuario un número, (2) generar y mostrar la tabla de multiplicar de ese número, y (3) repetir este proceso hasta que el usuario ingrese un 0 para finalizar.

Por ejemplo:

```
1 # Solución sin funciones
2
3 while True:
4     numero = int(input("¿Qué tabla desea escribir? (0 para
↪     acabar): "))
5     if numero == 0:
6         break
7
8     print(f"\nLA TABLA DEL {numero}")
9     for i in range(11):
10        print(f"{numero} x {i} = {numero * i}")
```

En este caso, el programa hace todo en un único bloque. Primero, el programa solicita el número de la tabla (línea 4) y luego, en un bucle, genera las multiplicaciones desde 0 hasta 10 (líneas 9 y 10). Si el usuario ingresa 0, el programa termina (líneas 5 y 6).

Esta solución plantea realizar todo en un único bloque de código, sin necesidad de definir *funciones*. Una función es un conjunto de instrucciones que realizan una tarea específica. Cuando definimos una función, no se ejecuta inmediatamente; se ejecutará cuando la llamemos. Llamar a una función significa invocar o ejecutar el bloque de código que define esa función. Es decir, cuando “llamamos” a una función, el programa va a la parte de código donde está definida y ejecuta las instrucciones que contiene.

La solución sin funciones es rápida y sencilla, especialmente en programas pequeños o cuando no se anticipa que el código se reutilice en otros contextos. Sin embargo, a medida que el programa crece o se vuelve más complejo, este enfoque puede llevar a un código repetitivo y menos organizado. La falta de modularidad puede hacer que el código sea más difícil de leer y mantener, especialmente si necesitamos modificar el comportamiento del programa o añadir nuevas características. Además, la estructura del programa se vuelve más difícil de gestionar si se requiere hacer cambios en la lógica principal.

Ahora, vamos a reestructurar el programa utilizando funciones. Como se ha mencionado, esto no sólo organiza el código, sino que facilita la reutilización y mejora la legibilidad. La solución sería algo como esto:

```
1 # Solución con funciones
2
3 def presentar_tabla(n):
4     """Muestra por pantalla la tabla de multiplicar del nú
↪     mero n."""
5     print(f"\nLA TABLA DEL {n}")
6     for i in range(11):
7         print(f"{n} x {i} = {n * i}")
8
9 def main():
10    """Gestiona el diálogo con el usuario y llama a
```

```

11     while True:
12         numero = int(input("¿Qué tabla desea escribir? (0
13         para acabar): "))
14         if numero == 0:
15             break
16         presentar_tabla(numero)
17 if __name__ == "__main__":
18     main()

```

En esta versión del programa, hemos dividido el código en dos funciones:

- La función `presentar_tabla(n)` se encarga de imprimir la tabla de multiplicar para el número `n`.
- La función `main()` gestiona la interacción con el usuario y llama a `presentar_tabla()` para mostrar la tabla cuando sea necesario.

El enfoque con funciones es más estructurado y modular. Al tener funciones separadas para tareas específicas (como `presentar_tabla()`), el código es más limpio y fácil de entender. Además, la reutilización del código se vuelve mucho más fácil, porque cada función puede ser invocada en cualquier parte del programa sin duplicar la lógica. En este caso, por ejemplo, si quisiéramos agregar nuevas funcionalidades (como imprimir una tabla en un archivo o mostrar más detalles), sólo necesitaríamos hacer cambios en una función sin tocar la parte del código que maneja la interacción con el usuario.

Es conveniente descomponer nuestros programas en funciones siempre que sea posible. La principal desventaja de usar funciones en programas muy pequeños es que podría parecer innecesario dividir el código en funciones separadas cuando la lógica es tan simple. Esto podría hacer que el código sea un poco más largo de lo necesario, pero este “coste” se justifica rápidamente cuando el programa crece o se vuelve más complejo.

Respecto a la solución con funciones planteada, conviene explicar la estructura:

```

if __name__ == "__main__":
    main()

```

que aparece al final de la solución planteada. Este bloque es una forma común en Python de asegurarse de que un determinado bloque de código sólo se ejecute cuando el script se ejecuta directamente y no cuando es importado como módulo en otro programa.

En Python, `__name__` es una variable especial que indica el contexto en el que un script se está ejecutando. Cuando un script es ejecutado directamente (por ejemplo, con `python3 script.py` desde una terminal), Python asigna el valor `"__main__"` a la variable `__name__`. Por contra, cuando un script es importado como un módulo en otro programa, `__name__` tomará el nombre del módulo y no será `"__main__"`.

5.1.1. Definición de funciones

La definición de una función en Python comienza con la palabra clave `def`, seguida del nombre de la función, los paréntesis que pueden incluir parámetros (si los tiene), y dos puntos `:`. Los parámetros son valores que una función puede recibir cuando es llamada. Estos valores permiten que la función trabaje con datos específicos en lugar de realizar solo una tarea generalizada. Dentro de la función, el código se indenta para indicar que las instrucciones forman parte de la función.

La sintaxis básica para definir una función es:

```
def nombre_de_la_funcion(parametros):  
    # Código de la función  
    # Realiza alguna tarea
```

Veamos un ejemplo básico de definición de función muy simple que no toma parámetros y no devuelve nada:

```
1 def saludar():  
2     print(";Hola, mundo!")
```

5.1.2. Parámetros en la función

Los parámetros permiten que una función sea flexible y reutilizable con diferentes entradas sin tener que escribir el mismo código una y otra vez. Los parámetros son valores que la función puede recibir para realizar su tarea. Se definen dentro de los paréntesis de la función. Los parámetros son como “variables” dentro de la función, que se usan para realizar los cálculos o tareas.

Veamos un ejemplo básico de definición de función muy simple con parámetros:

```
1 def saludar_persona(nombre):  
2     print(f";Hola, {nombre}!")
```

En este caso, la función `saludar_persona` toma un parámetro llamado `nombre`. Ahora, cuando llamamos a la función, le pasamos un valor para ese parámetro:

```
saludar_persona("Carlos") # Imprime: ;Hola, Carlos!
```

Parámetros con valores por defecto

En Python, es posible asignar valores por defecto a los parámetros de una función. Esto significa que si no se proporciona un valor para ese parámetro al llamar a la función, se utilizará el valor predeterminado. Los valores por defecto permiten hacer que los parámetros sean opcionales, facilitando la reutilización de las funciones.

La sintaxis para definir parámetros con valores por defecto es la siguiente:

```
1 def saludar(nombre="Usuario"):  
2     print(f"Hola, {nombre}!")  
3
```

```

4 # Llamar a la función sin pasar un argumento para 'nombre'
5 saludar() # Salida: Hola, Usuario!
6
7 # Llamar a la función pasando un argumento
8 saludar("Carlos") # Salida: Hola, Carlos!

```

En este ejemplo, el parámetro `nombre` tiene un valor por defecto de `"Usuario"`. Si no se pasa un valor para `nombre`, se utiliza este valor predeterminado. Si se pasa un valor, ese será el que se use en lugar del predeterminado.

Los parámetros con valores por defecto deben ir al final de la lista de parámetros. Si una función tiene parámetros con valores por defecto, estos deben colocarse después de los parámetros sin valor por defecto. De lo contrario, Python generará un error de sintaxis (`SyntaxError: non-default argument follows default argument`):

```

# Correcto
def saludar(nombre, mensaje="¡Hola!"):
    print(f"{mensaje}, {nombre}")

# Incorrecto: SyntaxError en tiempo de compilación
def saludar(mensaje="¡Hola!", nombre):
    print(f"{mensaje}, {nombre}")

```

Si una función tiene tanto parámetros con valores por defecto como parámetros sin valores por defecto, los parámetros sin valores por defecto deben ser pasados al llamar a la función, y los valores por defecto solo se aplicarán a los parámetros que no se hayan pasado:

```

def saludar(nombre, mensaje="¡Hola!"):
    print(f"{mensaje}, {nombre}")

# Llamadas correctas
saludar("Carlos") # Salida: ¡Hola!, Carlos
saludar("Carlos", "Buenos días") # Salida: Buenos días,
→ Carlos

```

Este tipo de flexibilidad hace que las funciones sean más fáciles de manejar, especialmente cuando tienen muchos parámetros, pero algunos de ellos tienen valores predeterminados que sólo cambian en ciertos casos.

5.1.3. Llamar a una función

Cuando llamamos o invocamos una función, lo que hacemos es ejecutar el código que hemos definido dentro de ella. Para ello, simplemente escribimos su nombre seguido de paréntesis. Si la función tiene parámetros, los valores deben ser proporcionados dentro de los paréntesis, separados por comas.

En el caso de que la función tenga varios parámetros, los valores deben ser pasados en el mismo orden en que los definimos en la declaración de la función. Es decir, el primer

5. Diseño de programas elementales en Python

valor que pasamos corresponde al primer parámetro, el segundo valor corresponde al segundo parámetro, y así sucesivamente. Los valores deben estar separados por comas.

Por ejemplo, si tienes una función que imprime dos números y la defines de esta manera:

```
1 def imprime(a, b):  
2     print("a:", a, "; b:", b)
```

Para llamarla y pasarle los valores que mostrará, deberás proporcionar dos números en el orden correspondiente:

```
imprime(5, 3)
```

En este caso, el valor 5 se pasa al parámetro `a`, y el valor 3 se pasa al parámetro `b`. Cuando la función se ejecuta, se imprimirá por pantalla el valor 5 de `a` y el valor 3 de `b`.

Es importante recordar que el número de parámetros debe coincidir con el número de argumentos que pasamos en la llamada. Si intentamos llamar a la función con un número incorrecto de argumentos, Python generará un error.

Por ejemplo, si intentas pasar solamente un argumento a la función `imprime`, obtendrás un error durante la ejecución del programa:

```
↪ # Error, la función espera dos parametros  
imprime(5) # TypeError: missing 1 required positional  
           argument: 'b'
```

Python permite usar parámetros con nombre (también conocidos como argumentos con palabra clave). En lugar de pasar los valores en el orden en que fueron definidos, puedes especificar qué parámetro está recibiendo cada valor. Esto se hace utilizando el nombre del parámetro seguido del operador de asignación (`=`) y el valor que deseas pasar (no lo confundas con que el parámetro tenga un valor por defecto: aquí el `=` aparece en la llamada, no en la definición).

Por ejemplo:

```
1 def saludar(nombre, edad):  
2     print(f"Hola, {nombre}. Tienes {edad} años.")
```

En este caso, al invocar la función, puedes pasar los parámetros por nombre:

```
saludar(edad=30, nombre="Carlos")
```

Aquí, aunque `nombre` se define primero en la función, estamos pasando el valor 30 a `edad` y el valor `"Carlos"` a `nombre`. Esto puede ser útil para hacer el código más legible y para asegurarte de que los valores son asignados correctamente a los parámetros, especialmente cuando tienes funciones con varios parámetros. Además, permite que no sea necesario seguir estrictamente el orden de los parámetros, lo que mejora la flexibilidad.

5.1.4. La sentencia `return`

Una función puede devolver un valor usando la sentencia `return`. Cuando una función tiene `return`, el valor especificado es enviado de vuelta a quien llamó a la función. Una

función que no tiene `return` devolverá `None` de forma predeterminada.

Veamos un ejemplo de función con `return`:

```
1 def sumar(a, b):
2     return a + b
```

Aquí, la función `sumar` toma dos parámetros, `a` y `b`, y devuelve la suma de ambos. Cuando llamamos a esta función, podemos “capturar” el valor devuelto:

```
resultado = sumar(5, 3) # Llamada a la función
print(resultado) # Imprime 8
```

Cuando la función se ejecuta, el resultado de evaluar $5 + 3$ (código de la función `sumar` con los parámetros recibidos) se almacena en la variable `resultado`, con lo que mostrará por pantalla el valor 8.

La sentencia `return` puede usarse en cualquier parte de la función. Si se encuentra un `return`, la ejecución de la función se detiene inmediatamente, y el valor se devuelve a la parte que hizo la llamada.

Por ejemplo:

```
1 def mayor_de_dos(a, b):
2     if a > b:
3         return a # Si 'a' es mayor, devolvemos 'a'
4     else:
5         return b # Si no, devolvemos 'b'
6
7 # Llamada a la función
8 print(mayor_de_dos(5, 10)) # Imprime 10
```

Aquí, la función devuelve el valor mayor de los dos números comparados. Si `a > b`, se devuelve `a`; si no, se devuelve `b`. La ejecución de la función se detiene en el primer `return` que se encuentre.

Si una función no tiene una sentencia `return`, entonces no devuelve un valor explícito. En este caso, el valor por defecto que Python retorna es `None`.

Por ejemplo:

```
1 def saludar_persona(nombre):
2     print(f";Hola, {nombre}!") # Imprime el saludo
3
4 # Llamamos a la función
5 saludar_persona("Carlos") # Imprime: ;Hola, Carlos!
6
7 # Esta llamada no tiene valor de retorno, así que:
8 saludo = saludar_persona("Carlos")
9 print(saludo) # Imprime: None
```

En este ejemplo, la función `saludar_persona` no tiene un `return`, por lo que cuando intentamos capturar su salida en la variable `saludo`, esta obtiene el valor `None`.

Retorno de múltiples valores

En Python, una función puede devolver más de un valor separándolos por comas. En realidad, lo que ocurre es que Python agrupa estos valores en una tupla¹. Por ejemplo:

```
1 def operaciones_basicas(a, b):
2     suma = a + b
3     resta = a - b
4     multiplicacion = a * b
5     return suma, resta, multiplicacion # Se devuelven tres
↪     valores
6
7 # Captura de múltiples valores en variables separadas
8 resultado_suma, resultado_resta, resultado_multiplicacion =
↪     operaciones_basicas(10, 5)
9
10 print(f"Suma: {resultado_suma}, Resta: {resultado_resta},
↪     Multiplicación: {resultado_multiplicacion}")
```

Esto facilita la programación, ya que permite retornar varios resultados sin necesidad de encapsularlos en una lista o diccionario (tipos de datos estructurados estudiados en un capítulo posterior). También se pueden capturar todos los valores en una única variable si no se conoce de antemano cuántos valores retornará la función:

```
1 resultado = operaciones_basicas(10, 5)
2 print(resultado) # Salida: (15, 5, 50)
```

Así, resultado contendrá una tupla con los tres valores, que luego pueden accederse mediante resultado[0], resultado[1], etcétera.

5.1.5. Documentar funciones: docstrings

Un *docstring* (cadena de documentación) es una cadena literal que aparece como **primera sentencia** de un módulo, función, clase o método. Aunque sintácticamente es una cadena cualquiera, Python la asocia automáticamente al objeto en el atributo especial `__doc__`, y herramientas como `help()` la utilizan para mostrar la documentación al usuario. Por ejemplo:

```
1 def calcular_area(radio):
2     """Calcula el área de un círculo dado su radio."""
3     return 3.1416 * radio * radio
4
5 print(calcular_area.__doc__)
6 # Salida: Calcula el área de un círculo dado su radio.
7
```

¹Este tipo de dato estructurado se estudia en detalle más adelante.

```

8 help(calcular_area)
9 # Salida:
10 # calcular_area(radio)
11 #     Calcula el área de un círculo dado su radio.

```

Para funciones simples basta con una línea entre comillas triples. Para funciones más complejas, es habitual seguir alguna convención de estilo: las más populares son *Google style*, *NumPy style* y *Sphinx/reST style*. A modo de ejemplo, una versión más completa al estilo Google sería:

```

1 def calcular_area(radio):
2     """Calcula el área de un círculo.
3
4     Args:
5         radio (float): radio del círculo, en metros.
6
7     Returns:
8         float: área del círculo, en metros cuadrados.
9     """
10    return 3.1416 * radio * radio

```

Adoptar siempre la misma convención dentro de un proyecto es importante para mantener la coherencia, y permite que generadores de documentación como *Sphinx* o *pdoc* produzcan automáticamente referencias a partir del código fuente.

5.1.6. Anotaciones de tipo

En Python, podemos indicar los tipos de datos de los parámetros de una función utilizando las anotaciones de tipo (del inglés, *type hints*). Esto significa que podemos especificar qué tipo de datos se espera que reciba cada parámetro de la función. Por ejemplo, si queremos indicar que una función debería recibir un número entero como parámetro, podemos anotarlo de la siguiente manera:

```

1 def sumar(a: int, b: int) -> int:
2     return a + b

```

En este ejemplo, las anotaciones `a: int` y `b: int` indican que esperamos que ambos parámetros sean de tipo entero (`int`), y la anotación `-> int` después de los paréntesis de la función indica que esperamos que la función devuelva un valor de tipo entero.

Aunque esta anotación de tipo proporciona claridad sobre lo que esperamos como entrada y salida de la función, es importante recordar que Python es un lenguaje de tipado dinámico. Esto significa que el intérprete de Python **no** valida en tiempo de ejecución que los tipos coincidan con los anotados: las anotaciones son simplemente metadatos asociados al objeto función.

Por ejemplo, en el caso de la función `sumar`, podríamos llamarla con parámetros de tipo incorrecto y el intérprete no rechazaría la llamada:

```
↪ sumar(5, "10") # Python no verifica las anotaciones; aceptar  
    ía la llamada
```

Aunque Python no genera error inmediatamente, esta función intentará sumar un entero (5) y una cadena ("10"), lo que resultará en un error en tiempo de ejecución:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Si se desea verificar las anotaciones *antes* de ejecutar el programa, existen herramientas externas de análisis estático como `mypy` (<https://mypy-lang.org/>) o `pyright` (<https://github.com/microsoft/pyright>), que examinan el código fuente y avisan de inconsistencias entre las anotaciones y los usos reales. Estas herramientas son la forma habitual de aprovechar las anotaciones de tipo en proyectos grandes.

5.1.7. Ámbito de las variables

En Python, cada variable tiene un *ámbito* (también llamado *alcance* o *scope*), que determina dónde puede ser accedida en el código, y un *tiempo de vida*, que define cuánto tiempo permanece disponible en memoria durante la ejecución. Existen dos ámbitos básicos: variables *locales* y variables *globales*.

Una variable local es aquella definida dentro de una función y solo es accesible dentro de ella. Por ejemplo:

```
1 def ejemplo_local():  
2     x = 10 # Variable local  
3     print(x) # Accesible dentro de la función  
4  
5 ejemplo_local()  
6 # print(x) # Generaría un error: 'x' no existe fuera de la  
↪ función
```

Las variables globales son aquellas definidas fuera de cualquier función y pueden leerse desde cualquier parte del programa:

```
1 mensaje = "Hola, mundo" # Variable global  
2  
3 def mostrar_mensaje():  
4     print(mensaje) # Lee la variable global  
5  
6 mostrar_mensaje()
```

Sin embargo, si dentro de una función se asigna un valor a una variable global sin declararla explícitamente como `global`, Python creará una variable local en su lugar. Para modificar la variable global dentro de la función, hay que declararla con la palabra clave `global`:

```
1 contador = 0 # Variable global  
2
```

```

3 def incrementar():
4     global contador # Se declara explícitamente como global
5     contador += 1
6
7 incrementar()
8 print(contador) # Muestra 1

```

Regla LEGB. Cuando una función usa un nombre, Python lo busca siguiendo cuatro ámbitos en orden: **L**ocal (variables definidas dentro de la propia función), **E**nclosing (variables de funciones que la engloban si las hay), **G**lobal (variables del módulo) y **B**uilt-in (nombres del lenguaje, como `print` o `len`). El primer ámbito en el que se encuentra el nombre es el que se usa.

Para modificar una variable del ámbito *enclosing* desde una función anidada se utiliza la palabra clave `nonlocal`, análoga a `global` pero referida al ámbito de la función contenedora:

```

1 def contador_externo():
2     n = 0
3     def incrementar():
4         nonlocal n # Se refiere a la 'n' de contador_externo
5         n += 1
6         return n
7     return incrementar

```

Las variables globales son una mala práctica. Compartir información entre funciones a través de variables globales debe evitarse: genera efectos secundarios indeseados, dificulta la depuración y reduce la modularidad y la reutilización del código. La forma recomendada de comunicar funciones es a través de *parámetros* (datos de entrada) y *valores de retorno* (datos de salida), tal como se explica en la siguiente sección. Este enfoque permite que cada función sea independiente, facilita su prueba, su reutilización y su mantenimiento, y elimina el riesgo de que modificaciones accidentales en un sitio afecten al resto del programa.

5.2. Comunicación entre funciones

La comunicación entre funciones se refiere al proceso mediante el cual una función interactúa con otras dentro de un programa. Esto se puede lograr de diferentes maneras, y el método más común es pasar datos entre funciones. La comunicación puede hacerse mediante el uso de parámetros y valores de retorno.

Cuando una función necesita usar información proporcionada por otra, generalmente se hace mediante los parámetros. Los parámetros actúan como entradas de la función, y permiten que esta realice operaciones o cálculos utilizando datos específicos. Estos

parámetros se definen cuando la función es creada, y se rellenan cuando se invoca la función.

Por ejemplo, imagina que tienes una función para calcular el área de un círculo. La función necesitaría el radio del círculo como parámetro para poder realizar el cálculo:

```
1 def calcular_area(radio):
2     return 3.1416 * radio * radio
```

Aquí, `radio` es un parámetro que se pasa a la función cuando se llama. El valor de `radio` es utilizado dentro de la función para calcular el área, y el resultado se devuelve con la sentencia `return`.

Una vez realizado el cálculo, la función puede devolver un valor mediante la sentencia `return`. Este valor puede ser utilizado por otra parte del programa, ya sea para realizar más cálculos o para mostrar el resultado al usuario. Cuando se llama a una función, el programa va a esa función, ejecuta su código y luego vuelve al lugar donde la función fue invocada, trayendo consigo el valor que la función ha retornado.

Por ejemplo:

```
1 def calcular_area(radio):
2     return 3.1416 * radio * radio
3
4 area = calcular_area(5)
5 print(area) # Imprime el área calculada
```

En este caso, la función `calcular_area` recibe el parámetro 5 como el valor de `radio`, realiza el cálculo, y devuelve el resultado. Este valor se captura en la variable `area` en el programa principal, y luego se imprime.

Paso de parámetros: paso por asignación. A diferencia de lenguajes como C (donde los parámetros se pasan por valor) o C++ (donde se pueden pasar por valor o por referencia), **en Python los parámetros se pasan por asignación** (también llamado *call by sharing* o *pass-by-object-reference*). Esto significa que cuando se llama a una función, el parámetro de la función queda *ligado al mismo objeto* al que apunta el argumento de la llamada: no se copia el objeto ni se crea un alias nuevo distinto.

La consecuencia práctica depende de si el objeto es *mutable* (listas, diccionarios, conjuntos, objetos definidos por el usuario) o *immutable* (enteros, flotantes, cadenas, tuplas, booleanos):

- **Si el objeto es immutable**, no puede modificarse en sitio. Cualquier asignación dentro de la función crea un nuevo objeto y reasigna el nombre local, pero el objeto original no cambia.
- **Si el objeto es mutable**, los cambios que se hagan a través de los métodos del objeto (por ejemplo, `append()` en una lista) *sí* se ven desde fuera, porque el objeto al que apuntan ambos nombres es el mismo.

Por ejemplo, una función que modifica una lista:

```

1 def modificar_lista(lista):
2     """Añade un nuevo elemento a la lista recibida."""
3     lista.append(100)
4
5 mi_lista = [1, 2, 3]
6 modificar_lista(mi_lista)
7 print(mi_lista)  # [1, 2, 3, 100]

```

Para evitar que una función modifique accidentalmente una estructura mutable que recibe, se puede trabajar con una copia y devolverla:

```

1 def modificar_lista_segura(lista):
2     """Devuelve una copia con un elemento adicional, sin
3     tocar la original."""
4     copia = lista.copy()
5     copia.append(100)
6     return copia
7
8 mi_lista = [1, 2, 3]
9 nueva_lista = modificar_lista_segura(mi_lista)
10 print(mi_lista)      # [1, 2, 3] (la original no cambia)
11 print(nueva_lista)   # [1, 2, 3, 100]

```

5.3. Ejercicios

EJERCICIO 5.1

Escribe un programa en Python que convierta temperaturas de grados Celsius a grados Fahrenheit. El programa debe solicitar al usuario la temperatura en Celsius y mostrar el valor equivalente en Fahrenheit (con un único decimal). Recuerda que la fórmula para calcular los grados Fahrenheit a partir de grados Celsius es

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

Ejemplo de ejecución esperada

```

Introduce la temperatura en Celsius: 25
La temperatura en Fahrenheit es: 77.0

```

Solución (girar para leer)

```
1 def convertir_a_fahrenheit(celsius):  
2     return (celsius * 9/5) + 32  
3  
4 def main():  
5     celsius = float(input("Introduce la temperatura en  
6     Celcius: "))  
7     fahrenheit = convertir_a_fahrenheit(celsius)  
8     print(f"La temperatura en Fahrenheit es: {  
9     fahrenheit:.1f}")  
10 if __name__ == "__main__":  
11     main()
```

Código 5.1: Solución al Ejercicio 5.1.

EJERCICIO 5.2

Escribe un programa en Python que reciba un número entero y determine si es par o impar. El programa debe mostrar un mensaje indicando si el número es par o impar.

Ejemplo de ejecución esperada

Introduce un número: 7
El número 7 es impar.

Solución (girar para leer)

```

13 def main():
12     if __name__ == "__main__":
11
10         print(f"El número {numero} es {resultado}")
9         resultado = es-par-o-impar(numero)
8         numero = int(input("Introduce un número: "))
7     def main():
6
5         return "impar"
4         else:
3             return "par"
2         if numero % 2 == 0:
1             es-par-o-impar(numero):

```

Código 5.2: Solución al Ejercicio 5.2.

EJERCICIO 5.3

Escribe un programa en Python que calcule el área de un rectángulo. El programa debe solicitar al usuario la base y la altura del rectángulo y mostrar el área (con dos decimales).

Ejemplo de ejecución esperada

```
Introduce la base del rectángulo: 5
Introduce la altura del rectángulo: 8
El área del rectángulo es: 40.00
```

Solución (girar para leer)

```
11 def calcular_area_rectangulo(base, altura):
10     return base * altura
9
8 def main():
7     base = float(input("Introduce la base del rectángulo: "))
6     altura = float(input("Introduce la altura del rectángulo: "))
5     area = calcular_area_rectangulo(base, altura)
4     print(f"El área del rectángulo es: {area:.2f}")
3
2 if __name__ == "__main__":
1     main()
```

Código 5.3: Solución al Ejercicio 5.3.

EJERCICIO 5.4

Escribe un programa en Python que reciba tres números y determine cuál es el mayor de los tres. El programa debe solicitar al usuario los tres números y mostrar el número mayor. Incorpora un control de errores ante entradas incorrectas por parte del usuario.

Ejemplo de ejecución esperada

```
Introduce el primer número: 8
Introduce el segundo número: 12
Introduce el tercer número: 5
El número mayor es 12
```

Solución (girar para leer)

```

1  def numero_mayor(a, b, c):
2      return max(a, b, c)
3      # max es una función ya existente en Python,
4      # el máximo de los parámetros pasados
5      devuelve
6  def main():
7      try:
8          a = float(input("Introduce el primer número: "))
9          b = float(input("Introduce el segundo número: "))
10         c = float(input("Introduce el tercer número: "))
11         mayor = numero_mayor(a, b, c)
12         print(f"El número mayor es {mayor}")
13     except ValueError:
14         print("Por favor, ingresa un número válido.")
15
16 if __name__ == "__main__":
17     main()
```

Código 5.4: Solución al Ejercicio 5.4.

EJERCICIO 5.5

Escribe un programa en Python que determine si un número es primo. Un número es primo si sólo tiene dos divisores positivos: 1 y el mismo número. El programa debe solicitar al usuario un número y mostrar si es primo o no.

Ejemplo de ejecución esperada

Introduce un número: 17

El número 17 es primo.

Solución (girar para leer)

```
1 def es_primo(numero):
2     if numero < 2:
3         return False
4     for i in range(2, int(numero ** 0.5) + 1):
5         if numero % i == 0:
6             return False
7     return True
8
9 def main():
10     numero = int(input("Introduce un número: "))
11     if es_primo(numero):
12         print(f"El número {numero} es primo.")
13     else:
14         print(f"El número {numero} no es primo.")
15
16 if __name__ == "__main__":
17     main()
```

Código 5.5: Solución al Ejercicio 5.5.

EJERCICIO 5.6

Escribe un programa en Python que cuente cuántos dígitos tiene un número entero. El programa debe solicitar al usuario un número y mostrar la cantidad de dígitos.

Ejemplo de ejecución esperada

Introduce un número: 5421
El número tiene 4 dígitos.

Solución (girar para leer)

```

1  def contar_dígitos(numero):
2      return len(str(abs(numero)))
3      # lo transformamos a cadena y miramos su longitud
4
5  def main():
6      numero = input("Introduce un número: ")
7      cantidad_dígitos = contar_dígitos(numero)
8      print(f"El número tiene {cantidad_dígitos} dígitos.")
9
10 if __name__ == "__main__":
11     main()

```

Código 5.6: Solución al Ejercicio 5.6.

EJERCICIO 5.7

Escribe un programa en Python que genere un número aleatorio entre 1 y 100, y el usuario debe adivinarlo. El programa debe indicar si el número ingresado por el usuario es mayor o menor que el número aleatorio.

Ejemplo de ejecución esperada

```
Adivina el número (entre 1 y 100): 50
El número es mayor.
Adivina el número: 75
El número es menor.
Adivina el número: 62
¡Correcto! El número era 62.
```

Solución (girar para leer)

```
1 import random
2
3 def adivina_el_numero():
4     numero_aleatorio = random.randint(1, 100)
5     intento = None
6     while intento != numero_aleatorio:
7         intento = int(input("Adivina el número (entre 1
8         y 100): "))
9         if intento > numero_aleatorio:
10            print("El número es mayor.")
11            elif intento < numero_aleatorio:
12                print("El número es menor.")
13            print(f"¡Correcto! El número era {numero_aleatorio}
14            ").
15 def main():
16     adivina_el_numero()
17
18 if __name__ == "__main__":
19     main()
```

Código 5.7: Solución al Ejercicio 5.7.

EJERCICIO 5.8

Escribe un programa en Python que calcule el interés compuesto de una inversión a lo largo del tiempo. La fórmula es:

$$A = P \left(1 + \frac{r}{n} \right)^{nt}$$

Donde P es el capital inicial, r es la tasa de interés anual (en decimal), n es el número de veces que se aplica el interés por año, y t es el número de años.

El programa debe solicitar al usuario estos valores y mostrar el monto final de la inversión (con dos decimales). Recuerda que si el usuario introduce el porcentaje en formato entero (por ejemplo, 5 para un 5%), tendrás que dividirlo entre 100 antes de aplicar la fórmula.

Ejemplo de ejecución esperada

```
Introduce el capital inicial: 1000
Introduce la tasa de interés anual (en porcentaje): 5
Introduce el número de veces que se aplica el interés al año: 12
Introduce la cantidad de años: 10
El monto final después de 10 años será: 1647.01
```

Solución (girar para leer)

```

14 main()
13 if __name__ == "__main__":
12
11     {monto_final:.2f}"
10     print(f"El monto final después de {años} años será:
9     monto_final = calcular_interes_compuesto(principal,
8     años = int(input("Introduce la cantidad de años: "))
7     veces_por_año = int(input("Introduce el número de
6     tasa = float(input("Introduce la tasa de interés
5     inicial: "))
4     principal = float(input("Introduce el capital
3     def main():
2     return principal * (1 + tasa / veces_por_año) **
1     def calcular_interes_compuesto(principal, tasa,
    veces_por_año, años):
    veces_por_año * años)

```

Código 5.8: Solución al Ejercicio 5.8.

EJERCICIO 5.9

Escribe un programa en Python que determine si un número ingresado por el usuario es capicúa. Un número capicúa se lee igual de izquierda a derecha que de derecha a izquierda.

Ejemplo de ejecución esperada

Introduce un número: 12321

El número 12321 es capicúa.

Solución (girar para leer)

```

1  def es_capicua(numero):
2      return str(numero) == str(numero)[::-1]
3
4  def main():
5      numero = int(input("Introduce un número: "))
6      if es_capicua(numero):
7          print(f"El número {numero} es capicúa.")
8      else:
9          print(f"El número {numero} no es capicúa.")
10
11  if __name__ == "__main__":
12      main()

```

Código 5.9: Solución al Ejercicio 5.9.

EJERCICIO 5.10

Escribe un programa en Python que calcule el Máximo Común Divisor (MCD) y el Mínimo Común Múltiplo (MCM) de dos números utilizando el algoritmo de Euclides.

Ejemplo de ejecución esperada

Introduce el primer número: 56
 Introduce el segundo número: 98
 El MCD de 56 y 98 es 14.
 El MCM de 56 y 98 es 392.

Solución (girar para leer)

```

1  def mcd(a, b):
2      while b != 0:
3          a, b = b, a % b
4      return a
5
6  def mcm(a, b, mcd):
7      return a * b // mcd
8
9  def main():
10     a = int(input("Introduce el primer número: "))
11     b = int(input("Introduce el segundo número: "))
12     resultado = mcd(a, b)
13     print(f"El MCD de {a} y {b} es {resultado}.")
14     resultado = mcm(a, b, resultado)
15     print(f"El MCM de {a} y {b} es {resultado}.")
16
17 if __name__ == "__main__":
18     main()

```

Código 5.10: Solución al Ejercicio 5.10.

EJERCICIO 5.11

Escribe un programa en Python que calcule la potencia de un número sin utilizar el operador **. El programa debe solicitar al usuario la base y el exponente, y calcular la base elevada al exponente mediante un bucle o recursión.

Ejemplo de ejecución esperada

```
Introduce la base: 3
Introduce el exponente: 4
El resultado de 3^4 es 81.
```

Solución (girar para leer)

```

1  def potencia(base, exponente):
2      resultado = 1
3      for _ in range(exponente):
4          # si la variable de iteración no se va a
5          # utilizar, puede
6          # usarse - en su lugar
7          resultado *= base
8      return resultado
9  def main():
10     base = int(input("Introduce la base: "))
11     exponente = int(input("Introduce el exponente: "))
12     resultado = potencia(base, exponente)
13     print(f"El resultado de {base}^{exponente} es {
14         resultado}")
15     if __name__ == "__main__":
16         main()
```

Código 5.11: Solución al Ejercicio 5.11.

EJERCICIO 5.12

Escribe un programa en Python que convierta un número entre 0 y 99 en su equivalente en texto. El programa debe manejar correctamente los números del 0 al 15 y las decenas (20, 30, etc.).

Ejemplo de ejecución esperada

Introduce un número entre 0 y 99: 42
El número en texto es: cuarenta y dos.

Solución (girar para leer)

```

1 def convertir_a_texto(numero):
2     # para resolver este ejercicio usamos listas, se
3     unidades = ["", "uno", "dos", "tres", "cuatro", "
4     cinco", "seis", "siete", "ocho", "nueve"]
5     decenas = ["", "diez", "veinte", "treinta", "
6     cuarenta", "cincuenta", "sesenta", "setenta", "
7     ochenta", "noventa"]
8     if numero < 10:
9         return unidades[numero]
10    elif numero > 30:
11        return f"veinti{unidades[numero-20]}"
12    else:
13        return f"{decenas[numero // 10]} y {unidades[
14        numero % 10]}"
15    def main():
16        numero = int(input("Introduce un número entre 0 y
17        99: "))
18        texto = convertir_a_texto(numero)
19        print(f"El número en texto es: {texto}.")
20    if __name__ == "__main__":
21        main()

```

Código 5.12: Solución al Ejercicio 5.12.

EJERCICIO 5.13

Escribe un programa en Python que determine si un número es perfecto. Un número perfecto es aquel cuya suma de sus divisores propios (sin incluir el propio número) es igual al número.

Ejemplo de ejecución esperada

Introduce un número: 28

El número 28 es perfecto porque $1 + 2 + 4 + 7 + 14 = 28$.

Solución (girar para leer)

```

1  def obtener_divisores(numero):
2      return [i for i in range(1, numero) if numero % i
3              == 0]
4  def es_numero_perfecto(numero):
5      return sum(obtener_divisores(numero)) == numero
6
7  def main():
8      numero = input("Introduce un número: ")
9      if es_numero_perfecto(numero):
10         divisores = obtener_divisores(numero)
11         suma = " + ".join(str(d) for d in divisores)
12         print(f"El número {numero} es perfecto porque {
13             suma} = {numero}.")
14     else:
15         print(f"El número {numero} no es perfecto.")
16     if __name__ == "__main__":
17         main()

```

Código 5.13: Solución al Ejercicio 5.13.

EJERCICIO 5.14

Escribe un programa en Python que reciba un número entero positivo y calcule la suma de sus dígitos. El programa debe solicitar al usuario un número y mostrar la suma de sus dígitos.

Ejemplo de ejecución esperada

Introduce un número: 5421

La suma de los dígitos es: 12

Solución (girar para leer)

```

1  def suma_digitos(numero):
2      suma = 0
3      while numero > 0:
4          suma += numero % 10
5          numero = numero // 10
6      return suma
7
8  def main():
9      numero = int(input("Introduce un número: "))
10     resultado = suma_digitos(numero)
11     print(f"La suma de los dígitos es: {resultado}")
12
13     if __name__ == "__main__":
14         main()

```

Código 5.14: Solución al Ejercicio 5.14.

EJERCICIO 5.15

Escribe un programa en Python que reciba dos matrices de 2x2 y calcule su suma. El programa debe solicitar al usuario los elementos de las dos matrices y mostrar la matriz resultante de la suma.

Ejemplo de ejecución esperada

Introduce los elementos de la primera matriz:

Introduce el elemento [1][1]: 1

Introduce el elemento [1][2]: 2

Introduce el elemento [2][1]: 3

Introduce el elemento [2][2]: 4

Introduce los elementos de la segunda matriz:

Introduce el elemento [1][1]: 5

Introduce el elemento [1][2]: 6

Introduce el elemento [2][1]: 7

Introduce el elemento [2][2]: 8

La suma de las matrices es:

6 8

10 12

Solución (girar para leer)

```

1  def suma_matrices(matriz1, matriz2):
2      matriz_resultante = []
3      for i in range(2):
4          fila = []
5          for j in range(2):
6              fila.append(matriz1[i][j] + matriz2[i][j])
7          matriz_resultante.append(fila)
8      return matriz_resultante
9
10 def main():
11     matriz1 = []
12     matriz2 = []
13
14     print("\nIntroduce los elementos de la primera matriz")
15     for i in range(2):
16         fila = []
17         for j in range(2):
18             fila.append(int(input(f"Introduce el
19 elemento [{i+1}][{j+1}]: ")))
20         matriz1.append(fila)
21
22     print("\nIntroduce los elementos de la segunda
23 matriz:")
24     for i in range(2):
25         fila.append(int(input(f"Introduce el
26 elemento [{i+1}][{j+1}]: ")))
27         matriz2.append(fila)
28
29     matriz_resultante = suma_matrices(matriz1, matriz2)
30     print("\nLa suma de las matrices es:")
31     for fila in matriz_resultante:
32         print(" ".join(map(str, fila)))
33
34 if __name__ == "__main__":
35     main()

```

Código 5.15: Solución al Ejercicio 5.15.

EJERCICIO 5.16

Escribe un programa en Python que genere la secuencia de Fibonacci hasta el n-ésimo término. El programa debe solicitar al usuario el valor de n y mostrar la secuencia de Fibonacci hasta ese término.

Ejemplo de ejecución esperada

Introduce un número: 6

La secuencia de Fibonacci hasta el término 6 es: 0 1 1 2 3 5

Solución (girar para leer)

```

1  def fibonacci(n):
2      a, b = 0, 1
3      secuencia = []
4      for _ in range(n):
5          secuencia.append(a)
6          a, b = b, a + b
7      return secuencia
8
9  def main():
10     n = int(input("Introduce un número: "))
11     secuencia = fibonacci(n)
12     print(f"La secuencia de Fibonacci hasta el término
    {n} es: {' '.join(map(str, secuencia))}")
13     # esto se podría haber hecho con un bucle for:
14     # for elemento in secuencia:
15         # print(elemento, end=' ')
16
17     if __name__ == "__main__":
18         main()

```

Código 5.16: Solución al Ejercicio 5.16.

EJERCICIO 5.17

Escribe un programa en Python que cuente cuántas vocales (mayúsculas y minúsculas) contiene una cadena de texto. El programa debe solicitar al usuario una cadena y mostrar la cantidad de vocales encontradas.

Ejemplo de ejecución esperada

Introduce una cadena: Hola Mundo
La cadena contiene 4 vocales.

Solución (girar para leer)

```

1  def contar_vocales(cadena):
2      contador = 0
3      for char in cadena:
4          if char in "aeiouAEIOU":
5              contador += 1
6      return contador
7
8  def main():
9      cadena = input("Introduce una cadena: ")
10     cantidad_vocales = contar_vocales(cadena)
11     print(f"La cadena contiene {cantidad_vocales} vocales.")
12
13     if __name__ == "__main__":
14         main()

```

Código 5.17: Solución al Ejercicio 5.17.

EJERCICIO 5.18

Escribe un programa en Python que reciba una cadena de texto e imprima la cadena invertida. El programa debe solicitar al usuario una cadena y mostrar la versión invertida de la misma.

Ejemplo de ejecución esperada

Introduce una cadena: Python

La cadena invertida es: nohtyP

Solución (girar para leer)

```

1  def invertir_cadena(cadena):
2      cadena_invertida = ""
3      for i in range(len(cadena) - 1, -1, -1):
4          cadena_invertida += cadena[i]
5      return cadena_invertida
6
7  def main():
8      cadena = input("Introduce una cadena: ")
9      cadena_invertida = invertir_cadena(cadena)
10     print(f"La cadena invertida es: {cadena_invertida}")
11
12 if __name__ == "__main__":
13     main()

```

Código 5.18: Solución al Ejercicio 5.18.

EJERCICIO 5.19

Escribe un programa en Python que convierta un número binario ingresado por el usuario a su equivalente en octal. El programa debe solicitar al usuario un número binario y mostrar el valor en octal.

Ejemplo de ejecución esperada

Introduce un número binario: 1011

El número binario 1011 en octal es: 13

Solución (girar para leer)

```

1  def binario_a_octal(binario):
2      decimal = 0
3      for i in range(len(binario)):
4          decimal += int(binario[i]) * (2 ** (len(binario)
5              - i - 1))
6          decimal > 0:
7              octal = str(decimal % 8) + octal
8              decimal = decimal // 8
9      return octal
10
11 def main():
12     binario = input("Introduce un número binario: ")
13     octal = binario_a_octal(binario)
14     print(f"El número binario {binario} en octal es: {
15         octal}")
16     if __name__ == "__main__":
17         main()

```

Código 5.19: Solución al Ejercicio 5.19.

EJERCICIO 5.20

Escribe un programa en Python que reciba una lista de números y calcule su promedio. El programa debe solicitar al usuario los números, los almacenará en una lista y luego calculará el promedio de los mismos.

Ejemplo de ejecución esperada

Introduce una lista de números separados por espacio: 3 2 1 4 55 12 42 8
El promedio de los números introducidos es: 15.88

Solución (girar para leer)

```

1 def calcular_promedio(lista):
2     suma = 0
3     for num in lista:
4         suma += num
5     promedio = suma / len(lista)
6     return promedio
7
8 def main():
9     numeros = input("Introduce una lista de números
10    separados por espacio: ").split()
11    # lee los números, separados por el espacio, guarda
12    # los en una lista
13    numeros = [float(num) for num in numeros]
14    # itera en cada elemento de la lista y los
15    # convierte a número real
16    # devuelve siempre una cadena
17    if len(numeros) == 0:
18        print("No se introdujeron números.")
19    else:
20        promedio = calcular_promedio(numeros)
21        print(f"El promedio de los números introducidos
22    es: {promedio:.2f}")
23
24 if __name__ == "__main__":
25     main()

```

Código 5.20: Solución al Ejercicio 5.20.

6

Desarrollo modular y diseño descendente

En este capítulo nos centraremos en el diseño de programas modulares en Python: aprenderemos a organizar el código en funciones reutilizables y módulos independientes, y a gestionar la interacción entre ellos. Este enfoque facilita la creación de aplicaciones más mantenibles, escalables y comprensibles.

La programación modular (o desarrollo modular) implica dividir un programa grande en componentes más pequeños, llamados *módulos*, que pueden ser desarrollados, probados y depurados de manera independiente. Cada módulo realiza una parte del trabajo del programa, y estos módulos interactúan entre sí para cumplir el objetivo global. En Python, los módulos son archivos `«.py»` que contienen funciones, clases o variables; los paquetes (que se introducirán más adelante) son directorios que agrupan varios módulos.

El desarrollo modular tiene múltiples ventajas: la *reutilización* del código (los módulos pueden reutilizarse en distintos programas o proyectos), la *facilidad de mantenimiento* (un fallo en un módulo puede corregirse sin afectar al resto del programa), el *trabajo en equipo* (módulos distintos pueden ser desarrollados en paralelo por distintas personas) y la *facilidad de pruebas* (cada módulo puede probarse por separado, asegurando su corrección antes de la integración).

El enfoque opuesto se denomina *programación monolítica*: el código se escribe en un solo bloque, sin una división clara en módulos independientes. Este enfoque puede ser útil para programas muy pequeños y rápidos de escribir, pero a medida que el código crece se vuelve difícil de gestionar y propenso a errores difíciles de localizar. En la práctica, salvo casos triviales, conviene adoptar siempre un enfoque modular.

6.1. Diseño descendente

El diseño descendente (también conocido como *top-down design*) es la técnica de diseño que mejor encaja con el desarrollo modular. Consiste en partir de una visión general del problema e ir descomponiéndolo progresivamente en partes más pequeñas y manejables. En programación, esto se traduce en definir primero la funcionalidad global del programa y, después, ir detallando los módulos o funciones necesarios para llevarla a cabo.

El proceso de diseño descendente consta de los siguientes pasos:

- 1.- Identificación de las necesidades:** comienza con una descripción general de lo que debe hacer el programa.
- 2.- Descomposición en partes más pequeñas:** divide las tareas globales en sub-tareas o funciones.
- 3.- Definición de módulos:** para cada sub-tarea, defines un módulo o función que resuelva un aspecto del problema global.
- 4.- Refinamiento continuo:** añade detalles de implementación gradualmente, a medida que profundizas en cada módulo.

Esta técnica favorece la claridad en la planificación (al partir de una visión general, se asegura que el programa cumpla los requisitos antes de entrar en detalles complejos), la organización (cada módulo tiene una tarea clara y bien definida) y la flexibilidad (al descomponer el problema en partes pequeñas, es más fácil modificar o mejorar una parte sin afectar al sistema completo).

En Python, el diseño descendente se implementa creando funciones y módulos que resuelvan cada parte del problema. Lo habitual es que un programa comience con una función principal, llamada por convención `main()`, que define el flujo general; a partir de ahí, se invocan funciones de niveles inferiores que se ocupan de tareas cada vez más específicas.

6.2. Uso de `import`: módulos y paquetes

Para que un programa se pueda partir en varios ficheros y aprovechar el código de otros, Python ofrece la sentencia `import`. La sentencia `import` se usa para *traer* código de otro archivo o módulo y poder utilizar las funciones, clases y variables definidas en él.

6.2.1. Importar un módulo completo

Si en un fichero llamado «`calculos.py`» tenemos definidas varias funciones, podemos importarlo desde otro fichero del mismo directorio con:

```
import calculos
```


Tras este `import`, todas las funciones del módulo están accesibles a través del nombre del módulo, seguido del operador punto y el nombre de la función:

```
calculos.es_primo(numero)
```

6.2.2. Importar nombres concretos

Si solo se necesitan algunas funciones de un módulo, se pueden importar de forma selectiva con la sintaxis `from ... import ...`:

```
from calculos import es_primo
```

Tras esto, la función `es_primo` se utiliza directamente, sin prefijar con el nombre del módulo:

```
es_primo(numero)
```

El antipatrón `from modulo import *`. Existe la variante `from calculos import *`, que importa todos los nombres públicos del módulo de golpe. **En general es una mala práctica**: contamina el espacio de nombres del programa con identificadores cuya procedencia no queda explícita, dificultando la lectura y aumentando el riesgo de colisiones (un nombre del módulo importado puede pisar otro nombre que ya tuvieras definido). Salvo en sesiones interactivas o casos muy concretos (módulos diseñados explícitamente para esto, como ciertas bibliotecas matemáticas), conviene evitar el `import *`.

6.2.3. Alias al importar

Para abreviar nombres largos o evitar colisiones, se puede dar un alias al módulo o a un nombre importado mediante la palabra clave `as`:

```
import calculos as calc
calc.es_primo(numero)
```

Esta forma es muy habitual al trabajar con bibliotecas de terceros: por ejemplo, `import numpy as np` o `import pandas as pd` son convenciones casi universales en el ecosistema científico de Python (Capítulo 12).

6.2.4. Dónde busca Python los módulos: `sys.path`

Cuando se ejecuta un `import nombre`, Python busca un fichero `«nombre.py»` (o un paquete `«nombre/»`) en una lista ordenada de directorios accesible mediante `sys.path`. Esa lista se inicializa con:

- el directorio del script que se está ejecutando,
- los directorios indicados en la variable de entorno `PYTHONPATH` (si está definida), y
- los directorios de la biblioteca estándar y de los paquetes instalados.

Si un `import` falla con `ModuleNotFoundError`, lo primero que conviene comprobar es si el fichero está en un directorio incluido en `sys.path`:

```
import sys
print(sys.path)
```

6.2.5. Paquetes: módulos agrupados en directorios

Cuando un proyecto crece, los módulos sueltos no bastan: conviene agruparlos en *paquetes*. Un paquete es, simplemente, un directorio que contiene varios módulos relacionados y un fichero especial llamado `«__init__.py»` (que puede estar vacío). Su presencia indica a Python que ese directorio debe tratarse como un paquete y permite importar sus módulos con la notación de punto:

```
mi_proyecto/
|-- main.py
'-- utilidades/
    |-- __init__.py
    |-- matematicas.py
    '-- cadenas.py
```

Desde `«main.py»`, podemos importar funciones del paquete así:

```
from utilidades.matematicas import factorial
import utilidades.cadenas as cadenas
```

A partir de Python 3.3, los *namespace packages* (paquetes sin `«__init__.py»`) también funcionan, pero el uso explícito de `«__init__.py»` sigue siendo la convención más clara y compatible.

6.3. Caso práctico: programa dirigido por menú

Una de las maneras más comunes de estructurar un programa interactivo es a través de un programa dirigido por menú, en el que se presenta al usuario un conjunto de opciones y el flujo del programa depende de la elección que haga. Aplicaremos a este caso el enfoque modular y descendente: comenzaremos con la estructura de alto nivel (presentar el menú, leer la opción, ejecutar la operación) y descompondremos en módulos cada operación específica.

Cada opción del menú estará asociada a una función independiente que realice una tarea concreta. Por ejemplo, una opción podría ser calcular el número de cifras de un entero, encapsulada en una función `num_cifras()`. La interacción con el usuario se gestiona de forma centralizada en un módulo principal, que delega cada operación específica al módulo que la implementa.

6.3.1. Análisis del problema

Vamos a desarrollar un programa interactivo que presente un menú con varias opciones para realizar cálculos con números enteros. El menú debe mostrarse iterativamente hasta que el usuario decida finalizar el programa. Un ejemplo de ejecución es:

Ejemplo de ejecución esperada

```

MENÚ DE OPERACIONES
=====
0 - Finalizar
1 - Calcular el número de cifras de un entero
2 - Sumar las cifras de un entero
3 - Extraer una cifra de un entero
4 - Calcular la imagen especular de un entero
5 - Comprobar si un número es primo

Seleccione una operación [0-5]: 4
Escriba un número entero: 8802361
El número imagen especular de 8802361 es 1632088.

```

Aplicando el diseño descendente, dividiremos el programa en dos ficheros:

- El módulo de funciones de cálculo (fichero «`calculos.py`»), que contiene las funciones que realizan las operaciones matemáticas con enteros.
- El módulo principal (fichero «`calculadora.py`»), encargado de manejar la interacción con el usuario y coordinar la ejecución de las operaciones.

6.3.2. Módulo de funciones de cálculo

Este módulo contiene únicamente las funciones de cálculo, sin lógica de interacción con el usuario:

```

1  # Fichero calculos.py
2
3  def num_cifras(n: int) -> int:
4      """Devuelve el número de cifras de `n` en base 10."""
5      return len(str(abs(n)))
6
7
8  def suma_cifras(n: int) -> int:
9      """Devuelve la suma de las cifras de `n` en base 10."""
10     return sum(int(digito) for digito in str(abs(n)))
11
12

```

```
13 def extraer_cifra(n: int, i: int) -> int:
14     """Devuelve la `i`-ésima cifra menos significativa de `n`
    ↪     `.`."""
15     n = str(abs(n))
16     if 1 <= i <= len(n):
17         return int(n[-i])
18     else:
19         raise ValueError("Índice fuera de rango")
20
21
22 def imagen_especular(n: int) -> int:
23     """Devuelve el número que representa la imagen especular
    ↪     de `n`."""
24     signo = -1 if n < 0 else 1
25     return signo * int(str(abs(n))[::-1])
26
27
28 def es_primo(n: int) -> bool:
29     """Devuelve True si `n` es un número primo, False en caso
    ↪     contrario."""
30     if n < 2:
31         return False
32     if n in (2, 3):
33         return True
34     if n % 2 == 0 or n % 3 == 0:
35         return False
36     i = 5
37     while i * i <= n:
38         if n % i == 0 or n % (i + 2) == 0:
39             return False
40         i += 6
41     return True
42
```

```

43 if __name__ == "__main__":
44     # Pruebas unitarias básicas
45     assert num_cifras(12345) == 5
46     assert num_cifras(-999) == 3
47     assert suma_cifras(123) == 6
48     assert suma_cifras(-555) == 15
49     assert extraer_cifra(98765, 1) == 5
50     assert extraer_cifra(98765, 3) == 7
51     assert imagen_especular(123) == 321
52     assert imagen_especular(-456) == -654
53     assert es_primo(2) is True
54     assert es_primo(9) is False
55     assert es_primo(17) is True
56
57     print("Todas las pruebas pasaron correctamente.")

```

Cada operación es una función simple que toma un parámetro (n), realiza la operación correspondiente y retorna el resultado. La función `es_primo()` está optimizada respecto a la forma elemental: tras descartar los casos pequeños y los múltiplos de 2 y 3, comprueba divisores de la forma $6k \pm 1$ hasta $\sqrt{n}$, lo que reduce significativamente el número de divisiones.

Algunas funciones realizan comprobaciones y, si no se cumplen, generan una excepción mediante la sentencia `raise` (introducida en el Capítulo 3).

Pruebas unitarias rápidas con `assert`. Al final del módulo aparecen una serie de sentencias `assert`. La sentencia `assert` verifica que una condición se cumple en tiempo de ejecución: si es `True`, el programa continúa normalmente; si es `False`, se genera una excepción `AssertionError` y la ejecución se detiene. Es útil para escribir pruebas unitarias rápidas sin necesidad de un marco de pruebas como `unittest` o `pytest`. En el módulo «`calculos.py`», cada `assert` comprueba que una función devuelve el resultado esperado para una entrada concreta, facilitando la detección temprana de fallos.

¡Atención! `assert` y modo optimizado

Cuando Python se ejecuta con la opción `-O` (modo optimizado, por ejemplo `python -O script.py`), **todas las sentencias `assert` se ignoran**. Por tanto, `assert` es adecuado para pruebas y comprobaciones de invariantes durante el desarrollo, pero **no debe usarse para validar entradas del usuario, datos externos o requisitos de seguridad**: en producción podría desactivarse y la validación sería ignorada. Para esos casos, lanzar excepciones explícitamente con `raise` es la opción correcta.

6.3.3. Módulo principal

El módulo principal gestiona la interacción con el usuario y delega los cálculos al módulo «`calculos.py`»:

```
1 # Fichero calculadora.py
2
3 import calculos
4
5
6 def mostrar_menu():
7     """Muestra el menú de opciones disponibles."""
8     print("\nMENÚ DE OPERACIONES")
9     print("=====")
10    print("0 - Finalizar")
11    print("1 - Calcular el número de cifras de un entero")
12    print("2 - Sumar las cifras de un entero")
13    print("3 - Extraer una cifra de un entero")
14    print("4 - Calcular la imagen especular de un entero")
15    print("5 - Comprobar si un número es primo")
16
17
18 def pedir_opcion() -> int:
19     """Solicita al usuario una opción del menú."""
20     while True:
21         try:
22             opcion = int(input("Seleccione una operación
23 ↪ [0-5]: "))
24             if 0 <= opcion <= 5:
25                 return opcion
26             else:
27                 print("Por favor, ingrese un número entre 0 y
28 ↪ 5.")
29         except ValueError:
30             print("Entrada no válida. Ingrese un número
31 ↪ entero.")
```

```
31 def ejecutar_operacion(opcion: int):
32     """Ejecuta la operación correspondiente según la opción
    ↪     seleccionada."""
33     if opcion == 0:
34         print("Finalizando el programa.")
35         return False
36
37     try:
38         numero = int(input("Escriba un número entero: "))
39     except ValueError:
40         print("Entrada no válida. Debe ingresar un número
    ↪         entero.")
41         return True
42
43     if opcion == 1:
44         print(f"El número {numero} tiene {calculos.num_cifras
    ↪         (numero)} cifras.")
45     elif opcion == 2:
46         print(f"Las cifras de {numero} suman {calculos.
    ↪         suma_cifras(numero)}.")
47     elif opcion == 3:
48         try:
49             posicion = int(input("Seleccione la posición de
    ↪             una cifra: "))
50             print(f"La cifra en la posición {posicion} de {
    ↪             numero} es {calculos.extraer_cifra(numero,
    ↪             posicion)}.")
51         except ValueError as e:
52             print(e)
53     elif opcion == 4:
54         print(f"El número imagen especular de {numero} es {
    ↪         calculos.imagen_especular(numero)}.")
55     elif opcion == 5:
56         es_primo = calculos.es_primo(numero)
57         print(f"El número {numero} {'es' if es_primo else 'no
    ↪         es'} primo.")
58
59     return True
60
61
```

```
62 def main():
63     """Función principal que ejecuta el programa interactivo.
    ↪     """
64     while True:
65         mostrar_menu()
66         opcion = pedir_opcion()
67         if not ejecutar_operacion(opcion):
68             break
69
70
71 if __name__ == "__main__":
72     main()
```

La función `main()` gestiona el flujo general: muestra el menú con `mostrar_menu()`, lee la opción del usuario con `pedir_opcion()` y, si la opción no es la de finalización, llama a `ejecutar_operacion()` para ejecutar el cálculo correspondiente. Las excepciones se capturan a nivel de la interacción con el usuario para que un fallo de entrada no termine bruscamente el programa.

6.3.4. Niveles de abstracción

El diseño anterior sigue una metodología descendente con tres niveles de abstracción claramente diferenciados:

Nivel 1 (*nivel superior de abstracción*): la función `main()`, que interactúa con el usuario y gestiona el flujo de ejecución del programa.

Nivel 2 (*segundo nivel de abstracción*): las funciones `pedir_opcion()` y `ejecutar_operacion()`, que gestionan la entrada del usuario y la coordinación de las operaciones.

Nivel 3 (*nivel inferior de abstracción*): las funciones del módulo `«calculos.py»` (`num_cifras()`, `suma_cifras()`, etc.), que implementan los detalles técnicos de cada cálculo matemático.

Este tipo de descomposición es lo que hace que el código sea fácil de leer (se entiende primero a alto nivel y solo se baja al detalle cuando hace falta) y fácil de mantener: si más adelante se quisiera optimizar el cálculo de primalidad, solo habría que tocar la función `es_primo()` del módulo de cálculos, sin afectar al resto del programa.

6.4. Ejercicios

EJERCICIO 6.1

Escribe un programa en Python que verifique si una cadena de texto es un pangrama, es decir, si contiene todas las letras del alfabeto al menos una vez. El programa debe eliminar las mayúsculas y los espacios antes de verificar la condición. El flujo de trabajo del programa debe seguir un diseño descendente, comenzando con la función `main()` y desglosándose en módulos más pequeños para verificar si la cadena es un pangrama.

Ejemplo de ejecución esperada

Introduce una cadena de texto: El veloz murciélago hindú comía feliz un cardillo y kiwi,
La cadena es un pangrama.

Solución (girar para leer)

```

1 import string
2
3 def es_pangrama(cadena: str) -> bool:
4     """Verifica si una cadena es un pangrama (contiene
5     todas las letras del alfabeto)."""
6     cadena = cadena.lower() # Convertir a minúsculas
7     alfabeto = set(string.ascii_lowercase) # Alfabeto
8     en minúsculas
9     return set(cadena) >= alfabeto # Comparar si la
10    cadena contiene todas las letras
11
12 def main():
13     texto = input("Introduce una cadena de texto: ")
14     if es_pangrama(texto):
15         print("La cadena es un pangrama.")
16     else:
17         print("La cadena no es un pangrama.")
18
19 if __name__ == "__main__":
20     main()

```

Código 6.1: Solución al Ejercicio 6.1.

EJERCICIO 6.2

Escribe un programa en Python que reciba un texto y cuente cuántas veces aparece cada palabra en el texto. El programa debe ser modular, con una función `contar_palabras()` que se encargue de contar las palabras y una función principal `main()` que gestione la entrada y salida del usuario.

Ejemplo de ejecución esperada

Introduce un texto: hola mundo hola

La frecuencia de las palabras es: 'hola': 2, 'mundo': 1

Solución (girar para leer)

```

1  import string
2
3  def contar_palabras(texto: str) -> dict:
4      """Cuenta la frecuencia de cada palabra en el texto"""
5      texto = texto.lower().translate(str.maketrans("", "",
6      " , string.punctuation)) # Eliminar puntuación
7      palabras = texto.split() # Dividir el texto en
8      palabras
9      frecuencia = {}
10     for palabra in palabras:
11         if palabra in frecuencia:
12             frecuencia[palabra] += 1
13         else:
14             frecuencia[palabra] = 1
15     return frecuencia
16
17 def main():
18     texto = input("Introduce un texto: ")
19     resultado = contar_palabras(texto)
20     print("La frecuencia de las palabras es:",
21         resultado)
22
23 if __name__ == "__main__":
24     main()

```

Código 6.2: Solución al Ejercicio 6.2.

EJERCICIO 6.3

Escribe un programa en Python que calcule la suma de los dígitos de un número de manera recursiva. El programa debe tener un diseño modular, con una función `suma_digitos()` que realice la recursión y una función principal `main()` que gestione la entrada del usuario.

Ejemplo de ejecución esperada

Introduce un número: 9875
La suma de los dígitos es 29

Solución (girar para leer)

```

1  def suma_digitos(n: int) -> int:
2      """Calcula la suma de los dígitos de un número de
3      ← forma recursiva."""
4      if n == 0:
5          return 0
6          return n % 10 + suma_digitos(n // 10)
7  def main():
8      numero = input("Introduce un número: ")
9      resultado = suma_digitos(numero)
10     print(f"La suma de los dígitos es {resultado}")
11
12     if __name__ == "__main__":
13         main()

```

Código 6.3: Solución al Ejercicio 6.3.

EJERCICIO 6.4

Escribe un programa en Python que determine si un número tiene exactamente tres divisores (es decir, si es un número casi primo). Además, el programa debe contar cuántos números casi primos existen hasta un valor límite proporcionado por el usuario. El programa debe ser modular, con una función `es_casi_primo()` que determine si un número es casi primo, y una función principal `main()` para gestionar el flujo general.

Ejemplo de ejecución esperada

Introduce el límite: 10

La cantidad de números casi primos hasta 10 es: 2

Solución (girar para leer)

```

1  def es_casi_primo(n: int) -> bool:
2      """Determina si un número tiene exactamente
3      divisores (es casi primo)."""
4      divisores = 0
5      for i in range(1, n + 1):
6          if n % i == 0:
7              divisores += 1
8          return divisores == 3
9  def contar_casi_primos(límite: int) -> int:
10     """Cuenta cuántos números casi primos hay hasta el
11     límite proporcionado."""
12     contador = 0
13     for i in range(1, límite + 1):
14         if es_casi_primo(i):
15             contador += 1
16     return contador
17 def main():
18     límite = int(input("Introduce el límite: "))
19     cantidad = contar_casi_primos(límite)
20     print(f"La cantidad de números casi primos hasta {
21     límite} es: {cantidad}")
22     if __name__ == "__main__":
23         main()

```

Código 6.4: Solución al Ejercicio 6.4.

EJERCICIO 6.5

Escribe un programa en Python que reciba un número entero y determine la mayor secuencia de dígitos consecutivos dentro de ese número. El programa debe estar diseñado de manera modular, con una función `mayor_secuencia_consecutiva()` que encuentre la secuencia y una función principal `main()` que gestione la entrada y salida de datos.

———— Ejemplo de ejecución esperada ————

Introduce un número: 123456789987654

La mayor secuencia de dígitos consecutivos es 123456789

Solución (girar para leer)

```

1 def mayor-secuencia-consecutiva(n: int) -> str:
2     """Determina la mayor secuencia de dígitos
3     consecutivos en un número."""
4     num_str = str(n)
5     max-secuencia = ""
6     secuencia-actual = num_str[0]
7     for i in range(1, len(num_str)):
8         if int(num_str[i]) == int(num_str[i-1]) + 1:
9             secuencia-actual += num_str[i]
10        else:
11            if len(secuencia-actual) > len(
12                max-secuencia):
13                max-secuencia = secuencia-actual
14            secuencia-actual = num_str[i]
15        if len(secuencia-actual) > len(max-secuencia):
16            max-secuencia = secuencia-actual
17    return max-secuencia
18
19 def main():
20     numero = int(input("Introduce un número: "))
21     secuencia = mayor-secuencia-consecutiva(numero)
22     print(f"La mayor secuencia de dígitos consecutivos
23     es {secuencia}")
24
25 if __name__ == "__main__":
26     main()

```

Código 6.5: Solución al Ejercicio 6.5.

EJERCICIO 6.6

↳ Escribe un programa en Python que implemente una biblioteca de cifrado César y descifrado. La biblioteca debe contener dos funciones: `cifrar_cesar(mensaje: str, desplazamiento: int)` y `descifrar_cesar(mensaje_cifrado: str, desplazamiento: int)`. Luego, el programa debe solicitar al usuario un mensaje, cifrarlo con un desplazamiento dado, mostrar el mensaje cifrado y luego descifrarlo para verificar que se obtiene el mensaje original.

Ejemplo de ejecución esperada

```
Introduce el mensaje: este es un mensaje secreto
Introduce el desplazamiento: 3
Mensaje cifrado: hvwh hv xq phqvmh vhfuhwr
Mensaje descifrado: este es un mensaje secreto
```

Solución (girar para leer)

```

1  def cifrar_cesar(mensaje: str, desplazamiento: int) -> str
2  """Cifra un mensaje usando el cifrado César con un
3  desplazamiento dado."""
4  for char in mensaje:
5      if char.isalpha():
6          # Determina si la letra es mayúscula o minú
7          base = 65 if char.isupper() else 97
8          # Realiza el desplazamiento y asegura que
9          se mantenga dentro del alfabeto
10         mensaje_cifrado += chr((ord(char) - base +
11         desplazamiento) % 26 + base)
12     else:
13         mensaje_cifrado += char # Si no es una
14         letra, lo añade tal cual
15     return mensaje_cifrado
16
17 def descifrar_cesar(mensaje_cifrado: str,
18                     desplazamiento: int) -> str:
19     """Descifra un mensaje cifrado con el cifrado César
20     ."""
21     return cifrar_cesar(mensaje_cifrado, -
22                         desplazamiento)
23
24 def main():
25     mensaje = input("Introduce el mensaje: ")
26     desplazamiento = int(input("Introduce el
27     desplazamiento: "))
28     # Cifrar el mensaje
29     mensaje_cifrado = cifrar_cesar(mensaje,
30     desplazamiento)
31     # Descifrar el mensaje
32     mensaje_descifrado = descifrar_cesar(
33     mensaje_cifrado, desplazamiento)
34     print(f"Mensaje cifrado: {mensaje_cifrado}")
35     print(f"Mensaje descifrado: {mensaje_descifrado}")
36
37 if __name__ == "__main__":
38     main()

```


EJERCICIO 6.7

↳ Escribe un programa en Python que implemente una biblioteca para manipulación de series numéricas. La biblioteca debe contener tres funciones: `es_fibonacci(n: int)` que determina si un número pertenece a la serie de Fibonacci, `es_primo(n: int)` que verifica si un número es primo, y `serie_fibonacci(n: int)` que imprima por pantalla los primeros n números de la serie de Fibonacci. Luego, el programa debe solicitar al usuario un número para verificar si pertenece a la serie de Fibonacci y si es primo, y mostrar los primeros n números de la serie de Fibonacci.

Ejemplo de ejecución esperada

```
Introduce un número: 13
El número 13 es parte de la serie de Fibonacci.
El número 13 es primo.
Introduce cuántos términos de la serie Fibonacci deseas ver: 8
Serie de Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13]
```

Solución (girar para leer)

```

1 import math
2
3 def es_fibonacci(n: int) -> bool:
4     """Verifica si un número pertenece a la serie de
5     Fibonacci usando la fórmula matemática."""
6     if n < 0:
7         return False
8     # Verifica si  $5*n^2 + 4$  o  $5*n^2 - 4$  son cuadrados
9     # perfectos
10    return es_cuadrado_perfecto(5 * n * n + 4) or
11    es_cuadrado_perfecto(5 * n * n - 4)
12
13 def es_cuadrado_perfecto(n: int) -> bool:
14     """Verifica si un número es un cuadrado perfecto."""
15     return int(math.sqrt(n)) ** 2 == n
16
17 def es_primo(n: int) -> bool:
18     """Verifica si un número es primo."""
19     if n <= 1:
20         return False
21     for i in range(2, int(math.sqrt(n)) + 1):
22         if n % i == 0:
23             return False
24     return True

```

Código 6.7: Solución al Ejercicio 6.7.

Solución (girar para leer)

```

22 def serie_fibonacci(n: int) -> list:
23     """Genera los primeros `n` números de la serie de
24     Fibonacci."""
25     if n <= 0:
26         return []
27     elif n == 1:
28         return [0]
29     fibonacci = [0, 1]
30     for i in range(2, n):
31         fibonacci.append(fibonacci[-1] + fibonacci[-2])
32     return fibonacci
33
34 def main():
35     numero = int(input("Introduce un número: "))
36
37     # Comprobar si el número es parte de la serie de
38     # Fibonacci
39     if es_fibonacci(numero):
40         print(f"El número {numero} es parte de la serie
41         de Fibonacci.")
42     else:
43         print(f"El número {numero} no es parte de la
44         serie de Fibonacci.")
45     # Comprobar si el número es primo
46     if es-primo(numero):
47         print(f"El número {numero} es primo.")
48     else:
49         print(f"El número {numero} no es primo.")
50
51     # Mostrar los primeros n términos de la serie de
52     # Fibonacci
53     n = int(input("Introduce cuántos términos de la
54     serie Fibonacci deseas ver: "))
55     print(f"Serie de Fibonacci: {serie_fibonacci(n)}")
56
57     if __name__ == "__main__":
58         main()

```

Código 6.8: Solución al Ejercicio 6.7 (continuación).

EJERCICIO 6.8

Escribe un programa en Python que calcule el área de diferentes formas geométricas (círculo, cuadrado y rectángulo), mostrándola con dos decimales. El cálculo de cada área debe estar en un módulo independiente llamado «formas_geometricas.py». Luego, en el archivo principal, importa este módulo y usa las funciones para calcular el área de cada forma geométrica según la opción que elija el usuario.

Ejemplo de ejecución esperada

```

Selecciona la forma geométrica:
1. Círculo
2. Cuadrado
3. Rectángulo
Introduce tu opción: 1
Introduce el radio del círculo: 5
El área del círculo es 78.54

```

Solución (girar para leer)

```

1 import math
2
3 def area_circulo(radio: float) -> float:
4     """Calcula el área de un círculo dado su radio."""
5     return math.pi * radio ** 2
6
7 def area_cuadrado(lado: float) -> float:
8     """Calcula el área de un cuadrado dado su lado."""
9     return lado ** 2
10
11 def area_rectangulo(base: float, altura: float) -> float:
12     """Calcula el área de un rectángulo dado su base y
13     return base * altura

```

Código 6.9: Solución al Ejercicio 6.8 (archivo «formas_geometricas.py»).

Solución (girar para leer)

```

1  import formas_geometricas
2
3  def main():
4      print("Selecciona la forma geométrica:")
5      print("1. Círculo")
6      print("2. Cuadrado")
7      print("3. Rectángulo")
8
9      opcion = int(input("Introduce tu opción: "))
10
11     if opcion == 1:
12         radio = float(input("Introduce el radio del círculo: "))
13         area = formas_geometricas.area_circulo(radio)
14         print(f"El área del círculo es {area:.2f}")
15
16     elif opcion == 2:
17         lado = float(input("Introduce el lado del cuadrado: "))
18         area = formas_geometricas.area_cuadrado(lado)
19         print(f"El área del cuadrado es {area:.2f}")
20
21     elif opcion == 3:
22         base = float(input("Introduce la base del rectángulo: "))
23         altura = float(input("Introduce la altura del rectángulo: "))
24         area = formas_geometricas.area_rectangulo(base, altura)
25         print(f"El área del rectángulo es {area:.2f}")
26
27     else:
28         print("Opción no válida.")
29
30     if __name__ == "__main__":
31         main()

```

Código 6.10: Solución al Ejercicio 6.8 (fichero principal).

EJERCICIO 6.9

Escribe un programa en Python que convierta temperaturas entre Celsius, Fahrenheit y Kelvin. El cálculo de la conversión debe realizarse en un módulo independiente llamado «`conversion.temperaturas.py`». Luego, en el archivo principal, importa este módulo y usa las funciones para convertir una temperatura proporcionada por el usuario entre las tres unidades. Todas las temperaturas calculadas se mostrarán con dos decimales.

Ejemplo de ejecución esperada

```
Introduce la temperatura en grados Celsius: 25
La temperatura en Fahrenheit es 77.00
La temperatura en Kelvin es 298.15
```

Solución (girar para leer)

```

1  def celsius_a_fahrenheit(celsius: float) -> float:
2      """Convierte temperatura de grados Celsius a
3      Fahrenheit."""
4      return (celsius * 9/5) + 32
5
6  def celsius_a_kelvin(celsius: float) -> float:
7      """Convierte temperatura de grados Celsius a Kelvin
8      ."""
9      return celsius + 273.15
10
11 def fahrenheit_a_celsius(fahrenheit: float) -> float:
12     """Convierte temperatura de grados Fahrenheit a
13     Celsius."""
14     return (fahrenheit - 32) * 5/9
15
16 def fahrenheit_a_kelvin(fahrenheit: float) -> float:
17     """Convierte temperatura de grados Fahrenheit a
18     Kelvin."""
19     return kelvin - 273.15
20
21 def kelvin_a_fahrenheit(kelvin: float) -> float:
22     """Convierte temperatura de Kelvin a Fahrenheit."""
23     return (kelvin - 273.15) * 9/5 + 32

```

Código 6.11: Solución al Ejercicio 6.9 (archivo «conversion-temperaturas.py»).

Solución (girar para leer)

```

1  import conversion_temperaturas
2
3  def main():
4      celsius = float(input("Introduce la temperatura en
5      grados Celsius: "))
6      fahrenheit = conversion_temperaturas.
7      celsius_a_fahrenheit(celsius)
8      kelvin = conversion_temperaturas.celsius_a_kelvin(
9      celsius)
10     print(f"La temperatura en Fahrenheit es {fahrenheit
11     :.2f}")
12     print(f"La temperatura en Kelvin es {kelvin:.2f}")
13
14 if __name__ == "__main__":
15     main()

```

Código 6.12: Solución al Ejercicio 6.9 (archivo principal).

EJERCICIO 6.10

Escribe un programa en Python que convierta entre diferentes unidades de longitud, como metros, kilómetros, millas y yardas. El programa debe estar estructurado en dos partes: (1) un módulo independiente llamado «`conversion_unidades.py`» que contenga funciones para convertir entre las unidades; y (2) el archivo principal, que debe importar este módulo y usar las funciones para convertir entre las unidades de longitud, solicitando al usuario que elija la conversión deseada.

El programa debe permitir la conversión entre las siguientes unidades:

- Metros a Kilómetros
- Kilómetros a Metros
- Millas a Kilómetros
- Kilómetros a Millas
- Yardas a Metros
- Metros a Yardas

Ejemplo de ejecución esperada

Selecciona la conversión que deseas realizar:

1. Metros a Kilómetros
2. Kilómetros a Metros
3. Millas a Kilómetros
4. Kilómetros a Millas
5. Yardas a Metros
6. Metros a Yardas

Introduce tu opción: 1

Introduce el valor en metros: 500

500.0 metros son 0.5 kilómetros

Solución (girar para leer)

Código 6.13: Solución al Ejercicio 6.10 (archivo «conversion-unidades.py»).

```

1  def metros_a_kilometros(metros: float) -> float:
2      """Convierte metros a kilómetros."""
3      return metros / 1000
4
5  def kilometros_a_metros(kilometros: float) -> float:
6      """Convierte kilómetros a metros."""
7      return kilometros * 1000
8
9  def millas_a_kilometros(millas: float) -> float:
10     """Convierte millas a kilómetros."""
11     return millas * 1.60934
12
13 def kilometros_a_millas(kilometros: float) -> float:
14     """Convierte kilómetros a millas."""
15     return kilometros / 1.60934
16
17 def yardas_a_metros(yardas: float) -> float:
18     """Convierte yardas a metros."""
19     return yardas * 0.9144
20
21 def metros_a_yardas(metros: float) -> float:
22     """Convierte metros a yardas."""
23     return metros / 0.9144

```

Solución (girar para leer)

```

def main():
    print("Selecciona la conversión que desees realizar")
    print("1. Metros a Kilómetros")
    print("2. Kilómetros a Metros")
    print("3. Millas a Kilómetros")
    print("4. Kilómetros a Millas")
    print("5. Yardas a Metros")
    print("6. Metros a Yardas")
    option = int(input("Introduce tu opción: "))

    if option == 1:
        metros = float(input("Introduce el valor en metros: "))
        resultado = conversion_unidades.metros_a_kilometros(metros)
        print(f"{metros} metros son {resultado} kilo metros")

    elif option == 2:
        kilometros = float(input("Introduce el valor en kilometros: "))
        resultado = conversion_unidades.kilometros_a_metros(kilometros)
        print(f"{kilometros} kilometros son {resultado} metros")

    elif option == 3:
        millas = float(input("Introduce el valor en millas: "))
        resultado = conversion_unidades.millas_a_kilometros(millas)
        print(f"{millas} millas son {resultado} kilo metros")

    elif option == 4:
        kilometros = float(input("Introduce el valor en kilometros: "))
        resultado = conversion_unidades.kilometros_a_millas(kilometros)
        print(f"{kilometros} kilometros son {resultado} millas")

    elif option == 5:
        yardas = float(input("Introduce el valor en yardas: "))
        resultado = conversion_unidades.yardas_a_metros(yardas)
        print(f"{yardas} yardas son {resultado} metros")

    elif option == 6:
        metros = float(input("Introduce el valor en metros: "))
        resultado = conversion_unidades.metros_a_yardas(metros)
        print(f"{metros} metros son {resultado} yardas")

```

Solución (girar para leer)

```

48 main()
47 if __name__ == "__main__":
46
45     print("Opción no válida.")
44 else:
43
42     print(f"{metros} metros son {resultado} yardas")
41     (metros)
40     resultado = conversion_unidades.metros-a-yardas
39     metros: ""
38     if opcion == 6:
37         print(f"{yardas} yardas son {resultado} metros")
36         (yardas)
35         resultado = conversion_unidades.yardas-a-metros
34         yardas: ""
33         if opcion == 5:
32             yardas = float(input("Introduce el valor en
31             yardas: "))
30             resultado = conversion_unidades.yardas-a-metros
29             (yardas)
28             print(f"{yardas} yardas son {resultado} metros")
27         )
26         if opcion == 5:
25             yardas = float(input("Introduce el valor en
24             yardas: "))
23             resultado = conversion_unidades.yardas-a-metros
22             (yardas)
21             print(f"{yardas} yardas son {resultado} metros")
20         )
19         if opcion == 6:
18             metros = float(input("Introduce el valor en
17             metros: ""
16             resultado = conversion_unidades.metros-a-yardas
15             (metros)
14             print(f"{metros} metros son {resultado} yardas")
13         )
12     )
11
10
9
8
7
6
5
4
3
2
1

```

7

Enteros

En este capítulo exploraremos cómo manejar cálculos con números enteros en Python, resolviendo problemas comunes relacionados con la manipulación de cifras y la divisibilidad. Python proporciona una representación flexible de los enteros y evita los problemas de desbordamiento presentes en otros lenguajes de programación, lo que resulta especialmente útil cuando se trabaja con números muy grandes, como en cálculo científico, combinatoria o teoría de números.

7.1. Representación de números enteros en Python

En Python, los números enteros se representan con el tipo `int`, que tiene las siguientes características:

- No hay un límite de tamaño fijo: Python maneja enteros de **precisión arbitraria**. Esto significa que no se produce desbordamiento cuando se realizan operaciones matemáticas con números arbitrariamente grandes (el único límite es la memoria disponible).
- Los enteros se pueden escribir en base decimal, binaria (prefijo `0b`), octal (prefijo `0o`) o hexadecimal (prefijo `0x`).
- Soportan los operadores aritméticos estándar, como veremos en la siguiente sección.

Por ejemplo, el siguiente código define cuatro variables enteras escritas en distintas bases:

```
a = 2147483647    # Un entero en base 10
b = 0b1010        # Un entero en base 2 (binario)
c = 0o12           # Un entero en base 8 (octal)
d = 0xA           # Un entero en base 16 (hexadecimal)
```

```
print(a, b, c, d) # Salida: 2147483647 10 10 10
```

7.1.1. Conversiones entre bases

Aunque internamente todos los enteros se almacenan igual, es muy frecuente – sobre todo en programación de bajo nivel y manipulación de datos binarios – necesitar la representación textual de un entero en una base distinta. Python ofrece funciones específicas para ello:

- `bin(n)`: devuelve la representación binaria de `n` como cadena, prefijada con `0b`.
- `oct(n)`: devuelve la representación octal, prefijada con `0o`.
- `hex(n)`: devuelve la representación hexadecimal, prefijada con `0x`.
- `int(s, base)`: convierte una cadena `s` a entero interpretándola en la base indicada (entre 2 y 36).

Por ejemplo:

```
print(bin(255))          # Salida: 0b11111111
print(hex(255))          # Salida: 0xff
print(int("ff", 16))     # Salida: 255
print(int("1010", 2))    # Salida: 10
```

7.2. Operadores aritméticos

Python proporciona los operadores aritméticos habituales sobre enteros, recogidos en la Tabla 1.

Operador	Descripción	Ejemplo
+	Suma	$7 + 3 \rightarrow 10$
-	Resta	$7 - 3 \rightarrow 4$
*	Multipliación	$7 * 3 \rightarrow 21$
/	División real (<i>float</i>)	$7 / 3 \rightarrow 2.333\dots$
//	División entera	$7 // 3 \rightarrow 2$
%	Resto (módulo)	$7 \% 3 \rightarrow 1$
**	Exponenciación	$2 ** 10 \rightarrow 1024$

Tabla 1: Operadores aritméticos sobre enteros.

Cada uno de estos operadores tiene también su forma compuesta de asignación (`+=`, `-=`, `*=`, `//=`, etc.), de uso muy frecuente.

Además, Python ofrece dos funciones *built-in* muy útiles para trabajar con enteros:

- `divmod(a, b)`: devuelve la tupla `(a // b, a % b)` en una sola operación. Es más eficiente que calcular cociente y resto por separado cuando se necesitan ambos.
- `pow(base, exp)` y `pow(base, exp, mod)`: la primera forma es equivalente a `base ** exp`; la segunda calcula `(base ** exp) % mod` pero de manera mucho más eficiente. Esta segunda forma se denomina **exponenciación modular** y es fundamental en criptografía moderna (RSA, Diffie-Hellman).

```

print(divmod(17, 5))          # Salida: (3, 2)
print(pow(3, 4))              # Salida: 81
print(pow(3, 4, 5))           # Salida: 1    (porque 81 % 5 ==
    1)
print(pow(2, 1024, 1000003)) # Cálculo modular eficiente con
    exponente grande

```

Eficiencia de `pow(a, b, m)`

Calcular `(a ** b) % m` con `b` muy grande es prohibitivo: primero calcula la potencia entera completa (que puede tener millones de cifras) y luego le aplica el módulo. `pow(a, b, m)` usa *exponenciación rápida modular*, manteniendo en cada paso el resultado reducido módulo `m`. La diferencia de rendimiento es de varios órdenes de magnitud en cuanto los exponentes superan unos pocos miles de bits.

7.3. Desbordamientos

A diferencia de otros lenguajes donde los enteros tienen un tamaño fijo (como C++, donde están limitados a 4 u 8 bytes según la arquitectura), en Python no hay desbordamiento de enteros: el tipo `int` ajusta dinámicamente su tamaño. Los enteros en Python crecen automáticamente según la memoria disponible, por lo que se puede trabajar con números muy grandes sin preocuparse por el desbordamiento. Sin embargo, el rendimiento puede verse afectado cuando se manejan números extremadamente grandes.

Por tanto, errores debidos a problemas de precisión derivados de desbordamientos no van a aparecer en Python. Por ejemplo, el siguiente código:

```

1 def factorial(n):
2     if n < 0:
3         raise ValueError("El factorial no esta definido para
    negativos.")
4     result = 1
5     for i in range(1, n + 1):
6         result *= i
7     return result

```

calculará correctamente el factorial de cualquier número pasado por parámetro sin producir errores de desbordamiento (mientras haya memoria disponible). Por ejemplo, `factorial(1000)` devuelve un entero con 2.568 cifras y Python lo maneja sin problema.

7.4. El tipo `bool` y su relación con los enteros

En Python, el tipo `bool` es un *subtipo* de los enteros. Los valores `True` y `False` se representan internamente como los enteros 1 y 0, respectivamente. Esto significa que se pueden usar en contextos aritméticos directamente, lo cual a veces resulta cómodo (por ejemplo, para contar cuántas condiciones se cumplen). Por ejemplo:

```
print(int(True))      # Salida: 1
print(int(False))     # Salida: 0
print(True + True)    # Salida: 2
```

7.5. Bibliotecas numéricas: el módulo `math`

Python proporciona un conjunto amplio de funciones matemáticas en el módulo `math`. Algunas de las más utilizadas con enteros son:

- `math.factorial(n)`: calcula el factorial de un número.
- `math.gcd(a, b, ...)`: calcula el máximo común divisor de los argumentos.
- `math.lcm(a, b, ...)`: calcula el mínimo común múltiplo.
- `math.isqrt(n)`: calcula la parte entera de la raíz cuadrada (sin pasar por `float`, evitando errores de redondeo).
- `math.comb(n, k)` y `math.perm(n, k)`: combinaciones y permutaciones.

Por ejemplo:

```
import math
print(math.factorial(5))  # Salida: 120
print(math.gcd(48, 18))  # Salida: 6
print(math.isqrt(10**18)) # Sin perdida de precision
```

`math.sqrt` vs `math.isqrt`

`math.sqrt(n)` devuelve un `float` y, para enteros grandes, puede sufrir errores de redondeo (por ejemplo, devolver 9.999999999999998 en lugar de 10). Si solo necesitas la parte entera de la raíz, usa `math.isqrt(n)`: trabaja siempre en aritmética entera y es exacta.

7.6. Manipulación de números enteros

Los siguientes problemas se enfocan en cómo analizar y manipular los dígitos de un número entero cuando se expresa en base 10.

7.6.1. Contar el número de cifras de un entero

Para contar cuántas cifras tiene un número, se puede simplemente utilizar `len(str(n))` (transformamos el número a una cadena, y miramos cuál es la longitud de la cadena). Para ilustrar el proceso manual, implementamos el siguiente algoritmo:

```

1 def num_cifras(n):
2     """Devuelve el número de cifras de n en base 10."""
3     n = abs(n) # Convertimos a positivo si es negativo
4     cuenta = 1
5     while n >= 10:
6         n //= 10 # Eliminamos la última cifra
7         cuenta += 1
8     return cuenta
9
10 print(num_cifras(65030)) # Salida: 5
11 print(num_cifras(-103044)) # Salida: 6

```

El código define una función llamada `num_cifras` que tiene como objetivo contar el número de cifras que tiene un número entero `n` cuando se expresa en base 10.

En primer lugar, la función toma el valor absoluto del número (`n = abs(n)`) para asegurarse de que, si el número es negativo, se convierta en positivo, ya que el signo no afecta al número de cifras.

La variable `cuenta` se inicializa en 1 porque, al menos, el número tendrá una cifra. Luego, se utiliza un bucle `while` para dividir el número `n` por 10 repetidamente. En cada iteración, se elimina la última cifra usando el operador `//=`, que realiza una división entera y asigna el cociente de vuelta a `n`. Esto reduce el número por un factor de 10 en cada paso, eliminando la cifra menos significativa (la de más a la derecha).

En cada iteración, se incrementa la variable `cuenta` en 1 para llevar la cuenta de cuántas veces se ha dividido el número, lo que equivale a cuántas cifras tiene. El bucle continúa ejecutándose hasta que `n` sea menor que 10, lo que indica que solo queda la última cifra.

7.6.2. Suma de las cifras de un número entero

La suma de los dígitos de un número se puede calcular tal y como se muestra en el siguiente código:

```
1 def suma_cifras(n):
2     """Devuelve la suma de las cifras de n en base 10."""
3     n = abs(n)
4     suma = 0
5     while n > 0:
6         suma += n % 10 # Extraemos la última cifra y la
7             ↪      sumamos
8         n //= 10        # Eliminamos la última cifra
9     return suma
10 print(suma_cifras(65030))    # Salida: 14
11 print(suma_cifras(-103044)) # Salida: 12
```

El código define una función llamada `suma_cifras` que calcula la suma de las cifras de un número entero `n` cuando se expresa en base 10.

La función comienza tomando el valor absoluto del número (`n = abs(n)`), lo que garantiza que si `n` es negativo se convierte en positivo. Esto es importante porque el signo negativo no afecta a la suma de las cifras.

La variable `suma` se inicializa en 0 y acumulará la suma de las cifras del número.

El bucle `while n > 0` se ejecuta mientras `n` sea mayor que cero. Dentro del bucle, la operación `n % 10` extrae la última cifra del número: el operador `%` es el operador de módulo, y al hacer `n % 10` obtenemos el resto de dividir `n` entre 10, lo que equivale a la última cifra. Esa cifra extraída se acumula en `suma` con la instrucción `suma = suma + n % 10`.

Después de sumar la última cifra, el número `n` se divide por 10 con el operador de división entera y asignación `//=`. Esto elimina la última cifra. El bucle continúa ejecutándose hasta que `n` sea cero, es decir, hasta que no haya más cifras por procesar.

Una vez que el bucle termina, la función devuelve la variable `suma`, que contiene la suma total de las cifras del número original.

7.6.3. Obtener la *i*-ésima cifra menos significativa

Para extraer la *i*-ésima cifra menos significativa, eliminamos las cifras menos significativas hasta llegar a la deseada:

```
1 def cifra(n, i):
2     """Devuelve la i-esima cifra menos significativa de n."""
3     n = abs(n) # Trabajamos solo con valores positivos
4     for _ in range(i - 1):
5         n //= 10 # Eliminamos cifras
6     return n % 10 # Devolvemos la última cifra
7
8 print(cifra(6470431, 1))    # Salida: 1
9 print(cifra(6470431, 3))    # Salida: 4
10 print(cifra(-6470431, 5))   # Salida: 7
```

El código define una función llamada `cifra` que devuelve la i -ésima cifra menos significativa de un número entero n . La función recibe dos parámetros: n , que es el número del cual queremos extraer la cifra, e i , que es la posición de la cifra que queremos obtener, comenzando desde 1 (es decir, la primera cifra menos significativa corresponde a $i = 1$).

La primera línea dentro de la función es $n = \text{abs}(n)$, que convierte n en su valor absoluto. Esto asegura que trabajamos solo con valores positivos, ya que el signo de un número no afecta a sus cifras individuales.

El siguiente paso es eliminar las primeras $i - 1$ cifras del número. Esto se logra mediante el bucle `for _ in range(i - 1)`. En este caso, se usa el carácter `_` como variable de control, lo que indica que la variable no se va a utilizar dentro del bucle (es una convención en Python para indicar que el valor de la variable no es relevante y que solo se necesita ejecutar el bucle un número determinado de veces). En cada iteración del bucle, se realiza una división entera de n entre 10 (sentencia $n //= 10$), eliminando la última cifra. El bucle se repite $i - 1$ veces, eliminando así las primeras $i - 1$ cifras menos significativas.

Por ejemplo, si estamos buscando la tercera cifra menos significativa ($i = 3$), el bucle se ejecutará dos veces, eliminando las dos primeras cifras menos significativas del número. Después de que el bucle ha eliminado las primeras $i - 1$ cifras, lo que queda en n es un número cuyo último dígito es la cifra que queremos obtener. Para obtenerla, utilizamos el operador de módulo ($n \% 10$), que devuelve el resto de la división de n por 10, es decir, su último dígito. Esta cifra es la que se devuelve finalmente con `return n \% 10`.

7.6.4. Invertir un número (imagen especular)

Para invertir un número, podemos usar la siguiente función:

```

1 def imagen(n):
2     """Devuelve la imagen especular de n."""
3     negativo = n < 0
4     n = abs(n)
5     invertido = 0
6     while n > 0:
7         invertido = invertido * 10 + n \% 10 # Construimos el
8         n //= 10                               número invertido
9     return -invertido if negativo else invertido
10
11 print(imagen(123456))    # Salida: 654321
12 print(imagen(-103044))  # Salida: -440301

```

El código define una función llamada `imagen` que devuelve la imagen especular de un número entero n , es decir, el número obtenido al invertir el orden de sus cifras.

La primera línea de la función evalúa si el número n es negativo. Si lo es, la variable `negativo` se establecerá como `True`, en caso contrario como `False`. Esto es importan-

te para que, al final del proceso, podamos devolver la imagen especular con el signo adecuado.

Después se convierte `n` a su valor absoluto (`n = abs(n)`) para que el proceso de inversión no se vea afectado por el signo. A continuación, comienza el proceso de inversión: la variable `invertido` se inicializa a 0; a medida que vamos procesando las cifras del número, iremos construyendo la versión invertida en esta variable.

El bucle principal extrae la última cifra de `n` con el operador de módulo y la añade a `invertido`, que previamente se ha multiplicado por 10 para desplazar a la izquierda las cifras ya procesadas. Después se elimina la última cifra de `n` usando la división entera (`n //= 10`). El bucle se repite hasta que `n` sea 0, lo que significa que ya hemos procesado todas las cifras.

Una vez invertido el número, comprobamos si el número original era negativo. Si lo era, devolvemos el invertido con signo negativo; si era positivo, lo devolvemos tal cual. Concretamente, la última línea de la función,

```
1 | return -invertido if negativo else invertido
```

es una expresión condicional (también conocida como *expresión ternaria*) que permite devolver uno de dos valores dependiendo de la condición que se evalúe. El mismo comportamiento podría lograrse con una sentencia `if` tradicional:

```
1 | if negativo:
2 |     return -invertido
3 | else:
4 |     return invertido
```

La expresión ternaria es simplemente una forma más concisa de lograr lo mismo.

7.6.5. Comprobar la primalidad de un número

Un número es primo si es mayor o igual a 2 y no es divisible más que por 1 y por sí mismo. Para mejorar la eficiencia, verificamos divisores hasta la raíz cuadrada del número:

```
1 | import math
2 |
```

```

3 def es_primo(n):
4     """Devuelve True si n es un número primo, False en caso
↪     contrario."""
5     if n < 2:
6         return False
7     if n == 2:
8         return True
9     if n % 2 == 0:
10        return False
11    for divisor in range(3, math.isqrt(n) + 1, 2):
12        if n % divisor == 0:
13            return False
14    return True
15
16 print(es_primo(29))    # Salida: True
17 print(es_primo(30))    # Salida: False

```

El código define una función llamada `es_primo` que determina si un número `n`, dado por parámetro, es primo o no. La función comienza verificando si `n` es menor que 2. Si es así, devuelve `False` porque no existen números primos menores que 2 (por convención, el 1 no se considera primo).

A continuación aparece el caso especial del número 2: el único par que es primo. Si `n` es 2, la función devuelve `True`.

Si no se cumple ninguna de las condiciones anteriores, la función comprueba si `n` es divisible entre 2, lo que significa que `n` es un número par. Todos los pares, excepto el 2, no son primos; por lo tanto, si `n` es divisible entre 2, la función devuelve `False`.

Si el número no es menor que 2, no es 2, y no es divisible entre 2, entonces verificamos si tiene divisores impares. Para hacer esto, la función utiliza un bucle `for` que comienza desde 3 y va hasta la raíz cuadrada de `n`. Esto se debe a que si un número `n` tiene un divisor mayor que su raíz cuadrada, también tendrá uno menor, por lo que no es necesario comprobar divisores más grandes que la raíz cuadrada. Además, solo verificamos divisores impares, ya que hemos descartado los pares (excepto el 2): por eso el paso es 2 en la función `range`.

Hemos utilizado `math.isqrt(n)` para obtener la parte entera exacta de la raíz cuadrada de `n`, evitando así los errores de redondeo en coma flotante que tendríamos con `int(math.sqrt(n))` para enteros muy grandes. Se verifica la divisibilidad con `n % divisor == 0`. Si en algún momento encontramos que `n` es divisible por alguno de los divisores impares, la función devuelve `False`, indicando que `n` no es primo. Si no se encuentran divisores durante la iteración (lo que significa que `n` no es divisible por ningún número distinto de 1 y de sí mismo), la función devuelve `True`.

7.6.6. Cálculo del máximo común divisor

El máximo común divisor (MCD) de dos números se puede calcular de forma eficiente usando el **algoritmo de Euclides**. Este algoritmo se basa en una propiedad fundamental de los números enteros: el MCD de dos números a y b es el mismo que el MCD de b y el resto de a dividido por b , es decir, $MCD(a, b) = MCD(b, a \bmod b)$. Este proceso se repite hasta que el resto sea cero, momento en el cual el MCD será el último valor distinto de cero. El algoritmo es eficiente porque, a medida que se calculan los restos, los números involucrados se hacen más pequeños, lo que reduce significativamente la cantidad de iteraciones necesarias.

El siguiente código implementa este algoritmo:

```
1 def mcd(a, b):
2     """Devuelve el máximo común divisor de a y b usando el
   ↪     algoritmo de Euclides."""
3     a, b = abs(a), abs(b) # Trabajamos con valores absolutos
4     while b:
5         a, b = b, a % b # Aplicamos iterativamente el
   ↪         algoritmo de Euclides
6     return a
7
8 print(mcd(48, 18))          # Salida: 6
9 print(mcd(-103044, 65030)) # Salida: 2
```

En la primera línea de la función, los números a y b se convierten a sus valores absolutos utilizando la función `abs()`. Esto se hace porque el MCD es el mismo si los números son negativos o positivos, y el algoritmo de Euclides funciona sin importar el signo.

El bucle `while b` continúa ejecutándose mientras b no sea 0. En cada iteración, el valor de a se actualiza a b , y el valor de b se actualiza al resto de $a \% b$. Este proceso se repite hasta que b se convierte en 0, momento en el que termina el bucle. El valor final de a es el MCD de los dos números originales.

El algoritmo de Euclides es la base sobre la que se construye el *algoritmo de Euclides extendido*, que además de calcular $MCD(a, b)$ encuentra los coeficientes de Bézout (x, y tales que $ax + by = MCD(a, b)$). Esta extensión se utiliza, por ejemplo, para calcular el inverso modular, una operación que aparece en numerosos algoritmos numéricos.

7.7. Aplicación: representación binaria de enteros

La precisión arbitraria de los enteros en Python, combinada con sus operaciones de conversión a bytes, hace que el lenguaje sea especialmente cómodo para trabajar con representaciones binarias: serializar y deserializar enteros, interpretar cabeceras de ficheros o de protocolos, o manipular datos a bajo nivel. En esta sección recogemos algunas funcionalidades del lenguaje que aparecen de forma recurrente en ese tipo de tareas.

7.7.1. Bits, bytes y enteros: `bit_length`, `to_bytes`, `from_bytes`

Todo entero en Python lleva asociadas tres operaciones muy útiles para trabajar con representaciones binarias:

- `n.bit_length()`: devuelve el número mínimo de bits necesarios para representar `n` en binario, sin contar el signo ni los ceros a la izquierda.
- `n.to_bytes(longitud, byteorder)`: convierte el entero `n` en una secuencia de `longitud` bytes, en el orden indicado (`'big'` o `'little'`).
- `int.from_bytes(secuencia, byteorder)`: la operación inversa, reconstruye el entero a partir de los bytes.

Por ejemplo, para consultar cuántos bits ocupa un número y serializarlo como una secuencia de bytes:

```
clave = 0xDEADBEEFCAFEBAE
print(clave.bit_length())           # Salida: 64

paquete = clave.to_bytes(8, 'big') # b'\xde\xad\xbe\xef\xca\xfe\xba\xbe'
print(int.from_bytes(paquete, 'big') == clave) # Salida:
True
```

Este es el mecanismo natural para serializar y deserializar enteros cuando interpretamos cabeceras de protocolos, ficheros binarios o cualquier formato a bajo nivel.

7.8. Ejercicios

EJERCICIO 7.1

Escribe una función en Python que reciba un número entero n y determine el número total de divisores de n (incluyendo el número n mismo). El programa debe ser eficiente y calcular los divisores sólo hasta la raíz cuadrada de n para optimizar el tiempo de ejecución.

Ejemplo de ejecución esperada

Introduce un número: 36
El número 36 tiene 9 divisores

Solución (girar para leer)

```

1 import math
2
3 def numero_divisores(n):
4     """Devuelve el número total de divisores de n
5     incluyendo n mismo)."""
6     count = 0
7     # Iteramos solo hasta la raíz cuadrada de n
8     for i in range(1, int(math.sqrt(n)) + 1):
9         if n % i == 0: # Si i es divisor de n
10             count += 1 # Contamos el divisor i
11             # Si i no es la raíz
12             # cuadrada de n
13             count += 1 # Contamos el divisor
14             complementario n // i
15     return count
16
17 # Interacción con el usuario
18 n = int(input("Introduce un número: "))
19 print(f"El número {n} tiene {numero_divisores(n)}
20 divisores")

```

Código 7.1: Solución al Ejercicio 7.1.

EJERCICIO 7.2

Escribe una función en Python que reciba un número entero n y calcule la suma de los primeros n números primos. El programa debe ser eficiente en cuanto al tiempo de ejecución y manejar correctamente valores grandes de n . Al final, debe mostrar el resultado de la suma.

Ejemplo de ejecución esperada

Introduce un número: 5

La suma de los primeros 5 números primos es 28

Solución (girar para leer)

```

1  def es-primo(num):
2      """Devuelve True si el número es primo, False en
3      caso contrario."""
4      if num < 2:
5          return False
6      for i in range(2, int(num**0.5) + 1):
7          # comprobamos si el número es divisible por algún
8          # número hasta su raíz cuadrada
9          if num % i == 0:
10             return False
11         return True
12     def suma-primos(n):
13         """Devuelve la suma de los primeros n números
14         primos."""
15         suma = 0
16         num = 2
17         count = 0
18         while count < n:
19             if es-primo(num):
20                 # Usamos la función es-primo
21                 # para verificar si num es primo
22                 suma += num
23                 count += 1
24             num += 1
25         return suma
26     # Interacción con el usuario
27     n = int(input("Introduce un número: "))
28     print(f"La suma de los primeros {n} números primos es {
29     suma-primos(n)}")

```

Código 7.2: Solución al Ejercicio 7.2.

EJERCICIO 7.3

Escribe una función en Python que reciba un número entero n y determine si n es un cuadrado perfecto. Un número es un cuadrado perfecto si puede expresarse como el cuadrado de un número entero. Es decir, existe un número entero m tal que $n = m^2$.

———— Ejemplo de ejecución esperada ————

Introduce un número: 25

El número 25 es un cuadrado perfecto.

Solución (girar para leer)

```

1 import math
2
3 def cuadrado-perfecto(n):
4     """Devuelve True si n es un cuadrado perfecto,
5     False en caso contrario."""
6     raiz = int(math.sqrt(n))
7     return raiz * raiz == n
8
9 # Interacción con el usuario
10 n = int(input("Introduce un número: "))
11 if cuadrado-perfecto(n):
12     print(f"El número {n} es un cuadrado perfecto.")
13 else:
14     print(f"El número {n} no es un cuadrado perfecto.")

```

Código 7.3: Solución al Ejercicio 7.3.

EJERCICIO 7.4

Escribe una función en Python que reciba un número entero n y una base b , y calcule el número de dígitos de n cuando se representa en la base b . La función debe devolver la cantidad de dígitos en la representación de n en base b .

Ejemplo de ejecución esperada

Introduce un número: 255

Introduce la base: 16

El número 255 en base 16 tiene 2 dígitos.

Solución (girar para leer)

```

1 def numero_digitos_base(n, b):
2     """Devuelve el número de dígitos de n en base b."""
3     count = 0
4     while n > 0:
5         n //= b # Dividimos el número entre la base b
6         count += 1 # Incrementamos el contador de dígitos
7     return count if count > 0 else 1 # Si count es 0,
8     el número tiene 1 dígito (para el caso n=0)
9 # Interacción con el usuario
10 n = int(input("Introduce un número: "))
11 b = int(input("Introduce la base: "))
12 print(f"El número {n} en base {b} tiene {numero_digitos_base(n, b)} dígitos.")

```

Código 7.4: Solución al Ejercicio 7.4.

EJERCICIO 7.5

Escribe una función en Python que reciba un número entero n y determine si n es un número triangular. Un número triangular es aquel que puede representarse como la suma de los primeros k números naturales, es decir, $n = 1 + 2 + 3 + \dots + k$. El k -ésimo número triangular se puede calcular como $n = k(k + 1)/2$.

— Ejemplo de ejecución esperada —

Introduce un número: 10

El número 10 es triangular.

Solución (girar para leer)

Código 7.5: Solución al Ejercicio 7.5.

```

1 def es_triangular(n):
2     """Devuelve True si n es un número triangular,
3     False en caso contrario."""
4     suma = 0
5     while suma < n:
6         suma += 1
7         k += 1
8     return (suma == n)
9
10 # Interacción con el usuario
11 n = int(input("Introduce un número: "))
12 if es_triangular(n):
13     print(f"El número {n} es triangular.")
14 else:
15     print(f"El número {n} no es triangular.")

```

EJERCICIO 7.6

Escribe un programa en Python que reciba dos números enteros a y b y determine si son **números amistosos**. Dos números son amistosos si la suma de los divisores propios de a es igual a b y la suma de los divisores propios de b es igual a a . Por ejemplo, los números 220 y 284 son números amistosos, porque la suma de los divisores de 220 (excluyendo 220) es 284, y la suma de los divisores de 284 (excluyendo 284) es 220.

Ejemplo de ejecución esperada

Introduce el primer número: 220
 Introduce el segundo número: 284
 Los números 220 y 284 son amistosos.

Solución (girar para leer)

```

1  def suma_divisores(n):
2      """Devuelve la suma de los divisores propios de n (
3      excluyendo n mismo)."""
4      suma = 0
5      for i in range(1, int(n**0.5) + 1):
6          if n % i == 0:
7              suma += i
8              if i != n // i:
9                  suma += n // i
10         return suma
11     def son_amistosos(a, b):
12         """Devuelve True si a y b son números amistosos,
13         False en caso contrario."""
14         return suma_divisores(a) == b and suma_divisores(b) == a
15     # Interacción con el usuario
16     a = int(input("Introduce el primer número: "))
17     b = int(input("Introduce el segundo número: "))
18     if son_amistosos(a, b):
19         print(f"Los números {a} y {b} son amistosos.")
20     else:
21         print(f"Los números {a} y {b} no son amistosos.")

```

Código 7.6: Solución al Ejercicio 7.6.

EJERCICIO 7.7

Escribe un programa en Python que reciba un número entero n y determine si n es un número práctico. Un número práctico es un número entero positivo tal que todos los números menores que él son la suma de un subconjunto de sus divisores propios. Por ejemplo, el número 6 es práctico, porque sus divisores son 1, 2 y 3, y todos los números menores que 6 pueden formarse sumando los divisores de 6. Los números prácticos son una generalización de los números perfectos.

Ejemplo de ejecución esperada

Introduce un número: 6
El número 6 es práctico.

Solución (girar para leer)

```

1  def es_practico(n):
2      """Devuelve True si n es un número práctico, False
3      en caso contrario."""
4      suma = 0
5      for i in range(1, n):
6          if n % i == 0:
7              suma += i
8      return suma <= n
9  # Interacción con el usuario
10 n = int(input("Introduce un número: "))
11 if es_practico(n):
12     print(f"El número {n} es práctico.")
13 else:
14     print(f"El número {n} no es práctico.")

```

Código 7.7: Solución al Ejercicio 7.7.

EJERCICIO 7.8

Escribe un programa en Python que reciba una lista de números enteros y calcule el máximo común divisor de todos ellos utilizando el algoritmo de Euclides.

Ejemplo de ejecución esperada

Introduce los números (separados por espacios): 48 180 36
El máximo común divisor de los números es 12

Solución (girar para leer)

```

1 import math
2
3 def mcd(a, b):
4     """Devuelve el máximo común divisor de a y b usando
5     el algoritmo de Euclides."""
6     while b:
7         a, b = b, a % b
8     return a
9
10 # Interacción con el usuario
11 entrada = input("Introduce los números separados por
12 espacios: ")
13 mcd_resultado = mcd(int(entrada[0]), # El primer número se
14 asigna a mcd_resultado
15 for num in entrada[1:]:
16     mcd_resultado = mcd(mcd_resultado, int(num))
17     # Calculamos el MCD con cada número posterior
18
19 print(f"El máximo común divisor de los números es {
20 mcd_resultado}")

```

Código 7.8: Solución al Ejercicio 7.8.

EJERCICIO 7.9

Escribe una función en Python que reciba un número entero n y calcule el n -ésimo número de Fibonacci de manera eficiente utilizando programación dinámica^a.

Ejemplo de ejecución esperada

Introduce un número: 100

El 100-ésimo número de Fibonacci es 354224848179261915075

^aLa programación dinámica es una técnica de optimización de algoritmos que se utiliza para resolver problemas que pueden ser descompuestos en subproblemas solapados. Es especialmente útil para problemas donde el mismo subproblema se resuelve varias veces durante la ejecución de un algoritmo. La idea clave en programación dinámica es almacenar los resultados de los subproblemas para evitar volver a calcularlos, lo que mejora la eficiencia del algoritmo.

Solución (girar para leer)

```

1 def fibonacci_grande(n):
2     """Devuelve el n-ésimo número de Fibonacci usando
3     programación dinámica (tabulación)."""
4     if n <= 1:
5         return n
6     a, b = 0, 1
7     for _ in range(2, n + 1):
8         a, b = b, a + b
9     return b
10 # Interacción con el usuario
11 n = int(input("Introduce un número: "))
12 print(f"El {n}-ésimo número de Fibonacci es {
13     fibonacci_grande(n)}")

```

Código 7.9: Solución al Ejercicio 7.9.

EJERCICIO 7.10

Escribe un programa en Python que reciba un número entero n y calcule su factorización prima, es decir, descomponga n en los factores primos que lo multiplican.

Ejemplo de ejecución esperada

Introduce un número: 60

La factorización prima de 60 es: 2, 2, 3, 5

Solución (girar para leer)

```

1 def factorizacion-prima(n):
2     """Imprime la factorización prima de n."""
3     # comprobamos los divisores para los números
4     menores que n
5     # comenzamos con el número 2
6     while n % 2 == 0:
7         print(2, end=" ", " ")
8         n //= 2
9     # Ahora verificamos los números impares desde 3 en
10    adelante
11    divisor = 3
12    while divisor * divisor <= n:
13        while n % divisor == 0:
14            print(divisor, end=" ", " ")
15            n //= divisor
16        divisor += 2
17    # Si n es mayor que 2, entonces n es un número
18    primo
19    if n > 2:
20        print(n)
21    factorizacion-prima(n)
22    print(f"La factorización prima de {n} es: ", end=" ")

```

Código 7.10: Solución al Ejercicio 7.10.

8

Flotantes (números reales)

En este capítulo exploraremos cómo Python maneja los números reales, centrándonos en su representación, las operaciones básicas y las limitaciones inherentes al uso de la *aritmética en coma flotante*. Veremos cómo Python utiliza el tipo `float` y cómo se gestionan casos especiales como el desbordamiento (`inf`) y los valores no definidos (`nan`). También analizaremos las herramientas que Python ofrece para realizar cálculos con números reales de forma fiable, como la biblioteca `math` y los módulos `decimal` y `fractions`. La aritmética en coma flotante es relevante en ciencia de datos y en muchos otros dominios, como la ciberseguridad: numerosos algoritmos numéricos y modelos de aprendizaje automático toman decisiones basadas en valores reales, y entender cómo (y cuándo) fallan estos cálculos resulta esencial.

8.1. Representación de números reales en Python

En Python, los números reales se representan con el tipo `float`, un número de **coma flotante de doble precisión** (64 bits). Sigue el estándar IEEE 754, que regula cómo se almacenan y manipulan los números de coma flotante en la mayoría de plataformas.

Internamente, los 64 bits se dividen en tres partes:

- El **bit de signo** (1 bit), que determina si el número es positivo o negativo.
- El **exponente** (11 bits), que define la potencia de 2 por la que se multiplica la parte significativa.
- La **mantisa** (52 bits), que representa la parte fraccionaria del número.

El rango de valores que puede representar el tipo `float` es amplio, aproximadamente $[-1,7 \cdot 10^{308}, 1,7 \cdot 10^{308}]$, con una precisión relativa de unos 15 a 16 dígitos decimales significativos. Para que te hagas una idea de la magnitud, este rango es suficiente para enumerar todas y cada una de las estrellas del Universo visible.

8. Flotantes (números reales)

Sin embargo, esta representación binaria **no permite expresar de forma exacta todos los números decimales**. Por ejemplo, 0,1 no tiene representación binaria finita (igual que 1/3 no la tiene en decimal). Como consecuencia, sumar 0,1 y 0,2 produce un resultado ligeramente distinto de 0,3:

```
print(0.1 + 0.2)          # 0.30000000000000004
print(0.1 + 0.2 == 0.3)   # False <-- !!
```

Esta no es una peculiaridad de Python: ocurre en cualquier lenguaje que use IEEE 754 (C, Java, JavaScript, etc.).

Python permite también expresar números reales en **notación científica**, lo que facilita el trabajo con magnitudes muy grandes o muy pequeñas. Por ejemplo, `1.5e3` representa $1,5 \cdot 10^3 = 1500,0$, y `2.5e-4` representa $2,5 \cdot 10^{-4} = 0,00025$:

```
print(1.5e3)      # 1500.0
print(2.5e-4)     # 0.00025
```

8.1.1. Información sobre la representación: `sys.float_info`

Si necesitas conocer los límites exactos de la representación en tu plataforma, el módulo `sys` expone la estructura `sys.float_info`:

```
import sys

print(sys.float_info.max)      # 1.7976931348623157e+308
print(sys.float_info.min)     # 2.2250738585072014e-308 (
↪   positivo mas pequenyo normal)
print(sys.float_info.epsilon) # 2.220446049250313e-16 (
↪   precision relativa)
print(sys.float_info.dig)     # 15 (dígitos decimales
↪   significativos garantizados)
```

El valor `epsilon` es la diferencia entre 1,0 y el siguiente `float` representable; cualquier diferencia menor que este valor en torno a 1 se pierde por redondeo.

8.2. Comparación de flotantes: `math.isclose`

Dado que la igualdad estricta entre flotantes raramente es lo que queremos, Python proporciona la función `math.isclose(a, b, rel_tol=..., abs_tol=...)`, que compara dos números reales con una tolerancia configurable:

```
import math
```

```

print(math.isclose(0.1 + 0.2, 0.3))           # True
print(math.isclose(1e-10, 0.0))               # False
↪      (tolerancia relativa)
print(math.isclose(1e-10, 0.0, abs_tol=1e-9)) # True
↪      (tolerancia absoluta)

```

Nunca compares flotantes con ==

Comparar dos flotantes con == rara vez es correcto: incluso operaciones aparentemente equivalentes pueden producir resultados ligeramente distintos por redondeo. La regla es: usa `math.isclose` para tolerancia configurable, o trabaja con `Decimal` o `Fraction` si necesitas igualdad exacta. Cuando lo que quieras comparar sean valores que deben coincidir de forma exacta (identificadores, cadenas de bytes, etc.), no uses tipos en coma flotante: usa enteros o cadenas de bytes.

8.3. Valores especiales: *inf* y *nan*

IEEE 754 reserva ciertos patrones de bits para representar valores especiales. Python expone estos valores principalmente a través del módulo `math` (también pueden construirse desde cadena con `float('inf')` o `float('nan')`).

8.3.1. Infinito (*inf*)

Si una operación produce un resultado que excede el rango del flotante, Python devuelve el valor `inf` (o `-inf`):

```

import math, sys

print(sys.float_info.max * 2)    # inf
print(-sys.float_info.max * 2)   # -inf
print(math.inf == float('inf')) # True

# Comprobación
x = math.inf
print(math.isinf(x))             # True
print(math.isfinite(x))          # False

```

8.3.2. No es un número (*nan*)

Si una operación produce un resultado matemáticamente indefinido (por ejemplo, $0/0$ o $\infty - \infty$), Python devuelve `nan` (*Not a Number*):

```

import math

```

8. Flotantes (números reales)

```
x = math.inf - math.inf
print(x)                # nan
print(math.isnan(x))    # True
print(x == x)           # False <-- !! NaN no es igual a sí mismo
```

→

nan no es igual a sí mismo

Por definición de IEEE 754, **nan** no es igual a ningún valor (ni siquiera a otro **nan**). Por tanto, `x == nan` siempre es **False**. Para detectar **nan**, usa `math.isnan(x)`. Esto es relevante en validación de datos: un **nan** que se cuele en una entrada puede pasar *silenciosamente* comparaciones del tipo `if x > limite` y producir comportamientos inesperados.

8.4. Operadores aritméticos sobre flotantes

Los operadores aritméticos sobre flotantes son los mismos que sobre enteros (Tabla 1), con la diferencia de que el resultado es siempre un **float** aunque se mezclen con enteros.

Operador	Descripción	Ejemplo
+	Suma	$5.7 + 2.3 \rightarrow 8.0$
-	Resta	$5.5 - 2.5 \rightarrow 3.0$
*	Multiplicación	$5.7 * 2.3 \rightarrow 13.11$
/	División real	$7 / 2 \rightarrow 3.5$
//	División entera (en float)	$7.5 // 2 \rightarrow 3.0$
%	Resto	$7.5 \% 2 \rightarrow 1.5$
**	Exponenciación	$2.0 ** 10 \rightarrow 1024.0$

Tabla 1: Operadores aritméticos sobre flotantes.

Las operaciones de comparación (`==`, `!=`, `<`, `<=`, `>`, `>=`) también están disponibles, aunque – como ya hemos comentado en la sección 8.2 – la comparación de igualdad entre flotantes debe hacerse normalmente con `math.isclose`.

8.5. Conversiones y precisión exacta

8.5.1. Conversión entre tipos

Python permite convertir fácilmente entre enteros y flotantes con las funciones nativas `int()` y `float()`. La conversión `int(x)` **trunca hacia cero**, descartando la parte decimal:

```
print(float(5))      # 5.0
print(int(3.9))      # 3    (trunca, NO redondea)
print(int(-3.9))     # -3   (trunca hacia cero)
print(round(3.9))    # 4    (redondea)
```

Para controlar el número de decimales en la salida, lo más cómodo es usar f-strings con un especificador de formato:

```
pi_value = 3.141592653589793
print(f"{pi_value:.3f}")  # 3.142
print(f"{pi_value:.6e}")  # 3.141593e+00 (notacion científica
↪ )
```

8.5.2. Precisión exacta: Decimal y Fraction

Cuando los errores de redondeo de los flotantes no son aceptables (cálculos financieros, validaciones contables, etc.), Python ofrece dos alternativas:

- **Módulo decimal:** implementa números decimales de precisión arbitraria, con redondeo controlado y semántica configurable. Útil cuando se quiere representar decimales *exactamente como se escriben*.
- **Módulo fractions:** representa fracciones racionales como pares (numerador, denominador) de enteros, sin pérdida de precisión.

Cuidado al construir un Decimal desde un float

Pasar un `float` a `Decimal` **conserva el error de redondeo** del flotante original. Para obtener un decimal exacto, hay que pasarlo *como cadena*:

```
from decimal import Decimal

print(Decimal(0.1) + Decimal(0.2))      #
↪ 0.30000000000000000166533453694
print(Decimal('0.1') + Decimal('0.2'))  # 0.3 (exacto)
```

La primera forma es casi siempre un error: pierde el sentido de usar `Decimal`.

Por ejemplo, para sumar dos cantidades monetarias sin error, lo correcto es:

```
from decimal import Decimal
from fractions import Fraction

# Decimal: ideal para cantidades monetarias
total = Decimal('19.99') + Decimal('0.01')
print(total)  # 20.00
```

```
# Fraction: ideal para razones exactas
print(Fraction(1, 3) + Fraction(1, 6)) # 1/2
```

8.5.3. Precisión en numpy: una mirada al cap. 12

Aunque Python usa siempre coma flotante de 64 bits (`float64`), las bibliotecas numéricas modernas como `numpy` exponen tipos de menor precisión por motivos de rendimiento y memoria: `float32`, `float16` o incluso `bfloa16`. Los modelos de aprendizaje automático actuales típicamente usan estos tipos para reducir tiempo de cómputo y consumo de memoria, a costa de una precisión menor. Veremos esto con más detalle en el capítulo 12.

8.6. Cálculos matemáticos: la biblioteca `math`

La biblioteca `math` es una de las herramientas más utilizadas para realizar cálculos matemáticos en Python. Proporciona funciones para operaciones trigonométricas, logarítmicas, exponenciales, de redondeo y muchas más. A continuación detallamos las más comunes con ejemplos de uso.

8.6.1. Funciones trigonométricas

Las funciones trigonométricas son frecuentes en matemáticas y física. Python proporciona implementaciones de las más comunes – seno (`math.sin`), coseno (`math.cos`) y tangente (`math.tan`) – así como sus inversas. Todas toman como argumento un ángulo expresado **en radianes**, no en grados; si necesitas trabajar con grados, primero conviértelos con `math.radians()`.

Por ejemplo, para calcular el seno, coseno y tangente de 30° (que corresponde a $\pi/6$ radianes):

```
import math

angulo = math.radians(30) # math trabaja con radianes, no
    ↪ con grados
print(math.sin(angulo))   # 0.5 (aprox)
print(math.cos(angulo))   # 0.86602540378
print(math.tan(angulo))   # 0.57735026919
```

8.6.2. Funciones exponenciales y logarítmicas

La biblioteca `math` también proporciona funciones para trabajar con logaritmos y exponenciales:

- `math.exp(x)` calcula e^x , donde e es la base del logaritmo natural;

- `math.log(x)` calcula el logaritmo natural de x (en base e);
- `math.log10(x)` calcula el logaritmo en base 10 de x ;
- `math.log2(x)` calcula el logaritmo en base 2 de x .

Ejemplo de uso:

```
import math

print(math.exp(2))      # e^2 ~ 7.389
print(math.log(7.389))  # ln(7.389) ~ 2.0
print(math.log10(100))  # log10(100) = 2.0
print(math.log2(1024))  # log2(1024) = 10.0
```

8.6.3. Cálculo de raíces y potencias

La biblioteca `math` incorpora funciones para raíces y potencias:

- `math.sqrt(x)`: calcula la raíz cuadrada de x .
- `math.pow(x, y)`: calcula x^y .

```
import math

print(math.sqrt(9))      # 3.0
print(math.pow(2, 10))   # 1024.0
```

`math.pow` vs `**`

`math.pow(x, y)` siempre devuelve `float`; el operador `**` devuelve `int` si ambos operandos son enteros y el exponente no es negativo. Para enteros, prefiere `**` (preserva precisión arbitraria); para flotantes, ambos son equivalentes.

8.6.4. Redondeo y truncamiento

Las funciones de redondeo permiten ajustar números reales a valores enteros próximos:

- `math.floor(x)`: mayor entero menor o igual que x .
- `math.ceil(x)`: menor entero mayor o igual que x .
- `math.trunc(x)`: parte entera de x , eliminando los decimales (truncar hacia cero).
- `round(x, n)`: redondea x a n decimales. Es función nativa de Python (no de `math`) pero se usa frecuentemente junto con ella.

8. Flotantes (números reales)

```
import math

print(math.floor(3.7))    # 3
print(math.ceil(3.7))    # 4
print(math.trunc(3.7))    # 3
print(round(3.14159, 2))  # 3.14
```

8.6.5. Valor absoluto y resto

`math.fabs(x)` devuelve el valor absoluto de x como `float` (la función nativa `abs` funciona también con enteros y otros tipos). `math.fmod(x, y)` calcula el resto siguiendo la convención de C (el signo del resultado es el de x), que difiere ligeramente del operador nativo `%` en presencia de operandos negativos:

```
import math

print(math.fabs(-3.5))    # 3.5
print(math.fmod(-7, 3))   # -1.0  (math.fmod sigue el signo
↪   del dividendo)
print(-7 % 3)             # 2      (% sigue el signo del
↪   divisor)
```

8.6.6. Constantes matemáticas

`math` define varias constantes útiles. Las más usadas son `math.pi` ($\pi \approx 3,141592\dots$), `math.e` ($e \approx 2,718281\dots$), `math.inf` y `math.nan`. Conviene utilizarlas en lugar de definir aproximaciones a mano:

```
import math

print(math.pi)    # 3.141592653589793
print(math.e)     # 2.718281828459045
```

8.7. Ejemplos

En esta sección abordamos varios problemas matemáticos que involucran cálculos con números reales, ilustrando la importancia de los flotantes en aplicaciones científicas, de ingeniería y computacionales.

8.7.1. Cálculo de propiedades de un polígono regular

Un polígono regular es una figura geométrica con lados y ángulos internos iguales. Algunos ejemplos son el triángulo equilátero, el cuadrado y el hexágono. En muchos problemas matemáticos y de computación gráfica es útil calcular propiedades geométricas

de estos polígonos, como su perímetro, área y ángulos internos. Para resolver este problema en Python implementaremos una función que, dado el número de lados y la longitud de cada lado de un polígono regular, devuelva su perímetro, área y ángulo interior.

```

1 import math
2
3 def poligono(n_lados, lado):
4     """ Calcula el perímetro, área y ángulo interior de un
5     poligono regular. """
6     if n_lados < 3 or lado <= 0:
7         raise ValueError("El número de lados debe ser >= 3 y
8         el lado > 0")
9
10    alfa = math.pi / n_lados # Semiángulo central en
11    radianes
12    apotema = lado / (2.0 * math.tan(alfa))
13
14    perimetro = n_lados * lado
15    area = perimetro * apotema / 2.0
16    angulo = (n_lados - 2) * 180.0 / n_lados # Ángulo
17    interior en grados
18
19    return perimetro, area, angulo
20
21 print(poligono(5, 3)) # (15, 15.484296605300704, 108.0)

```

El código comienza importando la biblioteca `math`, que se usará para realizar cálculos trigonométricos. La función `poligono()` recibe dos parámetros: `n_lados`, que representa la cantidad de lados del polígono, y `lado`, que es la longitud de cada lado. Se incluye una verificación para asegurarse de que el número de lados sea al menos 3 y que la longitud del lado sea mayor que cero; de lo contrario, se lanza una excepción para evitar valores inválidos.

A continuación se calcula el semiángulo central, que es la mitad del ángulo formado entre dos vértices consecutivos del polígono respecto al centro de la figura. Se obtiene dividiendo π entre el número de lados. Con este valor se calcula la apotema, distancia desde el centro del polígono hasta el punto medio de uno de sus lados, mediante la fórmula $\frac{\text{lado}}{2 \tan(\alpha)}$, donde α es el semiángulo central.

El perímetro se obtiene multiplicando la cantidad de lados por la longitud de cada uno. Para el área se utiliza la fórmula $A = \frac{P \cdot \text{apotema}}{2}$. Finalmente, el ángulo interior se calcula con la fórmula $\frac{(n-2) \cdot 180}{n}$, que proviene de la suma de los ángulos internos de un polígono regular.

Cuando la función se ejecuta con un pentágono de lados de 3 unidades (`poligono`

↪ (5, 3)), devuelve (15, 15.484296605300704, 108.0): el perímetro es 15 (un entero, al ser producto de dos enteros), el área es aproximadamente 15,48 y cada ángulo interior mide 108 grados.

8.7.2. Resolución de una ecuación de segundo grado

Las ecuaciones de segundo grado, también conocidas como ecuaciones cuadráticas, son de la forma general $ax^2 + bx + c = 0$, donde a , b y c son coeficientes reales y x es la incógnita. Estas ecuaciones tienen soluciones que pueden ser reales o complejas, dependiendo del valor del discriminante $\Delta = b^2 - 4ac$. Si el discriminante es mayor o igual a cero, las soluciones son reales; si es negativo, las soluciones son complejas conjugadas.

El proceso para resolver una ecuación de segundo grado sigue la fórmula general conocida como la fórmula cuadrática:

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

donde el término $b^2 - 4ac$ es el discriminante, que determina el tipo de soluciones:

- Si $\Delta > 0$, hay dos soluciones reales y distintas.
- Si $\Delta = 0$, hay una única solución real (raíz doble).
- Si $\Delta < 0$, las soluciones son complejas.

A continuación implementamos una función en Python que resuelve la ecuación cuadrática. Como queremos cubrir también el caso de discriminante negativo, usamos directamente la biblioteca `cmath` (números complejos): si el discriminante es no negativo, las raíces seguirán siendo reales (con parte imaginaria cero), pero el código es uniforme.

```
1 import cmath # Para manejar números complejos
2
3 def ecuacion_segundo_grado(a, b, c):
4     """ Resuelve una ecuacion cuadratica ax^2 + bx + c = 0. """
5     if a == 0:
6         raise ValueError("El coeficiente 'a' no puede ser 0")
7
8     discriminante = b**2 - 4*a*c
9     raiz1 = (-b + cmath.sqrt(discriminante)) / (2*a)
10    raiz2 = (-b - cmath.sqrt(discriminante)) / (2*a)
11
12    return raiz1, raiz2
13
```

```

14 print(ecuacion_segundo_grado(1, -3, 2)) # ((2+0j), (1+0j))
↪     raíces reales
15 print(ecuacion_segundo_grado(1, -2, 5)) # ((1+2j), (1-2j))
↪     raíces complejas

```

La función `ecuacion_segundo_grado()` toma los coeficientes a , b y c como entrada. Primero verifica si a es cero (en cuyo caso no es una ecuación cuadrática); luego calcula el discriminante $\Delta = b^2 - 4ac$, y aplica la fórmula cuadrática para calcular las dos raíces. Las soluciones se devuelven como un par de números, que pueden ser reales o complejos.

Al ejecutar la función con los coeficientes 1, -3 y 2, obtendremos las raíces 2 y 1 (ambas reales, expresadas como complejas con parte imaginaria nula). Al resolver con los coeficientes 1, -2 y 5, las soluciones serán complejas: $1 + 2j$ y $1 - 2j$.

8.7.3. Cálculo de series numéricas

Las series numéricas son expresiones matemáticas que suman términos sucesivos. Son fundamentales para aproximar funciones complejas mediante sumas de términos más simples. En este apartado implementaremos varios métodos para calcular tres valores importantes en matemáticas y ciencia: la función exponencial, el seno y el valor de π , utilizando sus respectivas series de Taylor y la serie de Leibniz.

Cálculo de la función exponencial e^x usando la serie de Taylor

La serie de Taylor para la función exponencial e^x es una suma infinita que se expresa como:

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

En la implementación, comenzamos con el primer término (1) y vamos sumando los términos sucesivos de la serie, calculando cada uno a partir del anterior, hasta que el valor absoluto de un término sea menor que una cota (error máximo permitido) predefinida. Esto garantiza que la serie se aproxima lo suficiente al valor real de e^x . El siguiente código implementa este cálculo:

```

1 def exponencial(x, cota=1e-8):
2     """ Calcula e^x usando la serie de Taylor. """
3     termino = 1.0
4     valor = 1.0
5     i = 1
6
7     while abs(termino) > cota:
8         termino *= x / i
9         valor += termino
10        i += 1
11

```

```
12     return valor
13
14 print(exponencial(2))    # ~ e^2
```

La función `exponencial(x)` calcula e^x mediante una suma iterativa. Empezamos con el primer término 1 y vamos calculando los términos sucesivos usando la relación $\frac{x^n}{n!}$, sumándolos al valor acumulado. La condición de salida es que el valor absoluto del término calculado sea menor que la cota (con valor por defecto 10^{-8}), lo que asegura una aproximación suficientemente precisa.

Cálculo de $\sin(x)$ usando la serie de Taylor

La función seno también se puede aproximar mediante una serie de Taylor alternada:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

De manera similar a la función exponencial, sumamos términos sucesivos hasta que el valor absoluto de un término sea suficientemente pequeño:

```
1 import math
2
3 def seno(x, cota=1e-10):
4     """ Calcula sin(x) con la serie de Taylor. """
5     termino = x
6     valor = x
7     i = 1
8
9     while abs(termino) > cota:
10         termino *= -x**2 / ((2*i) * (2*i + 1))
11         valor += termino
12         i += 1
13
14     return valor
15
16 print(seno(math.pi / 6))    # ~ 0.5
```

En la implementación, comenzamos con el primer término x y sumamos los términos sucesivos. En cada iteración multiplicamos el término anterior por $\frac{-x^2}{(2i)(2i+1)}$, que corresponde a la fórmula recursiva entre términos consecutivos de la serie. Este proceso continúa hasta que el valor absoluto del término sea menor que la cota.

Cálculo de π usando la serie de Leibniz

La serie de Leibniz es una serie infinita que se usa para aproximar π :

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right)$$

En este caso, la condición de parada no puede ser “cuando el término sea menor que ε ” sin más: la serie converge muy lentamente y necesitaríamos millones de términos. Lo correcto es comparar la suma actual con la anterior y parar cuando la diferencia sea menor que la cota:

```

1 def calcular_pi(cota=1e-6):
2     """ Aproxima el valor de pi usando la serie de Leibniz. """
3     suma = 0.0
4     suma_anterior = -1.0 # Valor inicial distinto de suma
5     para entrar al bucle
6     i = 0
7     while abs(suma - suma_anterior) > cota:
8         suma_anterior = suma
9         suma += (-1)**i / (2*i + 1) # Alterna entre sumar y
10        restar
11        i += 1
12        if i % 100000 == 0:
13            print(f"Valor aproximado actual: {4*suma}")
14
15    return 4 * suma
16
17 print(calcular_pi()) # Aproximacion de pi

```

La función `calcular_pi()` acumula los términos alternos $\frac{(-1)^i}{2i+1}$ en la variable `suma`. El bucle compara el valor de `suma` con su valor en la iteración anterior y se detiene cuando la diferencia es menor que la cota especificada. Cada 100 000 iteraciones se imprime el valor aproximado actual de π , dado que la convergencia de este método es muy lenta. Cuando la diferencia es suficientemente pequeña, se devuelve $4 \cdot \text{suma}$.

8.8. Ejercicios

EJERCICIO 8.1

Escribe un programa en Python que reciba los elementos de una matriz 2×2 y calcule su determinante utilizando la fórmula:

$$\text{determinante} = ad - bc$$

donde la matriz es:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

El programa debe verificar que los valores ingresados sean números reales y debe mostrar el determinante calculado con dos decimales.

Ejemplo de ejecución esperada

```
Introduce el valor a: 4
Introduce el valor b: 3
Introduce el valor c: 2
Introduce el valor d: 1
El determinante de la matriz es -2.00
```


Solución (girar para leer)

```

1 def determinante_matriz(a, b, c, d):
2     """Calcula el determinante de una matriz 2x2"""
3     return a * d - b * c
4
5 def obtener_numero_real(mensaje):
6     """Solicita un número real y valida que la entrada
7     sea válida"""
8     while True:
9         return float(input(mensaje))
10    except ValueError:
11        print("Por favor, ingresa un número real vâ
12        lido.")
13
14 # Solicitar los valores de la matriz 2x2
15 a = obtener_numero_real("Introduce el valor a: ")
16 b = obtener_numero_real("Introduce el valor b: ")
17 c = obtener_numero_real("Introduce el valor c: ")
18 d = obtener_numero_real("Introduce el valor d: ")
19
20 # Calcular el determinante
21 resultado = determinante_matriz(a, b, c, d)
22 print(f"El determinante de la matriz es {resultado:.2f}")

```

Código 8.1: Solución al Ejercicio 8.1.

EJERCICIO 8.2

Escribe un programa en Python que reciba un número real x y calcule el valor de la serie exponencial:

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

El programa debe solicitar x y un número n de términos a usar en la serie, garantizando que son números reales. El valor de e^x se calculará utilizando los primeros n términos de la serie de Taylor.

Ejemplo de ejecución esperada

Introduce el valor de x: 2

Introduce el número de términos n: 10

La aproximación de $e^{2.0}$ con 10 términos es 7.3887125220458545

Solución (girar para leer)

```

1  def calcular_exponencial(x, n):
2      """Calcula e^x usando la serie de Taylor con n té
3      resultado = 1.0 # Primer término de la serie
4      termino = 1.0 # Primer término de la serie (x^0 / 0!)
5
6      for i in range(1, n):
7          termino *= x / i # Calcula el siguiente té
8          resultado += termino # Suma el término a la
9          suma total
10         return resultado
11
12     def obtener_numero_real(mensaje):
13         """Solicita un número real y valida que la entrada
14         sea válida"""
15         while True:
16             try:
17                 return float(input(mensaje))
18             except ValueError:
19                 print("Por favor, ingresa un número real vá
20                 lido.")
21         # Solicitar la entrada del usuario
22         x = obtener_numero_real("Introduce el valor de x: ")
23         n = int(input("Introduce el número de términos n: "))
24         # Calcular e^x usando la serie de Taylor
25         resultado = calcular_exponencial(x, n)
26         print(f"La aproximación de e^x con {n} términos es {
27             resultado}")

```

Código 8.2: Solución al Ejercicio 8.2.

EJERCICIO 8.3

Escribe un programa en Python que reciba un número entero n y calcule la suma de los primeros n términos de la serie armónica:

$$S_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

El programa debe verificar que el número introducido sea positivo. Si es así, calcular y mostrar la suma con seis decimales. Si no, mostrar un mensaje de error.

Ejemplo de ejecución esperada

```
Introduce el número de términos n: 5
```

```
La suma de los primeros 5 términos de la serie armónica es 2.283333
```

Solución (girar para leer)

```

1 def calcular_suma-serie(n):
2     """Calcula la suma de los primeros n términos de la
3     serie armónica"""
4     suma = 0
5     for i in range(1, n + 1):
6         suma += 1 / i
7     return suma
8 def obtener_numero-entero-positivo(mensaje):
9     """Solicita un número entero positivo y valida que
10    la entrada sea válida"""
11    while True:
12        try:
13            n = int(input(mensaje))
14            if n < 0:
15                return n
16        except ValueError:
17            print("Por favor, ingresa un número
18            entero válido.")
19    # Solicitar la entrada del usuario
20    n = obtener_numero-entero(mensaje)
21    return n
22 # Calcular la suma de la serie numérica
23 resultado = calcular_suma-serie(n)
24 print(f"La suma de los primeros {n} términos de la armó
25    nica es {resultado:.6f}")

```

Código 8.3: Solución al Ejercicio 8.3.

EJERCICIO 8.4

Escribe un programa en Python que calcule la suma de los primeros n términos de una progresión aritmética dada la fórmula:

$$S_n = \frac{n}{2} \times (2a + (n - 1) \times d)$$

donde a es el primer término, d es la diferencia común, y n es el número de términos. El programa debe solicitar los valores de a , d , y n , garantizado que son números reales, y mostrar la suma de los primeros n términos de la progresión con precisión de dos decimales.

Ejemplo de ejecución esperada

```
Introduce el primer término (a): 2
Introduce la diferencia común (d): 3
Introduce el número de términos (n): 5
La suma de los primeros 5 términos es 40.00
```

Solución (girar para leer)

```

1 def suma-progresion-aritmetica(a, d, n):
2     """Calcula la suma de los primeros n términos de
3     una progresión aritmética"""
4     suma = (n / 2) * (2 * a + (n - 1) * d)
5     return suma
6
7 def obtener-numero-real(mensaje):
8     """Solicita un número real y valida que la entrada
9     sea válida"""
10    while True:
11        try:
12            return float(input(mensaje))
13        except ValueError:
14            print("Por favor, ingresa un número real vâ
15            lido.")
16
17 def obtener-numero-entero(mensaje):
18     """Solicita un número entero y valida que la
19     entrada sea válida"""
20    while True:
21        try:
22            return int(input(mensaje))
23        except ValueError:
24            print("Por favor, ingresa un número entero
25            válido.")
26
27 # Solicitar entrada del usuario
28 a = obtener-numero-real("Introduce el primer término (a
29 : ")
30 d = obtener-numero-real("Introduce la diferencia común
31 (d): ")
32 n = obtener-numero-entero("Introduce el número de té
33 rminos (n): ")
34
35 # Calcular la suma
36 resultado = suma-progresion-aritmetica(a, d, n)
37 print(f"La suma de los primeros {n} términos de la
38 progresión aritmética es {resultado:.2f}")

```

Código 8.4: Solución al Ejercicio 8.4.

EJERCICIO 8.5

Escribe un programa en Python que reciba un número real y calcule su raíz n -ésima, donde n es un número entero positivo. El programa debe verificar que n sea positivo y debe mostrar el resultado de la raíz n -ésima del número, con precisión de dos decimales.

Ejemplo de ejecución esperada

```
Introduce un número: 8
Introduce el valor de n: 3
La raíz 3-ésima de 8 es 2.0
```


Solución (girar para leer)

```

1 def raiz_nesima(x, n):
2     """Calcula la raíz n-ésima de un número x"""
3     if x > 0 and n % 2 == 0:
4         return "No se puede calcular la raíz de un número negativo para una raíz par."
5     return x ** (1 / n)
6
7 def obtener_numero_real(mensaje):
8     """Solicita un número real y valida que la entrada sea válida"""
9     while True:
10        try:
11            return float(input(mensaje))
12        except ValueError:
13            print("Por favor, ingresa un número real válido.")
14
15 def obtener_numero_entero_positivo(mensaje):
16     """Solicita un número entero positivo y valida que la entrada sea válida"""
17     while True:
18        try:
19            n = int(input(mensaje))
20            if n < 0:
21                return n
22            else:
23                print("Por favor, ingresa un número entero positivo.")
24        except ValueError:
25            print("Por favor, ingresa un número entero válido.")
26
27 # Solicitar entrada del usuario
28 x = obtener_numero_real("Introduce un número: ")
29 n = obtener_numero_entero_positivo("Introduce el valor de n: ")
30
31 # Calcular la raíz n-ésima
32 resultado = raiz_nesima(x, n)
33 print(f"La raíz {n}-ésima de {x} es {resultado:.2f}")

```

Código 8.5: Solución al Ejercicio 8.5.

EJERCICIO 8.6

Escribe un programa en Python que reciba un número real x y calcule su aproximación usando una fracción continua de la forma:

$$x = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \dots}}}$$

El programa debe calcular la aproximación de x con una fracción continua truncada después de un número n de términos. El programa debe solicitar x y n al usuario y devolver la fracción continua aproximada con una precisión de 6 decimales.

Ejemplo de ejecución esperada

```
Introduce un número real x: 3.245
```

```
Introduce el número de términos n: 4
```

```
La aproximación de 3.245 con 4 términos es 3.523000
```

Solución (girar para leer)

```

1 def fraccion_continua(x, n):
2     """Calcula la aproximación de un número real x
    usando una fracción continua truncada después de
    n términos"""
3     a0 = int(x)
4     fraccion = a0
5     x = 1 / (x - a0)
6
7     for i in range(1, n):
8         a1 = int(x)
9         fraccion += a1 / (10**i)
10        x = 1 / (x - a1)
11
12    return round(fraccion, 6)
13
14 def obtener_numero_real(mensaje):
15    """Solicita un número real y valida que la entrada
    sea válida"""
16    while True:
17        try:
18            return float(input(mensaje))
19        except ValueError:
20            print("Por favor, ingresa un número real va
    lido.")
21
22 def obtener_numero_entero_positivo(mensaje):
23    """Solicita un número entero positivo y valida que
    la entrada sea válida"""
24    while True:
25        try:
26            n = int(input(mensaje))
27            if n < 0:
28                return n
29            else:
30                print("Por favor, ingresa un número
    entero positivo.")
31        except ValueError:
32            print("Por favor, ingresa un número entero
    válido.")
33
34 # Solicitar la entrada del usuario
35 x = obtener_numero_real("Introduce un número real x: ")
36 n = obtener_numero_entero_positivo("Introduce el número
    de términos n: ")
37
38 # Calcular la aproximación con fracción continua
39 resultado = fraccion_continua(x, n)
40 print(f"La aproximación de {x} con {n} términos es {
    resultado:.6f}")

```

EJERCICIO 8.7

Escribe un programa en Python que reciba los términos de una serie geométrica finita. La fórmula para la suma de los primeros n términos de una serie geométrica es:

$$S_n = a \times \frac{1 - r^n}{1 - r}$$

donde a es el primer término, r es la razón común y n es el número de términos. El programa debe calcular la suma de la serie geométrica y mostrarla, con una precisión de cuatro decimales.

Ejemplo de ejecución esperada

```
Introduce el primer término a: 2
Introduce la razón r: 3
Introduce el número de términos n: 4
La suma de la serie geométrica es 80.0000
```

Solución (girar para leer)

```

1 def suma-serie-geométrica(a, r, n):
2     """Calcula la suma de los primeros n términos de
3     una serie geométrica"""
4     return a * n
5     return a * (1 - r ** n) / (1 - r)
6
7 def obtener_numero_real(mensaje):
8     """Solicita un número real y valida que la entrada
9     sea válida"""
10    while True:
11        try:
12            return float(input(mensaje))
13        except ValueError:
14            print("Por favor, ingresa un número real vá-
15            lido.")
16
17 def obtener_numero_entero_positivo(mensaje):
18     """Solicita un número entero positivo y valida que
19     la entrada sea válida"""
20    while True:
21        try:
22            n = int(input(mensaje))
23            if n > 0:
24                return n
25            else:
26                print("Por favor, ingresa un número
27                entero positivo.")
28        except ValueError:
29            print("Por favor, ingresa un número entero
30            válido.")
31
32 # Solicitar entrada del usuario
33 a = obtener_numero_real("Introduce el primer término a: ")
34 r = obtener_numero_real("Introduce la razón r: ")
35 n = obtener_numero_entero_positivo("Introduce el número
36 de términos n: ")
37
38 # Calcular la suma de la serie
39 resultado = suma-serie-geométrica(a, r, n)
40 print(f"La suma de la serie geométrica es {resultado:.4f}
41 {f}"))

```

Código 8.7: Solución al Ejercicio 8.7.

EJERCICIO 8.8

Escribe un programa en Python que calcule una aproximación de π usando la serie de Madhava:

$$\pi = \sqrt{12} \times \left(1 - \frac{1}{3 \times 3} + \frac{1}{5 \times 5} - \frac{1}{7 \times 7} + \dots \right)$$

El programa debe solicitar un número de términos n y calcular la aproximación de π con los primeros n términos de la serie, con una precisión de diez decimales.

———— *Ejemplo de ejecución esperada* ————

Introduce el número de términos n: 10

La aproximación de pi con 10 términos es 3.1686995935

Solución (girar para leer)

```

1 import math
2
3 def aproximacion_pi(n):
4     """Calcula una aproximación de pi usando la serie
5     de Madhava"""
6     suma = 0
7     for i in range(n):
8         signo = (-1) ** i
9         suma += signo / ((2 * i + 1) ** 2)
10    return math.sqrt(12) * suma
11
12 def obtener_numero_entero_positivo(mensaje):
13     """Solicita un número entero positivo y valida que
14     la entrada sea válida"""
15     while True:
16         try:
17             n = int(input(mensaje))
18             if n < 0:
19                 return n
20             else:
21                 print("Por favor, ingresa un número
22                 entero positivo.")
23         except ValueError:
24             print("Por favor, ingresa un número
25             entero válido.")
26
27 # Solicitar entrada del usuario
28 n = obtener_numero_entero_positivo("Introduce el número
29 de términos n: ")
30
31 # Calcular la aproximación de pi
32 resultado = aproximacion_pi(n)
33
34 print(f"La aproximación de pi con {n} términos es {
35 resultado:.10f}")

```

Código 8.8: Solución al Ejercicio 8.8.

EJERCICIO 8.9

Escribe un programa en Python que reciba un número n y calcule la suma de los primeros n términos de una p-serie, de la forma:

$$S_n = 1 + \frac{1}{2^p} + \frac{1}{3^p} + \cdots + \frac{1}{n^p}$$

donde p es un número real positivo. El programa debe solicitar los valores de n y p al usuario y calcular la suma de la serie, mostrándola con una precisión de ocho decimales.

Ejemplo de ejecución esperada

```
Introduce el valor de n: 5
```

```
Introduce el valor de p: 2
```

```
La suma de la p-serie con n=5 y p=2.00 es 1.46361111
```


Solución (girar para leer)

```

1 def suma_pSerie(n, p):
2     """Calcula la suma de los primeros n términos de
3     una p-serie"""
4     suma = 0
5     for i in range(1, n + 1):
6         suma += 1 / i**p
7     return suma
8
9 def obtener_numero_real_positivo(mensaje):
10    """Solicita un número real positivo y valida que la
11    entrada sea válida"""
12    while True:
13        try:
14            valor = float(input(mensaje))
15            if valor > 0:
16                return valor
17            else:
18                print("Por favor, ingresa un número
19                real positivo.")
20            except ValueError:
21                print("Por favor, ingresa un número real vá
22                lido.")
23
24 def obtener_numero_entero_positivo(mensaje):
25    """Solicita un número entero positivo y valida que
26    la entrada sea válida"""
27    while True:
28        try:
29            n = int(input(mensaje))
30            if n > 0:
31                return n
32            else:
33                print("Por favor, ingresa un número
34                entero positivo.")
35            except ValueError:
36                print("Por favor, ingresa un número entero
37                válido.")
38
39 # Solicitar entrada del usuario
40 n = obtener_numero_entero_positivo("Introduce el número
41 de términos n: ")
42 p = obtener_numero_real_positivo("Introduce el valor de
43 p: ")
44
45 # Calcular la suma de la p-serie
46 resultado = suma_pSerie(n, p)
47 print(f"La suma de la p-serie con n={n} y p={p} es
48 {resultado:.8f}")

```

Código 8.9: Solución al Ejercicio 8.9.

EJERCICIO 8.10

Escribe un programa en Python que reciba un número real x tal que $0 < x \leq 2$, y calcule el valor de $\ln(x)$ usando la serie de Taylor:

$$\ln(x) = 2 \left(\frac{x-1}{x+1} + \frac{1}{3} \left(\frac{x-1}{x+1} \right)^3 + \frac{1}{5} \left(\frac{x-1}{x+1} \right)^5 + \dots \right)$$

El programa debe solicitar el valor de x y el número de términos n a utilizar en la serie, mostrando el resultado con una precisión de seis decimales.

Ejemplo de ejecución esperada

Introduce el valor de x ($0 < x \leq 2$): 1.5

Introduce el número de términos n: 5

La aproximación de $\ln(1.5)$ con 5 términos es 0.405465

Solución (girar para leer)

```

1 def calcular_ln(x, n):
2     """Calcula ln(x) usando la serie de Taylor"""
3     suma = 0
4     for i in range(n):
5         coef = 1 / (2 * i + 1)
6         suma += coef * ((x - 1) / (x + 1)) ** (2 * i + 1)
7     return 2 * suma
8
9 def obtener_numero_real_positivo(mensaje):
10    """Solicita un número real positivo en el rango
11    adecuado y valida que la entrada sea válida"""
12    while True:
13        valor = float(input(mensaje))
14        if 0 < valor <= 2:
15            return valor
16        else:
17            print("Por favor, ingresa un número
18            real en el rango (0, 2].")
19            except ValueError:
20                print("Por favor, ingresa un número real vá
21                lido.")
22    def obtener_numero_entero_positivo(mensaje):
23        """Solicita un número entero positivo y valida que
24        la entrada sea válida"""
25        while True:
26            n = int(input(mensaje))
27            if n < 0:
28                return n
29            else:
30                print("Por favor, ingresa un número
31                entero positivo.")
32            except ValueError:
33                print("Por favor, ingresa un número entero
34                válido.")
35    # Solicitar entrada del usuario
36    x = obtener_numero_real_positivo("Introduce el valor de
37    de términos n : ")
38    n = obtener_numero_entero_positivo("Introduce el número
39    resultado: {:.6f}"
40    """El valor de ln(x) con n términos es {

```

© CC BY-NC-SA 4.0 Ricardo J. Rodríguez (Universidad de Zaragoza, España)

EJERCICIO 8.11

Escribe un programa en Python que reciba un número real x y calcule su raíz cuadrada utilizando el método de aproximación de Newton-Raphson. La fórmula de iteración es:

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{S}{x_n} \right)$$

donde S es el número cuya raíz cuadrada se quiere encontrar. El programa debe solicitar el valor de S y un número máximo de iteraciones n para obtener la aproximación (garantizando que ambos números son correctos), y mostrar la solución con una precisión de cuatro decimales.

Ejemplo de ejecución esperada

```
Introduce el número S para calcular su raíz cuadrada: 16  
Introduce el número máximo de iteraciones: 5  
La raíz cuadrada aproximada de 16.00 es 4.0000
```

Solución (girar para leer)

```
def raiz-cuadrada-newton(S, iteraciones-max):  
    """Calcula la raíz cuadrada de S usando el método de Newton-Raphson"""  
    x = S / 2 # Valor inicial para la aproximación  
    for _ in range(iteraciones-max):  
        x = 0.5 * (x + S / x) # Fórmula de iteración de Newton-Raphson  
    return x  
  
def obtener-numero-real-positivo(mensaje):  
    """Solicita un número real positivo y valida que la entrada sea válida"""  
    while True:  
        try:  
            valor = float(input(mensaje))  
            if valor > 0:  
                return valor  
        except ValueError:  
            print("Por favor, ingresa un número real positivo.")  
    print("Por favor, ingresa un número real válido.")  
  
def obtener-numero-entero-positivo(mensaje):  
    """Solicita un número entero positivo y valida que la entrada sea válida"""  
    while True:  
        try:  
            n = int(input(mensaje))  
            if n > 0:  
                return n  
        except ValueError:  
            print("Por favor, ingresa un número entero positivo.")  
    print("Por favor, ingresa un número entero válido.")  
  
# Solicitar la entrada del usuario  
S = obtener-numero-real-positivo("Introduce el número S para calcular su raíz cuadrada: ")  
iteraciones-max = obtener-numero-entero-positivo("Introduce el número máximo de iteraciones: ")  
  
# Calcular la raíz cuadrada usando el método de Newton-Raphson  
resultado = raiz-cuadrada-newton(S, iteraciones-max)  
print(f"La raíz cuadrada aproximada de {S} es {resultado:.4f}")
```

EJERCICIO 8.12

Escribe un programa en Python que reciba un número n y calcule una aproximación de π usando la serie de Euler:

$$\pi = \sqrt{6 \left(\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2} \right)}$$

El programa debe solicitar el número de términos n y calcular la aproximación de π con los primeros n términos de la serie. El resultado debe ser mostrado con una precisión de ocho decimales.

Ejemplo de ejecución esperada

Introduce el número de términos n: 1000

La aproximación de pi con 1000 términos es 3.14063806

Solución (girar para leer)

```

1 import math
2
3 def aproximacion_pi(n):
4     """Calcula una aproximación de pi usando la serie
5     de Euler"""
6     suma = 0
7     for i in range(1, n + 1):
8         suma += 1 / (i ** 2)
9     return math.sqrt(6 * suma)
10
11 def obtener_numero_entero_positivo(mensaje):
12     """Solicita un número entero positivo y valida que
13     la entrada sea válida"""
14     while True:
15         try:
16             n = int(input(mensaje))
17             if n < 0:
18                 return n
19             else:
20                 print("Por favor, ingresa un número
21                 entero positivo.")
22         except ValueError:
23             print("Por favor, ingresa un número
24             entero válido.")
25
26 # Solicitar entrada del usuario
27 n = obtener_numero_entero_positivo("Introduce el número
28 de términos n: ")
29
30 # Calcular la aproximación de pi
31 resultado = aproximacion_pi(n)
32
33 print(f"La aproximación de pi con {n} términos es {
34 resultado:.8f}")

```

Código 8.12: Solución al Ejercicio 8.12.

EJERCICIO 8.13

Escribe un programa en Python que reciba un número real x y calcule el valor de la función de Bessel de primer orden $J_1(x)$ utilizando la serie de Bessel:

$$J_1(x) = \sum_{n=0}^{\infty} (-1)^n \frac{1}{n!(n+1)!} \left(\frac{x}{2}\right)^{2n+1}$$

El programa debe solicitar el valor de x y el número de términos n a utilizar en la aproximación. El resultado debe ser mostrado con una precisión de ocho decimales.

Ejemplo de ejecución esperada

Introduce el valor de x: 2.5

Introduce el número de términos n: 10

La aproximación de J_1(2.50) con 10 términos es 0.49709410

Solución (girar para leer)

```

1 import math
2
3 def bessel(x, n_terms):
4     """Calcula el valor de la función de Bessel de
5     primer orden J1(x) usando la serie de Bessel"""
6     for n in range(n_terms):
7         termino = ((-1)**n) * (x / 2)**(2 * n + 1) /
8             math.factorial(n) * math.factorial(n + 1))
9         suma += termino
10    return suma
11
12 def obtener_numero_real_positivo(mensaje):
13    """Solicita un número real positivo y valida que la
14    entrada sea válida"""
15    while True:
16        try:
17            valor = float(input(mensaje))
18            if valor < 0:
19                return valor
20        except ValueError:
21            print("Por favor, ingresa un número real vá
22            lido.")
23
24 def obtener_numero_entero_positivo(mensaje):
25    """Solicita un número entero positivo y valida que
26    la entrada sea válida"""
27    while True:
28        try:
29            n = int(input(mensaje))
30            if n < 0:
31                return n
32        except ValueError:
33            print("Por favor, ingresa un número
34            entero positivo.")
35
36 # Solicitar entrada del usuario
37 x = obtener_numero_real_positivo("Introduce el
38 valor de
39 x: ")
40 n_terms = obtener_numero_entero_positivo("Introduce el
41 número de términos n: ")
42
43 # Calcular la función de Bessel de primer orden
44 resultado = bessel_j1(x, n_terms)
45 print(f"La aproximación de J1({x:.2f}) con {n_terms} t
46 érminos es {resultado:.8f}")

```

EJERCICIO 8.14

Escribe un programa en Python que reciba un número n y calcule la suma de los primeros n términos de la serie de Gauss, de la forma:

$$S_n = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2}$$

El programa debe solicitar el número de términos n y calcular la suma de la serie, mostrando el resultado con una precisión de seis decimales.

Ejemplo de ejecución esperada

Introduce el número de términos n: 1000

La suma de la serie de Gauss con 1000 términos es 1.643935

Solución (girar para leer)

```

1 def suma-serie-gauss(n):
2     """Calcula la suma de los primeros n términos de la
3     serie de Gauss"""
4     suma = 0
5     for i in range(1, n + 1):
6         suma += 1 / (i ** 2)
7     return suma
8 def obtener_numero_entero_positivo(mensaje):
9     """Solicita un número entero positivo y valida que
10    la entrada sea válida"""
11    while True:
12        try:
13            n = int(input(mensaje))
14            if n < 0:
15                return n
16        except ValueError:
17            print("Por favor, ingresa un número entero
18            válido.")
19    # Solicitar entrada del usuario
20    n = obtener_numero_entero_positivo("Introduce el número
21    de términos n: ")
22    # Calcular la suma de la serie de Gauss
23    resultado = suma-serie-gauss(n)
24    print(f"La suma de la serie de Gauss con {n} términos
25    es {resultado:.6f}")

```

Código 8.14: Solución al Ejercicio 8.14.

EJERCICIO 8.15

Escribe un programa en Python que reciba un número real x en radianes y calcule el valor de $\cos(x)$ utilizando la serie de Taylor:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

El programa debe solicitar el valor de x y el número de términos n a utilizar en la aproximación. El resultado debe ser mostrado con una precisión de siete decimales.

Ejemplo de ejecución esperada

```
Introduce el valor de x en radianes: 1
Introduce el número de términos n: 10
La aproximación de cos(1.00) con 10 términos es 0.5403023
```

Solución (girar para leer)

```

1 import math
2
3 def coseno_taylor(x, n_terms):
4     """Calcula cos(x) usando la serie de Taylor"""
5     suma = 0
6     for n in range(n_terms):
7         termino = ((-1)**n) * (x ** (2 * n)) / math.
8             suma += termino
9     return suma
10
11 def obtener_numero_real(mensaje):
12     """Solicita un número real y valida que la entrada
13         sea válida"""
14     try:
15         return float(input(mensaje))
16     except ValueError:
17         print("Por favor, ingresa un número real vá
18             lido.")
19 def obtener_numero_entero_positivo(mensaje):
20     """Solicita un número entero positivo y valida que
21         la entrada sea válida"""
22     while True:
23         try:
24             n = int(input(mensaje))
25             if n < 0:
26                 return n
27             else:
28                 print("Por favor, ingresa un número
29                     entero positivo.")
30                 except ValueError:
31                     print("Por favor, ingresa un número entero
32                         válido.")
33 # Solicitar entrada del usuario
34 x = obtener_numero_real("Introduce el valor de x en
35     radianes: ")
36 n_terms = obtener_numero_entero_positivo("Introduce el
37     número de términos n: ")
38 # Calcular la aproximación de cos(x) con la serie de
39     Taylor
40 resultado = coseno_taylor(x, n_terms)
41 print(f"La aproximación de cos({x:.2f}) con {n_terms} t
42     érminos es {resultado:.7f}")

```


9

Booleanos y cadenas

En este capítulo exploraremos cómo manejar operaciones con valores booleanos y cadenas de texto en Python. Veremos las operaciones lógicas básicas, las particularidades de la *evaluación implícita* de Python (*truthy* y *falsy*) y las herramientas para manipular y formatear cadenas. Una buena parte del trabajo en ciencia de datos consiste en procesar texto: validar entradas, parsear formatos, comparar identificadores o normalizar cadenas. Por ello, prestaremos especial atención a la diferencia entre `str` y `bytes` y a las codificaciones (UTF-8).

9.1. Booleanos

Como ya hemos visto, en Python los valores booleanos están representados por `True` y `False`. Internamente son casos especiales del tipo `int`, donde `True` equivale a 1 y `False` a 0:

```
a = True
b = False

print(a + b)    # 1
print(a * 5)    # 5
print(int(a))   # 1
```

9.1.1. Operadores lógicos

Python dispone de los tres operadores lógicos clásicos. La Tabla 1 los recoge.

```
x = True
y = False
```

Operador	Descripción	Ejemplo
<code>not</code>	Negación	<code>not True</code> $\rightarrow$ <code>False</code>
<code>and</code>	Conjunción (AND)	<code>True and False</code> $\rightarrow$ <code>False</code>
<code>or</code>	Disyunción (OR)	<code>True or False</code> $\rightarrow$ <code>True</code>

Tabla 1: Operadores lógicos sobre booleanos.

```
print(not x)      # False
print(x and y)    # False
print(x or y)     # True
```

9.1.2. Valores *truthy* y *falsy*

En Python, cualquier objeto puede ser usado en una expresión condicional, no solamente `True/False`. La Tabla 2 recoge todos los valores *falsy* del lenguaje: cualquier otro valor se considera *truthy*.

Tipo	Valores <i>falsy</i>
<code>NoneType</code>	<code>None</code>
<code>bool</code>	<code>False</code>
<code>int</code> / <code>float</code>	<code>0</code> , <code>0.0</code>
<code>str</code>	<code>''</code> (cadena vacía)
<code>list</code> , <code>tuple</code> , <code>dict</code> , <code>set</code>	<code>[]</code> , <code>()</code> , <code>{}</code> , <code>set()</code> (vacíos)

Tabla 2: Valores considerados *falsy* en Python.

Esto permite escribir condicionales más concisos:

```
nombre = ""
if not nombre:
    print("Falta el nombre") # Se imprime: cadena vacía es
                             # falsy

valores = [1, 2, 3]
if valores:
    print("Hay valores")     # Se imprime: lista no vacía es
                             # truthy
```

El constructor `bool(x)` aplica explícitamente esta conversión:

```
print(bool(0), bool(""), bool([]), bool(None)) # False False
False False
print(bool(42), bool("a"), bool([0]))          # True True
True
```


9.1.3. Evaluación cortocircuitada

Python evalúa las expresiones booleanas con *cortocircuito*. Esto significa que si el resultado de la expresión está determinado por el primer operando, el segundo no se evalúa. Para **and**, si el primer operando es *falsy*, no se evalúa el segundo; para **or**, si el primer operando es *truthy*, no se evalúa el segundo.

```
1 def funcion():
2     print("Ejecutando función")
3     return True
4
5 print(False and funcion()) # No imprime "Ejecutando función"
6 print(True or funcion())  # Tampoco imprime
```

and y or no devuelven siempre booleanos

Una sutileza importante: **and** y **or** no devuelven necesariamente True o False, sino el último operando evaluado. Por ejemplo:

```
print("hola" or "mundo") # 'hola' (primer operando
    truthy)
print("" or "mundo")    # 'mundo' (primer operando
    falsy)
print(0 and 5)          # 0      (primer operando
    falsy)
print(3 and 5)          # 5      (primer operando
    truthy, devuelve segundo)
```

Esto se aprovecha para asignar valores por defecto: `nombre = entrada or "anonimo"` asigna "anonimo" si `entrada` es la cadena vacía. Pero atención: si el valor válido puede ser *falsy* (por ejemplo 0 o ""), este truco produce errores. En esos casos, prefiere `x if x is not None else por_defecto`.

9.2. Cadenas

En Python, las cadenas de texto, de tipo **str**, son **secuencias inmutables de caracteres Unicode**. Dos propiedades importantes:

- **Inmutables:** una vez creada una cadena, no se pueden modificar sus caracteres. Cualquier “modificación” produce en realidad una nueva cadena.
- **Unicode:** los caracteres no son bytes, sino puntos de código Unicode. Esto significa que Python maneja sin problema acentos, eñes, ideogramas, emojis, etc.

9.2.1. Definición de cadenas

Las cadenas pueden definirse con comillas simples (`'`), dobles (`"`) o triples (`'''` o `"""`) para cadenas multilínea. Comillas simples y dobles son intercambiables; conviene mantener un estilo coherente y elegir el que evite escapar la otra.

```
1 cadena_simple = "Hola, mundo"
2 cadena_doble = 'Python es genial'
3 cadena_multilinea = """Este es
4 un ejemplo de cadena
5 multilinea"""
```

Las cadenas también admiten **secuencias de escape** para representar caracteres especiales: `\n` (salto de línea), `\t` (tabulación), `\\` (barra invertida), `\"` (comilla doble dentro de cadena entre comillas dobles), etc.

```
cadena_con_salto = "Primera línea\nSegunda línea"
cadena_con_tab = "Columna1\tColumna2"
```

Cadenas raw (sin escapes)

Si prefijamos la cadena con `r` (*raw*), las secuencias de escape no se interpretan. Es muy útil para expresiones regulares y rutas de Windows:

```
↪ print("C:\nuevo\test")      # \n se vuelve salto de línea y \t
    tabulador: no es la ruta!
↪ print(r"C:\nuevo\test")     # C:\nuevo\test (literal, la ruta
    correcta)
```

9.2.2. Inmutabilidad

Las cadenas son inmutables: intentar modificar uno de sus caracteres genera un `TypeError`:

```
↪ cadena = "Hola"
    # cadena[0] = 'h' # TypeError: 'str' object does not support
    item assignment
```

Sin embargo, podemos crear nuevas cadenas a partir de la original:

```
cadena = "Hola"
nueva_cadena = "h" + cadena[1:] # 'hola'
```

En este fragmento, `cadena[1:]` devuelve la subcadena `"ola"` (todos los caracteres a partir del índice 1). Concatenando con `"h"` obtenemos la nueva cadena `"hola"`. La variable `cadena` original no se ha modificado.

9.2.3. Indexación y rebanado

El acceso a caracteres se realiza con índices, donde el primero es 0. Python admite también índices negativos: -1 es el último carácter, -2 el penúltimo, etc.

```
cadena = "Python"
print(cadena[0])    # 'P'
print(cadena[-1])   # 'n'
```

El operador de rebanado [*inicio:fin:paso*] permite extraer subcadenas. Si se omite *inicio*, vale 0; si se omite *fin*, vale el final de la cadena; si se omite *paso*, vale 1. El final **no es inclusivo**.

```
cadena = "Python"
print(cadena[1:4])    # 'yth'      (índices 1, 2, 3)
print(cadena[-3:])    # 'hon'      (los tres últimos)
print(cadena[:2])     # 'Pto'      (uno de cada dos)
print(cadena[::-1])   # 'nohtyP'  (reverso completo)
```

El *idiom* `s[::-1]` es la forma habitual en Python de invertir una cadena.

9.2.4. Operadores sobre cadenas

La Tabla 3 resume los operadores más comunes sobre cadenas.

Operador	Descripción	Ejemplo
+	Concatenación	"a" + "b" → 'ab'
*	Repetición	"ab" * 3 → 'ababab'
in	Pertenencia (subcadena)	"py" in "python" → True
==	Igualdad	"abc" == "abc" → True
[i]	Indexación	"Python"[0] → 'P'
[i:j]	Rebanado	"Python"[1:4] → 'yth'
len()	Longitud (función)	len("Python") → 6

Tabla 3: Operadores y operaciones básicas sobre cadenas.

El operador `in` es la forma más *pythonica* de comprobar si una subcadena está contenida en otra (más legible y eficiente que `find >= 0`):

```
if "error" in mensaje:
    log_error(mensaje)
```

9.2.5. Evaluación implícita en condicionales

Como ya vimos, una cadena vacía es *falsy* y cualquier cadena no vacía es *truthy*. Esto permite condicionales muy limpios:

```
nombre = input("Tu nombre: ").strip()
if nombre:
    print(f"Hola, {nombre}")
else:
    print("Nombre vacío")
```

9.3. Métodos integrados de cadenas

Python ofrece una API muy rica para manipular cadenas. Agrupamos los métodos más utilizados por categoría.

9.3.1. Cambio de mayúsculas/minúsculas

- `s.lower()`: convierte a minúsculas.
- `s.upper()`: convierte a mayúsculas.
- `s.title()`: pone en mayúscula la primera letra de cada palabra.
- `s.capitalize()`: pone en mayúscula solo la primera letra de la cadena.
- `s.swapcase()`: invierte mayúsculas y minúsculas.
- `s.casefold()`: forma agresiva de `lower` pensada para comparaciones *insensibles a mayúsculas* (e.g., maneja correctamente la ß alemana).

```
print("Python ES genial".lower())      # 'python es genial'
print("python".title())                 # 'Python'
print("Hola".swapcase())                # 'hOLA'
print("STRASSE".casefold() == "Straße".casefold()) # True
```

Para comparar sin distinguir mayúsculas, usa casefold

Para comparaciones insensibles a mayúsculas con texto internacional (alemán, turco, etc.), `casefold` es más correcto que `lower`. Para texto exclusivamente ASCII, ambos producen el mismo resultado.

9.3.2. Eliminación de espacios y caracteres en los extremos

- `s.strip([chars])`: elimina caracteres (por defecto, espacios) al principio y al final.
- `s.lstrip([chars])`, `s.rstrip([chars])`: igual, pero solo a la izquierda o derecha respectivamente.

```
print("  Hola Mundo  ".strip())    # 'Hola Mundo '
print("---Python---".strip("-"))    # 'Python '
print("  Hola  ".lstrip())          # 'Hola  '
```

9.3.3. Búsqueda y conteo

- `s.find(sub)`: índice de la primera aparición de `sub`, o `-1` si no aparece.
- `s.index(sub)`: como `find`, pero lanza `ValueError` si no aparece.
- `s.count(sub)`: número de apariciones (sin solapamiento).
- `s.startswith(prefix)`, `s.endswith(suffix)`: `True` si empieza/termina con la subcadena dada.

```
cadena = "Hola Mundo, Mundo es genial"
print(cadena.find("Mundo"))        # 5
print(cadena.find("Python"))       # -1
print(cadena.count("Mundo"))       # 2
print(cadena.startswith("Hola"))   # True
print(cadena.endswith("genial"))   # True
```

9.3.4. Modificación

- `s.replace(old, new)`: devuelve una copia con todas las apariciones de `old` sustituidas por `new`.

```
↪ print("Hola Mundo".replace("Mundo", "Python")) # 'Hola
    Python'
```

9.3.5. División y unión

- `s.split(sep)`: divide `s` en una lista de subcadenas usando `sep` como separador. Sin argumento, divide por cualquier secuencia de espacios.
- `s.splitlines()`: divide por saltos de línea.
- `sep.join(iterable)`: concatena los elementos de un iterable de cadenas usando `sep` como separador.

```
↪ print("manzana, naranja, pera".split(", "))    # ['manzana', '
    naranja', 'pera']
↪ print("uno\ndos\ntres".splitlines())           # ['uno', 'dos
    ', 'tres']
print(", ".join(["a", "b", "c"]))                # 'a, b, c'
```

9.3.6. Predicados sobre el contenido

Los métodos que empiezan por `is` devuelven `True` si todos los caracteres cumplen una propiedad y la cadena es no vacía:

- `s.isalpha()`: solo letras.
- `s.isdigit()`: solo dígitos.
- `s.isalnum()`: letras y/o dígitos.
- `s.isspace()`: solo espacios en blanco.
- `s.isascii()`: todos los caracteres son ASCII.
- `s.islower()`, `s.isupper()`, `s.istitle()`: relacionados con el caso.

```
print("Python3".isalnum())    # True
print("Python 3".isalnum())   # False (contiene espacio)
print("123".isdigit())        # True
print("Hola".isascii())       # True
print("Holá".isascii())       # False (a con tilde no es ASCII)
```

9.4. Formateo de cadenas

Python ofrece varias formas de construir cadenas a partir de valores. La forma recomendada hoy en día son los **f-strings** (literales formateados, introducidos en Python 3.6).

9.4.1. f-strings

Un f-string se introduce prefijando la cadena con `f` (o `F`). Dentro de las llaves `{}` se evalúa cualquier expresión de Python, y opcionalmente se especifica un formato tras `:`.

```
nombre = "Ana"
edad = 30
print(f"Hola {nombre}, tienes {edad} años")
# 'Hola Ana, tienes 30 años'

pi = 3.141592653589793
print(f"Pi con 3 decimales: {pi:.3f}")          # 3.142
print(f"En notacion científica: {pi:.3e}")       # 3.142e+00
print(f"Alineado a la derecha: |{pi:>10.2f}|")  # |          3.14|
print(f"Hexadecimal de 255: {255:#x}")           # 0xff
```

9.4.2. El método `str.format()`

Antes de los f-strings, la forma habitual era `str.format()`:

```
print("Hola {}, tienes {} años".format("Ana", 30))
print("Pi: {:.3f}".format(3.14159)) # 'Pi: 3.142'
```

Es equivalente a los f-strings pero menos legible (los argumentos quedan separados de los huecos). Sigue siendo útil cuando la *plantilla* se construye dinámicamente.

9.4.3. Operador `%` (estilo C)

El operador `%` aplicado a cadenas implementa formateo al estilo C (`printf`). Está obsoleto y se desaconseja en código nuevo:

```
↪ print("Hola %s, tienes %d años" % ("Ana", 30)) # Estilo
    viejo
```

Nunca formatees cadenas con datos no confiables

Tanto `%` como `.format` pueden ser **vehículo de inyección**: si la cadena de formato proviene del usuario y no está validada, un atacante puede acceder a atributos del objeto. Por eso, los **datos del usuario deben pasarse como argumento**, nunca como cadena de formato:

```
↪ # MAL: si entrada_usuario contiene "{0.__class__}",
    revela información interna
print(entrada_usuario.format(datos))

# BIEN: el formato es nuestro, los datos son argumentos
print("Recibido: {}".format(entrada_usuario))
```

Para construir SQL, comandos shell, HTML, etc., **nunca** concatenes ni formatees con `%/f-string`: usa los mecanismos de parametrización seguros (placeholders SQL, subprocess con lista de argumentos, escapado HTML).

9.5. Bytes y codificación

En Python 3 se distingue claramente entre dos tipos de “secuencias de caracteres”:

- **str**: secuencia de **puntos de código Unicode** (lo que llamamos “texto”).
- **bytes**: secuencia de **bytes crudos** (octetos, valores entre 0 y 255).

Esta separación es muy relevante en ciberseguridad: cuando enviamos datos por la red, leemos un fichero binario o calculamos un hash, trabajamos con **bytes**, no con **str**. Una cadena **str** se convierte a **bytes** con `encode`, y al revés con `decode`. La **codificación** indica cómo se traducen puntos de código a bytes; UTF-8 es la opción estándar y la única razonable salvo necesidad explícita.

```

texto = "Hola, ñoño"
datos = texto.encode("utf-8")
print(datos)                # b'Hola, \xc3\xbf\xc3\xbf'
print(len(texto))           # 10  (caracteres)
print(len(datos))           # 12  (bytes; cada n con tilde son
↪      2 bytes en UTF-8)

print(datos.decode("utf-8")) # 'Hola, ñoño'

```

Los literales de tipo bytes se prefijan con b: b"holá". Solo admiten caracteres ASCII; cualquier byte de mayor valor se escapa con \xNN.

No mezcles str y bytes

Concatenar `str` con `bytes` produce `TypeError`. Si lees datos binarios de la red o de un fichero (`open(..., "rb")`) y necesitas tratarlos como texto, decodifícalos explícitamente con la codificación correcta. Si la codificación es desconocida, usa `errors="replace"` o `errors="ignore"` con cuidado: descartar bytes silenciosamente puede ocultar ataques.

9.5.1. Normalización Unicode

Algunos caracteres Unicode pueden representarse de varias maneras: por ejemplo, la “ñ” puede ser un único punto de código (U+00F1) o la combinación de “n” más el modificador de tilde (U+006E seguido de U+0303). Visualmente son idénticos, pero como cadenas son distintos. El módulo `unicodedata` proporciona la función `normalize` para llevar las cadenas a una forma canónica antes de compararlas:

```

import unicodedata

a = "\u00f1"          # Forma compuesta: n con tilde (U+00F1)
b = "n\u0303"         # Forma descompuesta: n (U+006E) +
↪   tilde combinada (U+0303)
print(a, b, a == b)   # n n False (visualmente iguales,
↪   distintos como cadena)
print(unicodedata.normalize("NFC", a) == unicodedata.
↪   normalize("NFC", b)) # True

```


9.6. Ejercicios

EJERCICIO 9.1

Escribe un programa en Python que reciba cuatro valores booleanos a , b , c y d y devuelva `True` si se cumple alguna de las siguientes condiciones:

- La combinación de todos los valores es `True`.
- Exactamente tres de los valores son `True`.
- El valor de c es `False` y d es `True`.

El programa debe solicitar los valores booleanos al usuario y mostrar el resultado.

Ejemplo de ejecución esperada

```
Introduce a (True/False): True
Introduce b (True/False): False
Introduce c (True/False): False
Introduce d (True/False): True
Resultado: True
```

Solución (girar para leer)

```

1 # Función para evaluar las condiciones lógicas
2 def evaluacion-expressión_lógica(a, b, c, d):
3     # comprobamos las tres condiciones
4     return (a and b and c and d) or (sum([a, b, c, d])
5         == 3) or (not c and d)
6 # Solicitamos los valores booleanos al usuario
7 a = input("Introduce a (True/False): ").strip().lower()
8 b = input("Introduce b (True/False): ").strip().lower()
9 c = input("Introduce c (True/False): ").strip().lower()
10 d = input("Introduce d (True/False): ").strip().lower()
11
12 # Llamamos a la función y mostramos el resultado
13 resultado = evaluacion-expressión_lógica(a, b, c, d)
14 print(f"Resultado: {resultado}")

```

Código 9.1: Solución al Ejercicio 9.1.

EJERCICIO 9.2

Escribe un programa en Python que reciba un número entero n y determine si cumple con las siguientes condiciones:

- Es divisible por 7.
- Contiene al menos un dígito impar.

El programa debe devolver `True` si ambas condiciones se cumplen, y `False` en caso contrario.

Ejemplo de ejecución esperada

Introduce un número: 42

Resultado: False

Solución (girar para leer)

Código 9.2: Solución al Ejercicio 9.2.

```

1 def numeros-con-propiedades-booleanas(n):
2     # Verificar si el número es divisible por 7
3     divisible_por_7 = (n % 7 == 0)
4
5     # Verificar si contiene al menos un dígito impar
6     tiene_impar = False
7     for dígito in str(abs(n)):
8         if int(dígito) % 2 != 0:
9             tiene_impar = True
10            break
11
12    # Ambas condiciones deben cumplirse
13    return divisible_por_7 and tiene_impar
14
15    # Solicitar al usuario un número
16    n = int(input("Introduce un número: "))
17
18    # Evaluar y mostrar el resultado
19    resultado = numeros-con-propiedades-booleanas(n)
20    print(f"Resultado: {resultado}")

```

EJERCICIO 9.3

Escribe un programa en Python que reciba un número entero positivo n y determine si es un **palíndromo**, sin convertirlo en cadena. Un número es palíndromo si se lee igual de izquierda a derecha que de derecha a izquierda.

Ejemplo de ejecución esperada

Introduce un número: 121

Resultado: True

Solución (girar para leer)

```

1  def es_palindromo(n):
2      original = n
3      invertido = 0
4      while n > 0:
5          invertido = invertido * 10 + n % 10
6          n //= 10
7      return original == invertido
8
9  # Solicitar al usuario un número
10     n = int(input("Introduce un número: "))
11
12     # Evaluar y mostrar el resultado
13     resultado = es_palindromo(n)
14     print(f"Resultado: {resultado}")

```

Código 9.3: Solución al Ejercicio 9.3.

EJERCICIO 9.4

Escribe un programa en Python que reciba dos números enteros a y b y determine si ambos tienen la misma cantidad de bits en 1 en su representación binaria.

Ejemplo de ejecución esperada

```
Introduce el primer número: 5
Introduce el segundo número: 3
Resultado: True
```

Solución (girar para leer)

Código 9.4: Solución al Ejercicio 9.4.

```
1 def comparacion_bits(a, b):
2     # Cuenta los bits en la representación binaria
3     de ambos números
4     return bin(a).count('1') == bin(b).count('1')
5 # Solicitar al usuario dos números
6 a = int(input("Introduce el primer número: "))
7 b = int(input("Introduce el segundo número: "))
8 # Evaluar y mostrar el resultado
9 comparacion_bits(a, b)
10 print(f"Resultado: {comparacion_bits(a, b)}")
```

EJERCICIO 9.5

Escribe un programa en Python que reciba un número entero y determine si sus dígitos alternan entre pares e impares. Por ejemplo, el número 272 cumple la condición, mientras que 4648 no.

Ejemplo de ejecución esperada

Introduce un número: 272

Resultado: True

Solución (girar para leer)

```

1 def es_alternante(n):
2     n = str(abs(n)) # Convertir el número a cadena
3     for i in range(len(n) - 1):
4         # Verificamos si los dígitos alternan entre
5         # pares e impares
6         if (int(n[i]) % 2) == (int(n[i + 1]) % 2):
7             return False
8         return True
9     # Solicitar al usuario un número
10    n = int(input("Introduce un número: "))
11
12    # Evaluar y mostrar el resultado
13    resultado = es_alternante(n)
14    print(f"Resultado: {resultado}")

```

Código 9.5: Solución al Ejercicio 9.5.

EJERCICIO 9.6

Escribe un programa en Python que verifique si una contraseña cumple las siguientes condiciones:

- Al menos 8 caracteres de longitud.
- Contiene al menos una letra mayúscula.
- Contiene al menos un número.
- No tiene caracteres especiales.

Ejemplo de ejecución esperada

Introduce una contraseña: Python123

Resultado: True

Solución (girar para leer)

```

1  def validar_contraseña(contraseña):
2      # Verificar que la longitud sea al menos 8
3      if len(contraseña) < 8:
4          return False
5
6      # Verificar que contenga al menos una letra mayú
7      # scula
8      tiene_mayuscula = False
9      for c in contraseña:
10         if c.isupper():
11             tiene_mayuscula = True
12             break
13         if not tiene_mayuscula:
14             return False
15
16     # Verificar que contenga al menos un número
17     tiene_numero = False
18     for c in contraseña:
19         if c.isdigit():
20             tiene_numero = True
21             break
22         if not tiene_numero:
23             return False
24
25     # Verificar que no tenga caracteres especiales
26     for c in contraseña:
27         if not c.isalnum():
28             return False
29
30     return True
31
32     # Solicitar la contraseña
33     contraseña = input("Introduce una contraseña: ")
34     # Evaluar y mostrar el resultado
35     resultado = validar_contraseña(contraseña)
36     print(f"Resultado: {resultado}")

```

Código 9.6: Solución al Ejercicio 9.6.

EJERCICIO 9.7

Escribe un programa en Python que reciba una lista de números enteros y determine si los números están en orden consecutivo ascendente o descendente.

Ejemplo de ejecución esperada

Introduce una lista de números (separados por espacio): 3 4 5 6

Resultado: True

Solución (girar para leer)

```

1 def verificar_consecutivos(numeros):
2     # Verificar si están en orden ascendente
3     ascendente = True
4     for i in range(len(numeros) - 1):
5         if numeros[i] + 1 != numeros[i + 1]:
6             ascendente = False
7             break
8     # Verificar si están en orden descendente
9     descendente = True
10    for i in range(len(numeros) - 1):
11        if numeros[i] - 1 != numeros[i + 1]:
12            descendente = False
13            break
14    return ascendente or descendente
15
16 # Solicitar los números
17 entrada = input("Introduce una lista de números (separados por espacio): ")
18 numeros = [int(x) for x in entrada.split()]
19 # Evaluar y mostrar el resultado
20 resultado = verificar_consecutivos(numeros)
21 print(f"Resultado: {resultado}")

```

Código 9.7: Solución al Ejercicio 9.7.

EJERCICIO 9.8

Escribe un programa en Python que reciba una fecha en formato DD/MM/YYYY como cadena de texto y determine si es una fecha válida.

Ejemplo de ejecución esperada

```
Introduce una fecha en formato DD/MM/YYYY: 29/02/2021  
Resultado: False
```

Solución (girar para leer)

```

1  def es_bisiesto(año):
2      if (año % 4 == 0 and (año % 100 != 0 or año %
3          return True
4          return False
5      evaluar_fecha):
6      # Dividir la fecha en día, mes y año
7      día, mes, año = fecha.split("/")
8      día = int(día)
9      mes = int(mes)
10     año = int(año)
11
12     # Verificar si el mes es válido
13     if mes < 1 or mes > 12:
14         return False
15
16     # Verificar el número de días según el mes
17     if mes == 2:
18         if es_bisiesto(año):
19             return día <= 29 # Año bisiesto, febrero
20             tiene 29 días
21             return día <= 28 # Año no bisiesto, febrero
22             tiene 28 días
23         elif mes in [4, 6, 9, 11]:
24             return día <= 30 # Meses con 30 días
25             return día <= 31 # Los demás meses tienen 31 días
26     # Solicitar la fecha al usuario
27     fecha = input("Introduce una fecha en formato DD/MM/
28     YYYY: ")
29     # Evaluar y mostrar el resultado
30     resultado = evaluar_fecha(fecha)
31     print(f"Resultado: {resultado}")

```

Código 9.8: Solución al Ejercicio 9.8.

EJERCICIO 9.9

Escribe un programa en Python que reciba tres valores numéricos y evalúe si se cumple al menos una de las siguientes condiciones:

- El primer número es mayor que 10 y el segundo es menor que 5.
- El segundo número es mayor que 10 y el tercero es menor que 0.
- El tercer número es mayor que 0 y la suma de los dos primeros números es par.

Ejemplo de ejecución esperada

```
Introduce el primer número: 12
Introduce el segundo número: 4
Introduce el tercer número: 2
Resultado: True
```

Solución (gitar para leer)

```
1 def evaluar_condiciones(a, b, c):
2     # Verificar las tres condiciones
3     if (a < 10 and b > 5):
4         return True
5     if (b < 10 and c > 0):
6         return True
7     if (c < 0 and a + b) % 2 == 0):
8         return True
9     return False
10
11 # Solicitar los tres números al usuario
12 a = int(input("Introduce el primer número: "))
13 b = int(input("Introduce el segundo número: "))
14 c = int(input("Introduce el tercer número: "))
15
16 # Evaluar y mostrar el resultado
17 resultado = evaluar_condiciones(a, b, c)
18 print(f"Resultado: {resultado}")
```

Código 9.9: Solución al Ejercicio 9.9.

EJERCICIO 9.10

Escribe un programa en Python que reciba una lista de valores booleanos y determine si alguno de los valores cumple con la siguiente expresión lógica:

$$(A \wedge B) \vee (C \wedge \neg D)$$

El programa debe verificar las condiciones basadas en la lista de valores proporcionados.

Ejemplo de ejecución esperada

Introduce una lista de valores booleanos (True/False, separados por espacio): True False True False
Resultado: True

Solución (girar para leer)

```

1  def validacion_expresion_logica(valores):
2      A, B, C, D = valores[0], valores[1], valores[2],
3      valores[3]
4      # Evaluamos la expresión lógica: (A and B) or (
5      and not D)
6      return (A and B) or (C and not D)
7      # Solicitar la lista de valores booleanos
8      entrada = input("Introduce una lista de valores
9      booleanos (True/False, separados por espacio): ")
10     valores = [True if x.strip().lower() == 'true' else
11                 False for x in entrada.split()]
12     # Evaluar y mostrar el resultado
13     resultado = validacion_expresion_logica(valores)
14     print(f"Resultado: {resultado}")

```

Código 9.10: Solución al Ejercicio 9.10.

EJERCICIO 9.11

Escribe un programa en Python que reciba una lista de valores booleanos y determine si más del 75 % de los valores son True.

Ejemplo de ejecución esperada

Introduce una lista de valores booleanos: True True False True False True
Resultado: True

Solución (girar para leer)

Código 9.11: Solución al Ejercicio 9.11.

```

1 def consenso_múltiples_evaluaciones(valores):
2     # Contamos cuántos valores son True
3     cuenta_true = 0
4     for valor in valores:
5         if valor:
6             cuenta_true += 1
7     # Comprobamos si más del 75 % de los valores son
8     # True
9     return cuenta_true > len(valores) * 0.75
10 # Solicitar la lista de valores booleanos
11 entrada = input("Introduce una lista de valores
12 booleanos (True/False, separados por espacio): ")
13 valores = [True if x.strip().lower() == 'true' else
14             False for x in entrada.split()]
15 # Evaluar y mostrar el resultado
16 resultado = consenso_múltiples_evaluaciones(valores)
17 print(f"Resultado: {resultado}")

```

EJERCICIO 9.12

Escribe un programa en Python que reciba una lista de valores booleanos y determine si los valores alternan entre True y False.

Ejemplo de ejecución esperada

Introduce una lista de valores booleanos: True False True False
Resultado: True

Solución (girar para leer)

```

1 def secuencias_condiciones_alternadas(valores):
2     for i in range(len(valores) - 1):
3         if valores[i] == valores[i + 1]:
4             return False
5         return True
6
7     # Solicitar la lista de valores booleanos
8     entrada = input("Introduce una lista de valores
9     booleanos (True/False, separados por espacio): ")
10    valores = [True if x.strip().lower() == 'true' else
11               False for x in entrada.split()]
12    # Evaluar y mostrar el resultado
13    resultado = secuencias_condiciones_alternadas(valores)
14    print(f"Resultado: {resultado}")

```

Código 9.12: Solución al Ejercicio 9.12.

EJERCICIO 9.13

Escribe un programa en Python que reciba tres listas de valores booleanos y determine si en cada posición al menos dos de los tres valores coinciden.

Ejemplo de ejecución esperada

Introduce tres listas de valores booleanos:

True False True

False False True

True True False

Resultado: True

Solución (girar para leer)

```

1  def consenso_logico_cruzado(lista1, lista2, lista3):
2      for i in range(len(lista1)):
3          if lista1[i] == lista2[i] or lista1[i] ==
4              lista3[i]:
5              continue
6          else:
7              return False
8              return True
9      # Solicitar las tres listas de valores booleanos
10     print("Introduce tres listas de valores booleanos:")
11     lista1 = [True if x.strip().lower() == 'true' else
12               False for x in input("Lista 1: ").split()]
13     lista2 = [True if x.strip().lower() == 'true' else
14               False for x in input("Lista 2: ").split()]
15     lista3 = [True if x.strip().lower() == 'true' else
16               False for x in input("Lista 3: ").split()]
17     # Evaluar y mostrar el resultado
18     resultado = consenso_logico_cruzado(lista1, lista2,
19                                         lista3)
20     print(f"Resultado: {resultado}")

```

Código 9.13: Solución al Ejercicio 9.13.

EJERCICIO 9.14

Escribe un programa en Python que reciba una cadena de bits (una secuencia de valores True y False) y determine si la cantidad de True en la cadena es **par** o **impar**. El programa debe devolver True si la cantidad de True es **par**, y False si la cantidad de True es **impar**.

Ejemplo de ejecución esperada

Introduce la cadena de bits (True/False, separados por espacio): True False True False True
Resultado: False

Solución (girar para leer)

```

1 def evaluar_paridad(bits):
2     cuenta_true = 0
3     for bit in bits:
4         if bit:
5             cuenta_true += 1
6     # Verificamos si la cantidad de True es par
7     return cuenta_true % 2 == 0
8
9 # Solicitar la cadena de bits al usuario
10 entrada = input("Introduce la cadena de bits (True/
    ↵ False, separados por espacio): ")
11 bits = []
12 for x in entrada.split():
13     if x.strip().lower() == 'true':
14         bits.append(True)
15     else:
16         bits.append(False)
17
18 # Evaluar y mostrar el resultado
19 resultado = evaluar_paridad(bits)
20 print(f"Resultado: {resultado}")

```

Código 9.14: Solución al Ejercicio 9.14.

EJERCICIO 9.15

Escribe un programa en Python que reciba una cadena de bits (una secuencia de valores `True` y `False`) y determine si los valores alternan entre `True` y `False`. El programa debe devolver `True` si los valores alternan correctamente, y `False` si no lo hacen.

Ejemplo de ejecución esperada

Introduce la cadena de bits (True/False, separados por espacio): `True False True False True`
 Resultado: `True`

Solución (girar para leer)

```

1 def verificar_secuencia_alternada(bits):
2     for i in range(len(bits) - 1):
3         if bits[i] == bits[i + 1]: # Si dos valores
4             return False
5         return True
6     # Solicitar la cadena de bits al usuario
7     entrada = input("Introduce la cadena de bits (True/
8     False, separados por espacio): ")
9     bits = []
10    for x in entrada.split():
11        if x.strip().lower() == 'true':
12            bits.append(True)
13        else:
14            bits.append(False)
15    # Evaluar y mostrar el resultado
16    resultado = verificar_secuencia_alternada(bits)
17    print(f"Resultado: {resultado}")

```

Código 9.15: Solución al Ejercicio 9.15.

EJERCICIO 9.16

Escribe un programa en Python que reciba una cadena y determine cuántas subcadenas palindrómicas contiene. Una subcadena es palindrómica si se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo, la cadena "abba" tiene 6 subcadenas palindrómicas: "a" (2 veces), "b" (2 veces), "bb" y "abba".

— Ejemplo de ejecución esperada —

Introduce una cadena: abba

Resultado: 6

Solución (girar para leer)

```

1  def es_palindromo(s):
2      return s[::-1] == s
3
4  def contar_palindromos(cadena):
5      contador = 0
6      for i in range(len(cadena)):
7          for j in range(i + 1, len(cadena) + 1):
8              if es_palindromo(cadena[i:j]):
9                  contador += 1
10         return contador
11
12     # Solicitar la cadena al usuario
13     cadena = input("Introduce una cadena: ")
14
15     # Evaluar y mostrar el resultado
16     resultado = contar_palindromos(cadena)
17     print(f"Resultado: {resultado}")

```

Código 9.16: Solución al Ejercicio 9.16.

EJERCICIO 9.17

Escribe un programa en Python que reciba una cadena de texto y la convierta a formato *Title Case*. En este formato, la primera letra de cada palabra debe ser mayúscula, pero deben aplicarse las siguientes reglas:

- Las palabras con menos de 4 caracteres deben mantenerse en minúscula (excepto la primera y la última palabra).
- Las preposiciones, artículos y conjunciones que aparecen en medio de la cadena deben mantenerse en minúscula (por ejemplo, "de", "a", "para", "y", "en"; considera que su longitud es siempre menor que 4).

Ejemplo de ejecución esperada

Introduce un texto: el rápido zorro marrón salta sobre la liebre perezosa
 Resultado: El Rápido Zorro Marrón Salta Sobre la Liebre Perezosa

Solución (girar para leer)

```

1 def convertir_title_case(cadena):
2     palabras = cadena.split()
3     resultado = ""
4     for i in range(len(palabras)):
5         if i == 0 or i == len(palabras) - 1 or len(
6             palabras[i]) >= 4:
7             resultado += palabras[i].capitalize() + " "
8         else:
9             resultado += palabras[i].lower() + " "
10    return resultado.strip()
11
12 # Solicitar el texto al usuario
13 texto = input("Introduce un texto: ")
14
15 # Evaluar y mostrar el resultado
16 resultado = convertir_title_case(texto)
17 print(f"Resultado: {resultado}")

```

Código 9.17: Solución al Ejercicio 9.17.

EJERCICIO 9.18

Escribe un programa en Python que reciba una cadena y devuelva todas las combinaciones posibles de caracteres únicos que se pueden formar con los caracteres de la cadena. No se deben incluir combinaciones repetidas (por ejemplo, si la cadena tiene caracteres repetidos, no debe generarse la misma combinación más de una vez).

Ejemplo de ejecución esperada

Introduce una cadena: aab

Resultado: a b ab

Ejemplo de ejecución esperada

Introduce una cadena: abba

Resultado: a b ab ba

Solución (girar para leer)

```

1  def combinaciones_unicas(cadena):
2      resultado = ""
3
4      # Generamos todas las subcadenas posibles
5      for i in range(1, len(cadena) + 1):
6          for j in range(len(cadena) - i + 1):
7              subcadena = cadena[j:j+i]
8
9      # Verificamos si la subcadena tiene
10         caracteres unicos
11         es_unica = True
12         for k in range(len(subcadena)):
13             for l in range(k + 1, len(subcadena)):
14                 if subcadena[k] == subcadena[l]:
15                     es_unica = False
16                     break
17                 if not es_unica:
18                     break
19         if es_unica and subcadena not in resultado:
20             resultado += subcadena + " "
21
22     return resultado.strip()
23
24     # Solicitar la cadena al usuario
25     cadena = input("Introduce una cadena: ")
26
27     # Evaluar y mostrar el resultado
28     resultado = combinaciones_unicas(cadena)
29     print(f"Resultado: {resultado}")

```

Código 9.18: Solución al Ejercicio 9.18.

EJERCICIO 9.19

Escribe un programa en Python que reciba una cadena de texto y cuente cuántas veces aparece cada palabra en la cadena. El programa debe devolver un diccionario con las palabras como claves y su frecuencia como valores.

Ejemplo de ejecución esperada

Introduce una cadena: Hola hola mundo mundo mundo

Resultado: hola:2 mundo:3

Solución (girar para leer)

```

1  def contar_frecuencia_palabras(cadena):
2      palabras = cadena.lower().split()
3      resultado = ""
4      # Recorrer las palabras y contar las frecuencias
5      for palabra in palabras:
6          contador = 0
7          for p in palabras:
8              if p == palabra:
9                  contador += 1
10             if "{"palabra}:{contador}" not in resultado:
11                 resultado += "{"palabra}:{contador}"
12             return resultado.strip()
13         # Solicitar la cadena al usuario
14         cadena = input("Introduce una cadena: ")
15         # Evaluar y mostrar el resultado
16         resultado = contar_frecuencia_palabras(cadena)
17         print(f"Resultado: {resultado}")

```

Código 9.19: Solución al Ejercicio 9.19.

EJERCICIO 9.20

Escribe un programa en Python que reciba una cadena y devuelva la subcadena más larga que contenga sólo caracteres únicos. Si hay más de una subcadena con la misma longitud, devuelve la que aparezca primero en la cadena original.

Ejemplo de ejecución esperada

Introduce una cadena: abcabcbb

Resultado: abc

Solución (girar para leer)

```

1 def subcadena_mas_larga_unica(cadena):
2     max_subcadena = ""
3     subcadena_actual = ""
4     for char in cadena:
5         if char not in subcadena_actual:
6             subcadena_actual += char
7         else:
8             subcadena_actual = subcadena_actual[
9                 subcadena_actual.index(char) + 1:] +
10             char
11     if len(subcadena_actual) > len(max_subcadena):
12         max_subcadena = subcadena_actual
13     return max_subcadena
14
15 # Solicitar la cadena al usuario
16 cadena = input("Introduce una cadena: ")
17
18 # Evaluar y mostrar el resultado
19 resultado = subcadena_mas_larga_unica(cadena)
20 print(f"Resultado: {resultado}")

```

Código 9.20: Solución al Ejercicio 9.20.

EJERCICIO 9.21

Escribe un programa en Python que reciba una cadena y determine si contiene caracteres repetidos consecutivos. Devuelve True si se encuentran, de lo contrario, devuelve False.

Ejemplo de ejecución esperada

Introduce una cadena: Hola Mundo

Resultado: False

Solución (girar para leer)

```

1  def comprobacion_caracteres_repetidos(cadena):
2      for i in range(len(cadena) - 1):
3          if cadena[i] == cadena[i + 1]:
4              return True
5          return False
6      # Solicitar la cadena al usuario
7      cadena = input("Introduce una cadena: ")
8      # Evaluar y mostrar el resultado
9      resultado = comprobacion_caracteres_repetidos(cadena)
10     print(f"Resultado: {resultado}")

```

Código 9.21: Solución al Ejercicio 9.21.

EJERCICIO 9.22

Escribe un programa en Python que reciba una cadena de texto y devuelva la cadena con cada palabra invertida, pero manteniendo el orden de las palabras. Por ejemplo, "Hola Mundo" debe convertirse en "aloH odnuM".

Ejemplo de ejecución esperada

Introduce una cadena: Hola Mundo

Resultado: aloH odnuM

Solución (girar para leer)

```

1  def inversion_palabra_por_palabra(cadena):
2      palabras = cadena.split() # Dividimos la cadena en
3      palabras
4      resultado = ""
5      # Recorremos cada palabra y reversionla manualmente
6      for palabra in palabras:
7          palabra_invertida = ""
8          for i in range(len(palabra) - 1, -1, -1): #
9              # Recorremos la palabra de atrás hacia
10             # adelante
11             palabra_invertida += palabra[i]
12             resultado += palabra_invertida + " "
13         return resultado.strip()
14     # Solicitar la cadena al usuario
15     cadena = input("Introduce una cadena: ")
16     # Evaluar y mostrar el resultado
17     resultado = inversion_palabra_por_palabra(cadena)
18     print(f"Resultado: {resultado}")

```

Código 9.22: Solución al Ejercicio 9.22.

EJERCICIO 9.23

Escribe un programa en Python que reciba una cadena y una subcadena y determine si la subcadena aparece en la cadena principal. Devuelve `True` si la subcadena está presente, de lo contrario, devuelve `False`.

Ejemplo de ejecución esperada

Introduce la cadena: Hola Mundo

Introduce la subcadena: Mundo

Resultado: `True`

Solución (girar para leer)

```

1 def verificacion_subcadena(cadena, subcadena):
2     # comprobamos si la subcadena aparece en la cadena
3     # Recorremos la cadena hasta el tamaño de la
4     # Comparamos la subcadena con la parte
5     # actual de la cadena
6     return True
7     # También se podría usar directamente:
8     # return subcadena in cadena
9
10    # Solicitar la cadena y la subcadena al usuario
11    cadena = input("Introduce la cadena: ")
12    subcadena = input("Introduce la subcadena: ")
13
14    # Evaluar y mostrar el resultado
15    resultado = verificacion_subcadena(cadena, subcadena)
16    print(f"Resultado: {resultado}")

```

Código 9.23: Solución al Ejercicio 9.23.

EJERCICIO 9.24

Escribe un programa en Python que reciba dos cadenas y determine si una es un anagrama de la otra. Un anagrama es una palabra o frase formada por la reordenación de las letras de otra, usando todas las letras originales exactamente una vez.

Ejemplo de ejecución esperada

Introduce la primera cadena: listen
 Introduce la segunda cadena: silent
 Resultado: True

Solución (girar para leer)

```

1  def es_anagrama(cadena1, cadena2):
2      # Eliminar espacios y convertir a minúsculas
3      cadena1 = cadena1.replace(" ", "").lower()
4      cadena2 = cadena2.replace(" ", "").lower()
5
6      # Comparamos si las letras de ambas cadenas son
7      iguales
8      if sorted(cadena1) == sorted(cadena2):
9          return True
10         return False
11
12     # Solicitar las cadenas al usuario
13     cadena1 = input("Introduce la primera cadena: ")
14     cadena2 = input("Introduce la segunda cadena: ")
15
16     # Evaluar y mostrar el resultado
17     resultado = es_anagrama(cadena1, cadena2)
18     print(f"Resultado: {resultado}")

```

Código 9.24: Solución al Ejercicio 9.24.

EJERCICIO 9.25

Escribe un programa en Python que reciba una cadena y un número n , y devuelva cuántas palabras en la cadena tienen exactamente n caracteres.

Ejemplo de ejecución esperada

Introduce una cadena: Hola Mundo programadores

Introduce la longitud de palabra: 5

Resultado: 1

Solución (girar para leer)

```

1 def contar_palabras_longitud_especifica(cadena, n):
2     # Dividimos la cadena en palabras
3     palabras = cadena.split()
4     contador = 0
5     # Contamos cuántas palabras tienen exactamente n
6     caracteres
7     for palabra in palabras:
8         if len(palabra) == n:
9             contador += 1
10
11     return contador
12
13 # Solicitar la cadena y la longitud al usuario
14 cadena = input("Introduce una cadena: ")
15 n = int(input("Introduce la longitud de palabra: "))
16
17 # Evaluar y mostrar el resultado
18 resultado = contar_palabras_longitud_especifica(cadena,
19 print(f"Resultado: {resultado}")

```

Código 9.25: Solución al Ejercicio 9.25.

EJERCICIO 9.26

Escribe un programa en Python que reciba una cadena y determine cuál es la secuencia de caracteres repetidos más larga en la cadena. La función debe devolver la secuencia más larga y su longitud. Si hay varias secuencias de la misma longitud, debe devolver la primera.

Ejemplo de ejecución esperada

Introduce una cadena: aaabbbbbccccc

Resultado: bbbbbb 5

Solución (girar para leer)

```

1 def secuencia_repetida_larga(cadena):
2     max_secuencia = ""
3     secuencia_actual = ""
4
5     # Iteramos sobre la cadena para encontrar las
6     secuencias repetidas
7     for i in range(len(cadena)):
8         if i == 0 or cadena[i] == cadena[i - 1]:
9             secuencia_actual += cadena[i]
10        else:
11            if len(secuencia_actual) > len(
12                max_secuencia):
13                max_secuencia = secuencia_actual
14            secuencia_actual = cadena[i]
15        # Verificar la última secuencia
16        if len(secuencia_actual) > len(max_secuencia):
17            max_secuencia = secuencia_actual
18    return max_secuencia, len(max_secuencia)
19
20 # Solicitar la cadena al usuario
21 cadena = input("Introduce una cadena: ")
22
23 # Evaluar y mostrar el resultado
24 secuencia, longitud = secuencia_repetida_larga(cadena)
25 print(f"Resultado: {secuencia} {longitud}")

```

Código 9.26: Solución al Ejercicio 9.26.

EJERCICIO 9.27

Escribe un programa en Python que reciba una cadena de texto y devuelva todas las combinaciones únicas de palabras que se pueden formar utilizando las palabras de la cadena. No se deben incluir combinaciones repetidas (por ejemplo, si una palabra aparece varias veces, no se debe generar la misma combinación más de una vez).

Ejemplo de ejecución esperada

Introduce una cadena: hola mundo hola

Resultado: hola mundo, hola hola

Solución (girar para leer)

```

1 def combinaciones_unicas(cadena):
2     # Dividimos la cadena en palabras
3     palabras = cadena.split()
4     resultado = ""
5     # Recorremos las palabras para crear combinaciones
6     for i in range(len(palabras)):
7         for j in range(i + 1, len(palabras)):
8             # Combinamos palabras y evitamos
9             # repeticiones
10            combinacion = palabras[i] + " " + palabras[j]
11            combinacion_invertida = palabras[j] + " " + palabras[i]
12            palabras[i]
13            # Verificamos que las combinaciones no estén
14            # repetidas
15            if combinacion not in resultado and combinacion_invertida not in resultado:
16                resultado += combinacion + " "
17            resultado += combinacion_invertida + " "
18            return resultado.strip(", ")
19        # Solicitar la cadena al usuario
20        cadena = input("Introduce una cadena: ")
21        # Evaluar y mostrar el resultado
22        resultado = combinaciones_unicas(cadena)
23        print(f"Resultado: {resultado}")
24

```

Código 9.27: Solución al Ejercicio 9.27.

EJERCICIO 9.28

Escribe un programa en Python que reciba una cadena de texto y determine la longitud de la palabra más larga que aparece en la cadena. El programa debe de escribir la palabra más larga y su longitud como resultado. La palabra más larga debe ser identificada independientemente de si la cadena contiene espacios u otros delimitadores.

Ejemplo de ejecución esperada

Introduce una cadena: La programación en Python es divertida

Resultado: programación 12

Solución (girar para leer)

```

1 def longitud_palabra_maxima(cadena):
2     palabras = cadena.split() # Dividimos la cadena en
3     palabras
4     max_palabra = ""
5     # Recorremos las palabras para encontrar la más
6     larga
7     for palabra in palabras:
8         if len(palabra) > len(max_palabra):
9             max_palabra = palabra
10    return max_palabra, len(max_palabra)
11
12 # Solicitar la cadena al usuario
13 cadena = input("Introduce una cadena: ")
14
15 # Evaluar y mostrar el resultado
16 palabra, longitud = longitud_palabra_maxima(cadena)
17 print(f"Resultado: {palabra} {longitud}")

```

Código 9.28: Solución al Ejercicio 9.28.

EJERCICIO 9.29

Escribe un programa en Python que reciba una cadena y determine la subcadena palindrómica más larga que aparece en la cadena. La subcadena debe ser la más larga que cumpla con la propiedad de ser palíndroma, y el programa debe devolver dicha subcadena y su longitud.

Ejemplo de ejecución esperada

Introduce una cadena: babad

Resultado: bab 3

Solución (girar para leer)

```

1  def es_palindromo(s):
2      # Función para verificar si una subcadena es palíndroma
3      return s[::-1] == s
4
5  def subcadena_palindromica_longitud(cadena):
6      max_subcadena = ""
7      # Recorremos todas las subcadenas posibles
8      for i in range(len(cadena)):
9          for j in range(i + 1, len(cadena) + 1):
10             subcadena = cadena[i:j]
11             if es_palindromo(subcadena) and len(subcadena) > len(max_subcadena):
12                 max_subcadena = subcadena
13
14     return max_subcadena, len(max_subcadena)
15
16 # Solicitar la cadena al usuario
17 cadena = input("Introduce una cadena: ")
18
19 # Evaluar y mostrar el resultado
20 subcadena, longitud = subcadena_palindromica_longitud(cadena)
21
22 print(f"Resultado: {subcadena} {longitud}")

```

Código 9.29: Solución al Ejercicio 9.29.

EJERCICIO 9.30

Escribe un programa en Python que reciba una cadena de texto y determine cuántos pares de subcadenas dentro de la cadena son anagramas. Dos subcadenas son anagramas si tienen las mismas letras, pero en diferente orden.

Ejemplo de ejecución esperada

Introduce una cadena: abba

Resultado: 4

Solución (girar para leer)

```

1  def son_anagramas(sub1, sub2):
2      # Verificar si dos subcadenas son anagramas
3      comparando las frecuencias de cada carácter
4      if len(sub1) != len(sub2):
5          return False
6      # Contamos la frecuencia de caracteres de cada
7      subcadena
8      for char in sub1:
9          if sub1.count(char) != sub2.count(char):
10             return False
11     return True
12
13 def contar_pares_anagramas(cadena):
14     count = 0
15     n = len(cadena)
16     # Recorremos todas las posibles subcadenas
17     for i in range(n):
18         for j in range(i + 1, n + 1):
19             for k in range(i + 1, n + 1):
20                 for l in range(k + 1, n + 1):
21                     # Verificamos si las subcadenas son
22                     anagramas
23                 if son_anagramas(cadena[i:j]),
24                     cadena[k:l]):
25                         count += 1
26     return count
27
28 # Solicitar la cadena al usuario
29 cadena = input("Introduce una cadena: ")
30
31 # Evaluar y mostrar el resultado
32 resultado = contar_pares_anagramas(cadena)
33 print(f"Resultado: {resultado}")

```

Código 9.30: Solución al Ejercicio 9.30.

10

Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

En este capítulo exploraremos los **tipos compuestos** de Python: listas, tuplas, conjuntos y diccionarios. Estos cuatro tipos cubren la inmensa mayoría de necesidades de almacenamiento estructurado en memoria y son la base sobre la que se construyen las bibliotecas científicas y de seguridad: `numpy`, `pandas`, parsers de protocolos, gestores de eventos, etc. Antes de entrar en cada tipo, introduciremos un concepto transversal – la *mutabilidad* – que explica por qué unos tipos pueden ser claves de un diccionario y otros no, y por qué algunas “modificaciones” no producen los efectos que cabría esperar. Cerraremos el capítulo presentando las **funciones de primera clase** y de **orden superior**, mecanismos clave del paradigma funcional que Python ofrece de manera natural y que aparecen continuamente en código moderno.

10.1. Tipos mutables e inmutables

Es importante distinguir entre tipos **mutables** e **inmutables** para entender cómo se gestionan los datos en memoria. Un *tipo mutable* permite modificar su contenido tras la creación, sin generar un nuevo objeto. Un *tipo inmutable*, en cambio, no admite cambios: cualquier “modificación” produce en realidad un objeto nuevo.

La mutabilidad tiene implicaciones prácticas importantes:

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

Mutables	Inmutables
<code>list</code> , <code>dict</code> , <code>set</code> , <code>bytearray</code>	<code>int</code> , <code>float</code> , <code>bool</code> , <code>str</code> , <code>tuple</code> , <code>frozenset</code> , <code>bytes</code>

Tabla 1: Tipos mutables e inmutables más comunes en Python.

- **Eficiencia:** modificar una lista en sitio es más barato que crear una nueva cadena cada vez.
- **Aliasing:** si dos variables referencian el mismo objeto mutable, los cambios realizados a través de una afectan a la otra. Esto puede ser deseable o ser fuente de *bugs* sutiles.
- **Hashabilidad:** solo los objetos inmutables (y compuestos por inmutables) son *hashables*, es decir, pueden usarse como claves de un diccionario o elementos de un conjunto. Una tupla (1, 2, 3) vale como clave; una lista [1, 2, 3] no.

Aliasing y argumentos mutables por defecto

Una práctica de programación habitual es escribir funciones con valores por defecto. Si el valor por defecto es un objeto mutable (lista, diccionario, conjunto), Python lo crea **una sola vez** y todas las llamadas comparten el mismo objeto. Esto produce un *bug* clásico:

```
def agregar(elem, lista=[]): # MAL: la lista se
    comparte entre llamadas
    lista.append(elem)
    return lista

print(agregar(1)) # [1]
print(agregar(2)) # [1, 2] <-- !! Estado conservado
entre llamadas
```

La forma correcta es usar `None` como centinela y crear el objeto dentro:

```
def agregar(elem, lista=None):
    if lista is None:
        lista = []
    lista.append(elem)
    return lista
```

10.2. Listas

Una **lista** (tipo `list`) es una colección *ordenada y mutable* de elementos. Los elementos pueden ser de cualquier tipo (incluso heterogéneos en una misma lista, aunque

normalmente se trabaja con listas homogéneas). El acceso es por índice empezando en 0, y el tamaño no es fijo: la lista crece y decrece dinámicamente según se añadan o eliminen elementos.

10.2.1. Creación

```

T = [0] * N                                # Lista de N ceros
M = [[0] * NC for _ in range(NF)]          # Matriz NF x NC
    inicializada a 0

letras_romanas = ['I', 'V', 'X', 'L', 'C', 'D', 'M']
valores         = [1, 5, 10, 50, 100, 500, 1000]
matriz_identidad = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

```

10.2.2. Indexación y rebanado

Igual que con las cadenas (capítulo 9), las listas admiten indexación con índices positivos y negativos, y **rebanado** (*slicing*) con la sintaxis [inicio:fin:paso]:

```

v = [10, 20, 30, 40, 50]
print(v[0])      # 10
print(v[-1])     # 50      (último elemento)
print(v[1:4])    # [20, 30, 40]
print(v[:2])     # [10, 30, 50]
print(v[::-1])   # [50, 40, 30, 20, 10]      (reverso)

```

A diferencia de las cadenas, en una lista podemos **asignar** a un *slice* para reemplazar varios elementos a la vez:

```

v = [10, 20, 30, 40, 50]
v[1:4] = [99, 99]      # Reemplaza posiciones 1..3
print(v)               # [10, 99, 99, 50]

```

10.2.3. Operadores y métodos comunes

- `lista otra+`: concatenación (devuelve nueva lista).
- `lista * n`: repetición.
- `x in lista`: pertenencia.
- `len(lista)`: longitud.

Los métodos más utilizados son:

- `lista.append(x)`: añade x al final.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

- `lista.extend(iter)`: añade los elementos del iterable al final.
- `lista.insert(i, x)`: inserta `x` en la posición `i`.
- `lista.remove(x)`: elimina la primera aparición de `x` (`ValueError` si no está).
- `lista.pop([i])`: elimina y devuelve el elemento en posición `i` (último por defecto).
- `lista.index(x)`: índice de la primera aparición de `x`.
- `lista.count(x)`: número de apariciones de `x`.
- `lista.sort()`: ordena *in situ*.
- `lista.reverse()`: invierte *in situ*.
- `lista.copy()`: copia superficial.

append vs extend

`append` añade su argumento como un **único elemento**; `extend` desempaqueta el iterable y añade cada elemento por separado:

```
a = [1, 2, 3]
a.append([4, 5]) # a = [1, 2, 3, [4, 5]]      sublista
→          anyadida
a = [1, 2, 3]
a.extend([4, 5]) # a = [1, 2, 3, 4, 5]      elementos
→          anyadidos
```

10.2.4. Listas por comprensión

Una **lista por comprensión** (*list comprehension*) es una sintaxis concisa para construir una lista a partir de un iterable. La forma general es:

```
[expresion for variable in iterable if condicion]
```

Es equivalente a un bucle `for` con `append`, pero más compacto y normalmente más rápido. Ejemplos:

```
cuadrados = [x ** 2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

pares = [x for x in range(20) if x % 2 == 0]
# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```



```

palabras = ["Hola", "mundo", "Python"]
longitudes = [len(p) for p in palabras]
# [4, 5, 6]

# Comprensiones anidadas: aplanar una matriz
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
plana = [x for fila in matriz for x in fila]
# [1, 2, 3, 4, 5, 6, 7, 8, 9]

```

10.2.5. Generadores y expresiones generadoras

Una **expresión generadora** es como una lista por comprensión pero con paréntesis en vez de corchetes. No construye la lista en memoria, sino que produce los elementos *bajo demanda*, lo que ahorra memoria cuando se procesan grandes volúmenes de datos:

```

# Suma del cuadrado de los primeros millon de enteros, sin
  crear la lista
total = sum(x ** 2 for x in range(1_000_000))

```

Las funciones que usan la palabra clave **yield** se llaman **funciones generadoras**: cada **yield** produce un valor y suspende la ejecución hasta la siguiente petición. Esto permite definir secuencias potencialmente infinitas o pipelines de procesamiento sin almacenarlas íntegras en memoria:

```

def primos():
    """Generador infinito de numeros primos."""
    candidato = 2
    while True:
        # Comprobacion ingenua: divisibilidad por primos
        # menores que sqrt
        es_primo = all(candidato % p != 0 for p in range(2,
            int(candidato**0.5) + 1))
        if es_primo:
            yield candidato
        candidato += 1

# Tomar los 10 primeros primos
from itertools import islice
print(list(islice(primos(), 10)))
# [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

```

10.2.6. Listas multidimensionales

En Python, las matrices se implementan como listas de listas. Por ejemplo, una matriz cuadrada de tamaño $DIM \times DIM$ inicializada a 0:

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

```
DIM = 12
matriz = [[0] * DIM for _ in range(DIM)]
```

Cuidado al inicializar listas anidadas con *

`[[0] * DIM] * DIM` **no** crea una matriz: crea una lista de DIM *referencias* a la misma lista interna. Modificar una fila modifica todas:

```
m = [[0] * 3] * 3
m[0][0] = 1
print(m)  # [[1, 0, 0], [1, 0, 0], [1, 0, 0]]  <-- !!
```

La forma correcta es la lista por comprensión `[[0] * DIM for _ in range(DIM)]`, que crea una lista interna **nueva** en cada iteración.

Para transponer e imprimir una matriz (cuadrada). La función `transponer` intercambia los elementos respecto a la diagonal in situ, por lo que solo es válida para matrices cuadradas; transponer una matriz rectangular requiere construir una matriz nueva con dimensiones intercambiadas.

```
def transponer(m):
    n = len(m)
    for i in range(n):
        for j in range(i + 1, n):
            m[i][j], m[j][i] = m[j][i], m[i][j]

def mostrar_matriz(m):
    for fila in m:
        print(" ".join(f"{elem:6}" for elem in fila))
```

En `mostrar_matriz`, `" ".join(...)` une los elementos de la fila con un espacio como separador. La f-string `f"{elem:6}"` formatea cada número alineándolo a la derecha en un campo de al menos 6 caracteres.

10.2.7. Algoritmos comunes con listas

A continuación recogemos varios algoritmos clásicos sobre listas. Todos pueden expresarse de forma más concisa con funciones *built-in* o por comprensión, pero conviene conocer la implementación manual.

```
for elemento in lista:
    print(elemento)
```

```

1 def buscar_elemento(lista, objetivo):
2     for i in range(len(lista)):
3         if lista[i] == objetivo:
4             return i # Índice de la primera aparicion
5     return -1        # No encontrado
6
7 print(buscar_elemento([10, 20, 30, 40, 50], 30)) # 2

```

```

1 def ordenacion_burbuja(lista):
2     n = len(lista)
3     for i in range(n):
4         for j in range(0, n - i - 1):
5             if lista[j] > lista[j + 1]:
6                 lista[j], lista[j + 1] = lista[j + 1], lista[
↪                 j]

```

```

1 def ordenacion_seleccion(lista):
2     for i in range(len(lista)):
3         min_idx = i
4         for j in range(i + 1, len(lista)):
5             if lista[j] < lista[min_idx]:
6                 min_idx = j
7     lista[i], lista[min_idx] = lista[min_idx], lista[i]

```

```

1 def eliminar_duplicados(lista):
2     vistos = set() # Conjunto: pertenencia en O(1)
3     resultado = []
4     for elemento in lista:
5         if elemento not in vistos:
6             vistos.add(elemento)
7             resultado.append(elemento)
8     return resultado
9
10 print(eliminar_duplicados([1, 2, 2, 3, 4, 4, 5])) # [1, 2,
↪ 3, 4, 5]

```

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

Suma, máximo, mínimo, conteo. Equivalentes a `sum`, `max`, `min`, `lista.count` pero implementados manualmente:

```
1 def sumar_lista(lista):
2     suma = 0
3     for elemento in lista:
4         suma += elemento
5     return suma
6
7 def encontrar_maximo(lista):
8     maximo = lista[0]
9     for elemento in lista[1:]:
10         if elemento > maximo:
11             maximo = elemento
12     return maximo
13
14 def contar_ocurrencias(lista, valor):
15     return sum(1 for elemento in lista if elemento == valor)
```

10.2.8. Ejemplos

```
def es_primo(n):
    tabla_primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
        ↪      37, 41, 43, 47,
        ↪      53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
    if n < 2:
        return False
    return n in tabla_primos
```

Validar la letra del NIF español. El NIF consiste en un DNI seguido de una letra obtenida del resto de dividir el DNI entre 23, indexando una tabla:

```
def validar_nif(dni, letra):
    tabla_nif = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P',
        ↪      'D', 'X', 'B',
        ↪      'N', 'J', 'Z', 'S', 'Q', 'V', 'H', 'L', 'C',
        ↪      'K', 'E']
    return tabla_nif[dni % 23] == letra.upper()
```

```
import math
```

```

def media(v):
    return sum(v) / len(v)

def desviacion(v):
    m = media(v)
    return math.sqrt(sum((x - m) ** 2 for x in v) / (len(v) -
    ↪      1))

```

Para la moda, conviene usar `collections.Counter`, que veremos en la sección 10.5.6.

10.3. Tuplas

Estadísticas básicas. Una **tupla** (tipo `tuple`) es una secuencia **ordenada e inmutable** de elementos. Es como una lista, pero no se puede modificar tras la creación. Son útiles para:

- Devolver varios valores desde una función (los valores quedan agrupados).
- Representar registros con campos posicionales (*coordenadas, punto, rango*).
- Servir como claves de un diccionario o elementos de un conjunto (las listas no son hashables, las tuplas sí).

10.3.1. Creación

Las tuplas se crean con paréntesis y comas separando los elementos. Para una tupla de un único elemento, la coma es **obligatoria**:

```

tupla1 = (1, 2, 3, 4, 5)
tupla2 = (10,)          # Tupla de un elemento; la coma
↪   es necesaria
tupla_vacia = ()
tupla3 = (1, "hola", 3.14, True)

```

Los paréntesis son opcionales en muchos contextos (la coma es lo que define la tupla), pero conviene escribirlos para claridad: `t = 1, 2, 3` es perfectamente válido.

El acceso por índice es como en las listas, incluyendo índices negativos y rebanado.

10.3.2. Desempaquetado y tuplas anidadas

El **desempaquetado** permite asignar los elementos de una tupla a variables individuales en una sola línea:

```

punto = (3, 4)
x, y = punto
# x = 3, y = 4

```

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

```
# Operador * para capturar elementos restantes
primero, *resto = [1, 2, 3, 4, 5]
# primero = 1, resto = [2, 3, 4]

# Intercambio sin variable temporal
a, b = 10, 20
a, b = b, a    # a = 20, b = 10
```

Las tuplas pueden anidarse arbitrariamente:

```
tupla_anidada = (1, (2, 3), (4, 5), 6)
print(tupla_anidada[1])    # (2, 3)
print(tupla_anidada[1][0]) # 2
```

10.3.3. Operaciones e inmutabilidad

```
print((1, 2) + (3, 4))    # (1, 2, 3, 4)
print((1, 2) * 3)         # (1, 2, 1, 2, 1, 2)
print((1, 2, 3) == (1, 2, 3)) # True
```

Inmutabilidad superficial

Una tupla es inmutable, pero **si contiene referencias a objetos mutables**, esos objetos sí pueden cambiar:

```
t = (1, [2, 3])
# t[0] = 99          # TypeError: la tupla es inmutable
t[1].append(4)      # OK: la lista interior si es mutable
print(t)            # (1, [2, 3, 4])
```

Esto significa que una tupla **con elementos mutables no es hashable** y, por tanto, no puede usarse como clave de un diccionario.

10.3.4. Tuplas con nombre: namedtuple

Una tupla normal accede a sus campos por índice (`t[0]`, `t[1]`), lo que es críptico cuando hay muchos campos. La función `namedtuple` del módulo `collections` crea un *tipo* de tupla con campos nombrados, manteniendo la inmutabilidad y la eficiencia:

```
from collections import namedtuple

Punto = namedtuple("Punto", ["x", "y"])
p = Punto(3, 4)
print(p.x, p.y)    # 3 4
print(p[0], p[1])  # 3 4 (sigue siendo una tupla)
print(p)           # Punto(x=3, y=4)
```

Para casos más complejos (con valores por defecto, métodos, validación, etc.), conviene usar `@dataclass` del módulo `dataclasses`:

```
from dataclasses import dataclass

@dataclass
class Punto:
    x: float
    y: float

p = Punto(3, 4)
print(p.x, p.y)
```

A diferencia de `namedtuple`, los *dataclasses* son mutables por defecto (se puede pedir `@dataclass(frozen=True)` para inmutabilidad).

10.3.5. Devolver varios valores

Una función que devuelve varios valores los empaqueta en una tupla, lo que el llamador puede desempaquetar directamente:

```
def operaciones(a, b):
    return a + b, a - b, a * b    # Tupla implícita

suma, resta, mult = operaciones(5, 3)
```

10.4. Conjuntos

Un **conjunto** (tipo `set`) es una colección **desordenada y mutable** de elementos **únicos**. No admite duplicados y los elementos deben ser *hashables* (inmutables). Los conjuntos son la estructura ideal para:

- Comprobar pertenencia rápida (operación `in` en tiempo $O(1)$ amortizado, frente a $O(n)$ en una lista).
- Eliminar duplicados.
- Realizar operaciones de teoría de conjuntos (unión, intersección, diferencia).

10.4.1. Creación

Se crean con `set()` o con llaves `{}` (con elementos; `{}` vacías son un *diccionario*, no un conjunto):

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

```
conjunto1 = {1, 2, 3, 4, 5}
conjunto2 = set([1, 2, 2, 3, 3, 4])
print(conjunto2)  # {1, 2, 3, 4} (duplicados eliminados)

vacio = set()      # Conjunto vacio (no se puede escribir {})
```

10.4.2. Operaciones básicas

- `s.add(x)`: añade `x` (sin efecto si ya está).
- `s.remove(x)`: elimina `x`, lanza `KeyError` si no está.
- `s.discard(x)`: elimina `x` silenciosamente si está.
- `s.pop()`: elimina y devuelve un elemento arbitrario.
- `x in s`: pertenencia.
- `len(s)`: cardinalidad.

10.4.3. Operaciones de teoría de conjuntos

Python implementa de forma natural unión, intersección, diferencia y diferencia simétrica:

Operador	Método	Descripción
	<code>s1.union(s2)</code>	Unión: $s_1 \cup s_2$
&	<code>s1.intersection(s2)</code>	Intersección: $s_1 \cap s_2$
-	<code>s1.difference(s2)</code>	Diferencia: $s_1 \setminus s_2$
^	<code>s1.symmetric_difference(s2)</code>	Diferencia simétrica: $s_1 \Delta s_2$
<=	<code>s1.issubset(s2)</code>	$s_1 \subseteq s_2?$
>=	<code>s1.issuperset(s2)</code>	$s_1 \supseteq s_2?$

Tabla 2: Operaciones de teoría de conjuntos en Python.

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b)  # {1, 2, 3, 4, 5}
print(a & b)  # {3}
print(a - b)  # {1, 2}
print(a ^ b)  # {1, 2, 4, 5}
print({1, 2} <= a)  # True
```


10.4.4. Conjuntos por comprensión

Igual que con las listas, los conjuntos admiten comprensiones (con llaves):

```
cuadrados_pares = {x ** 2 for x in range(10) if x % 2 == 0}
# {0, 4, 16, 36, 64}

dominios = {url.split("/")[2] for url in lista_urls}
```

10.4.5. Iteración

Los conjuntos se iteran con `for`, pero el orden **no está garantizado**:

```
for elemento in {1, 2, 3, 4}:
    print(elemento)
```

10.4.6. Conjuntos inmutables: frozenset

Como un `set` es mutable, no es *hashable* y no puede ser elemento de otro `set` ni clave de un diccionario. Para esos casos existe `frozenset`, su variante inmutable:

```
a = frozenset([1, 2, 3])
b = frozenset([4, 5])
conjunto_de_conjuntos = {a, b}      # OK: frozensets si son
↪      hashables
```

10.5. Diccionarios

Un **diccionario** (tipo `dict`) es una colección **mutable** de pares **clave-valor**. Las claves deben ser *hashables* (inmutables); los valores pueden ser de cualquier tipo. La búsqueda por clave es $O(1)$ amortizado (tabla hash), lo que convierte al diccionario en la estructura por excelencia para asociaciones rápidas.

Desde Python 3.7, el orden de inserción se preserva en el orden de iteración (antes era un detalle de implementación, ahora forma parte de la especificación).

10.5.1. Creación y uso básico

```
diccionario_vacio = {}

persona = {
    "nombre": "Juan",
    "edad": 25,
    "profesion": "Ingeniero"
}
```

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

```
print(persona["nombre"])           # Juan
# print(persona["telefono"])       # KeyError
print(persona.get("telefono"))     # None (en lugar de error)
↪ )
↪ print(persona.get("telefono", "desconocido")) # 'desconocido'
```

Para modificar o añadir, simplemente se asigna a la clave. `update` permite mezclar varios pares de una vez:

```
persona["edad"] = 26
persona["direccion"] = "Madrid"

persona.update({"telefono": "912345678", "edad": 27})
```

Para eliminar elementos:

```
del persona["direccion"]           # Elimina por clave
profesion = persona.pop("profesion") # Elimina y devuelve el
↪ valor
↪ clave_valor = persona.popitem()    # Elimina y devuelve un
par (LIFO desde 3.7)
persona.clear()                    # Vacía el diccionario
```

10.5.2. Iteración y vistas

```
for clave in persona:               # Por defecto, itera
↪ sobre claves
    print(clave)

for valor in persona.values():
    print(valor)

for clave, valor in persona.items():
    print(clave, valor)
```

`keys()`, `values()` e `items()` devuelven *vistas* dinámicas: si el diccionario cambia, la vista refleja los cambios sin necesidad de reconstruirla. Las vistas son iterables y eficientes en memoria (no copian los datos).

10.5.3. Diccionarios anidados

Es habitual anidar diccionarios para representar relaciones jerárquicas:

```

empleado = {
    "nombre": "Laura",
    "informacion_personal": {"edad": 30, "direccion": "
    Barcelona"},
    "empleo": {"cargo": "Gerente", "salario": 5000}
}
print(empleado["informacion_personal"]["edad"]) # 30

```

10.5.4. Diccionarios por comprensión

```

cuadrados = {x: x ** 2 for x in range(5)}
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Invertir un diccionario
original = {"a": 1, "b": 2, "c": 3}
invertido = {v: k for k, v in original.items()}
# {1: 'a', 2: 'b', 3: 'c'}

```

10.5.5. Copia y referencias

`dict.copy()` y `list.copy()` devuelven **copias superficiales**: los objetos contenidos se comparten. Para una copia profunda (con todos los objetos anidados duplicados), usa `copy.deepcopy`:

```

import copy

original = {"datos": [1, 2, 3]}
shallow = original.copy()
deep = copy.deepcopy(original)

original["datos"].append(99)
print(shallow["datos"]) # [1, 2, 3, 99] <-- compartido
print(deep["datos"]) # [1, 2, 3] <-- aislado

```

10.5.6. El módulo collections

El módulo estándar `collections` ofrece variantes especializadas de los tipos compuestos. Las más útiles:

- **Counter**: diccionario optimizado para **contar** elementos. Acepta cualquier iterable y devuelve un diccionario {elemento: frecuencia}.
- **defaultdict**: diccionario que crea automáticamente un valor por defecto para claves nuevas, ahorrando el patrón `if k not in d: d[k] = []`.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

- `OrderedDict`: como `dict`, pero con métodos extra para reordenar. Hoy en día tiene poco uso, ya que el `dict` estándar preserva el orden.
- `deque`: cola doblemente terminada con `append/popleft` en $O(1)$. Ideal para colas y *buffers* circulares.

```
from collections import Counter

texto = "abracadabra"
frec = Counter(texto)
print(frec)                # Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
→ print(frec.most_common(3)) # [('a', 5), ('b', 2), ('r', 2)]
```

defaultdict en acción. Agrupar palabras por su primera letra:

```
from collections import defaultdict

palabras = ["manzana", "mango", "pera", "platano", "kiwi"]
grupos = defaultdict(list)
for p in palabras:
    grupos[p[0]].append(p)
print(dict(grupos))
→ # {'m': ['manzana', 'mango'], 'p': ['pera', 'platano'], 'k': ['kiwi']}
```

```
from collections import deque

cola = deque([1, 2, 3])
cola.append(4)          # [1, 2, 3, 4]
cola.appendleft(0)      # [0, 1, 2, 3, 4]
cola.popleft()          # 0, cola = [1, 2, 3, 4]
```

10.6. Funciones de primera clase y de orden superior

deque para colas eficientes. En Python, las funciones son **ciudadanos de primera clase** (*first-class citizens*): pueden asignarse a variables, pasarse como argumentos a otras funciones, devolverse desde otras funciones y almacenarse en estructuras de datos. Una función que acepta otras funciones como argumento o devuelve una función como resultado se denomina **de orden superior** (*higher-order function*, HOF).

Estos dos conceptos son la base del paradigma funcional, y aunque Python es multi-paradigma, ofrece soporte de primera clase para HOFs en su biblioteca estándar.

Vocabulario

Función de primera clase y *función de orden superior* son conceptos distintos: el primero describe que las funciones son **valores** en el lenguaje; el segundo, que una función **trabaja con** otras funciones. Por otro lado, una *función de primer orden* (en el sentido lógico-matemático) es una que **no** trabaja con funciones – justo lo opuesto a HOF – y no es lo que aquí nos interesa.

10.6.1. Funciones lambda

Una expresión `lambda` define una función anónima en una sola línea. La sintaxis es `lambda` `parametros`: `expresion`:

```
cuadrado = lambda x: x ** 2
print(cuadrado(5))    # 25

# Equivalente a:
def cuadrado(x):
    return x ** 2
```

Las lambdas son cómodas para usos puntuales (claves de ordenación, argumento de `map/filter`), pero para cualquier función no trivial conviene definirla con `def` para mejorar la legibilidad y poder añadir docstring.

10.6.2. map, filter, reduce

Las tres HOF clásicas del paradigma funcional:

`map(f, iter)`. Aplica la función `f` a cada elemento del iterable.

```
numeros = [1, 2, 3, 4]
print(list(map(lambda x: x ** 2, numeros))) # [1, 4, 9, 16]
```

A menudo, una lista por comprensión es más legible: `[x ** 2 for x in numeros]`.

`filter(p, iter)`. Conserva los elementos para los que `p` es `True`.

```
numeros = [1, 2, 3, 4, 5, 6]
print(list(filter(lambda x: x % 2 == 0, numeros))) # [2, 4, 6]
```

Equivalente: `[x for x in numeros if x % 2 == 0]`.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

`reduce(f, iter)`. Aplica `f` acumulativamente, reduciendo el iterable a un único valor. Vive en el módulo `functools`:

```
from functools import reduce

print(reduce(lambda x, y: x + y, [1, 2, 3, 4])) # 10 (=
↪      1+2+3+4)
```

↪ Para sumas, productos, máximos y mínimos prefiere las funciones nativas `sum`, `math.prod`, `max`, `min`: son más rápidas y más legibles.

10.6.3. `sorted` con clave

`sorted` acepta un parámetro `key` que es una función aplicada a cada elemento para extraer la clave de comparación, y `reverse` para invertir el orden:

```
palabras = ["manzana", "kiwi", "banana", "cereza"]

print(sorted(palabras)) # alfabetico
print(sorted(palabras, key=len)) # por longitud
print(sorted(palabras, key=str.lower, reverse=True)) #
↪      alfabetico desc., insensible a mayusculas
```

Para ordenar listas de tuplas, `key` permite extraer el campo deseado:

```
personas = [("Ana", 30), ("Juan", 25), ("Luis", 28)]
print(sorted(personas, key=lambda p: p[1])) # ordenado por
↪      edad
```

10.7. Aplicación: estructuras de datos en el análisis de registros

Como ejemplo de aplicación, veamos cómo las estructuras compuestas que hemos visto aparecen en el análisis de datos de seguridad, donde es habitual procesar grandes volúmenes de registros (*logs*) y catálogos de indicadores. Recogemos algunas aplicaciones típicas como aperitivo para los capítulos siguientes.

Análisis de frecuencias con `Counter`. Un primer paso para criptoanálisis (cifrados clásicos) o detección de patrones en logs es contar ocurrencias:

```
from collections import Counter

cifrado = "WKLV LV D VHFUHW PHVVDJH"
frec = Counter(c for c in cifrado if c.isalpha())
print(frec.most_common(5)) # candidatas a ser 'E' en el
↪      texto plano
```

Sets para deduplicación de IOC. Los *indicadores de compromiso* (*Indicators of Compromise*, IOCs) – IPs, hashes, dominios maliciosos – suelen llegar de varias fuentes con duplicados; un `set` los unifica en $O(n)$ y permite intersecciones rápidas:

```
ioc_fuente_a = {"1.2.3.4", "evil.com", "deadbeef..."}
ioc_fuente_b = {"5.6.7.8", "evil.com"}

todos      = ioc_fuente_a | ioc_fuente_b      # union
comunes    = ioc_fuente_a & ioc_fuente_b      # interseccion (
↪      alta confianza)
solo_en_a  = ioc_fuente_a - ioc_fuente_b
```

Diccionarios como tablas hash de búsqueda. Para asociar hashes a nombres de *malware*, IPs a metadatos, etc.:

```
firmas = {
    "44d88612fea8a8f36de82e1278abb02f": "Eicar test file",
    "275a021bbfb6489e54d471899f7db9d1": "Mimikatz",
}

def identificar(hash_md5):
    return firmas.get(hash_md5, "Desconocido")
```

```
from collections import defaultdict

eventos = [
    ("2026-01-12 10:31", "192.168.1.10", "login_failed"),
    ("2026-01-12 10:32", "192.168.1.10", "login_failed"),
    ("2026-01-12 10:33", "203.0.113.7", "port_scan"),
]

por_ip = defaultdict(list)
for ts, ip, tipo in eventos:
    por_ip[ip].append((ts, tipo))

# Detectar IPs con >= 5 fallos de login
sospechosas = {ip for ip, evs in por_ip.items()
↪               if sum(1 for _, t in evs if t == "login_failed") >= 5}
```

namedtuple y dataclass para registros estructurados. Para representar alertas, eventos o paquetes con campos nombrados, sin el peso de definir una clase completa:

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Alerta:
    timestamp: str
    severidad: int
    fuente: str
    mensaje: str

a = Alerta("2026-01-12 10:31", 7, "ids", "Posible escaneo de
↪   puertos")
print(a.severidad)
```

En el capítulo 12 veremos cómo **pandas** generaliza estas estructuras a tablas de datos completas (**DataFrame**) y **numpy** a vectores y matrices numéricas eficientes.

10.8. Ejercicios

EJERCICIO 10.1

Escribe un programa en Python que reciba un número entero n y calcule su factorización prima, es decir, descomponga n en los factores primos que lo multiplican. El programa debe manejar correctamente números grandes y ser capaz de devolver la lista de factores primos en orden ascendente.

Ejemplo de ejecución esperada

Introduce un número: 60

La factorización prima de 60 es: 2, 2, 3, 5

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

Solución (girar para leer)

```
1 def factorizar_prima(n):  
2     factores = []  
3  
4     # Dividir entre 2 hasta que no se pueda más  
5     while n % 2 == 0:  
6         factores.append(2)  
7         n //= 2  
8  
9     # Comprobar los posibles divisores impares a partir  
10    de 3  
11    divisor = 3  
12    while divisor * divisor <= n:  
13        while n % divisor == 0:  
14            factores.append(divisor)  
15            n //= divisor  
16            divisor += 2  
17    # Si n es mayor que 2, entonces es un número primo  
18    if n > 2:  
19        factores.append(n)  
20  
21    return factores  
22  
23    # Solicitar al usuario el número para factorizar  
24    n = int(input("Introduce un número: "))  
25  
26    # Calcular la factorización prima  
27    factores = factorizar_prima(n)  
28  
29    # Mostrar el resultado  
30    print(f"La factorización prima de {n} es: ", " ".join(  
31        map(str, factores)))
```

Código 10.1: Solución al Ejercicio 10.1.

EJERCICIO 10.2

Escribe un programa en Python que reciba una lista de números enteros y calcule la suma de la sublista contigua de mayor suma. Por ejemplo, dada la lista $[34, -50, 42, 14, -5, 86]$, el programa debe devolver el valor máximo de la suma de sublistas, que en este caso es $42 + 14 - 5 + 86 = 137$.

Ejemplo de ejecución esperada

Introduce la lista de números: $[34, -50, 42, 14, -5, 86]$

La suma de la sublista contigua máxima es: 137

Solución (girar para leer)

```

1 def suma_sublista_maxima(lista):
2     max_suma = lista[0] # Iniciar con el primer
3     suma_actual = lista[0] # Suma acumulada de la
4     sublista actual
5     for num in lista[1:]: # Comenzamos desde el
6         siguiente elemento
7         suma_actual = max(suma_actual + num,
8                             num) # Decidir si incluir el número en la sublista
9         # o empezar una nueva sublista
10        max_suma = max(suma_actual, max_suma) # Actualizar la suma máxima si es necesario
11    return max_suma
12
13 # Solicitar al usuario una lista de números enteros
14 entrada = input("Introduce la lista de números: ")
15 lista = eval(entrada) # Convertir la cadena de texto
16 # Calcular la suma de la sublista contigua de mayor
17 suma
18 resultado = suma_sublista_maxima(lista)
19 # Mostrar el resultado
20 print(f"La suma de la sublista contigua máxima es: {
21     resultado}")

```

Código 10.2: Solución al Ejercicio 10.2.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.3

Escribe un programa en Python que reciba una lista de diccionarios con información de personas (nombre y edad) y ordene la lista por edad de forma ascendente. El programa debe devolver la lista ordenada.

Ejemplo de ejecución esperada

Introduce la lista de personas: [{ 'nombre': 'Juan', 'edad': 25},
{ 'nombre': 'Ana', 'edad': 30}, { 'nombre': 'Luis', 'edad': 20}]
La lista ordenada por edad es: [{ 'nombre': 'Luis', 'edad': 20},
{ 'nombre': 'Juan', 'edad': 25}, { 'nombre': 'Ana', 'edad': 30}]

Solución (girar para leer)

```
1 def ordenar_personas_por_edad(personas):  
2     return sorted(personas, key=lambda x: x['edad'])  
3  
4     # Solicitar al usuario la lista de personas  
5     entrada = input("Introduce la lista de personas: ")  
6     personas = eval(entrada) # Convertir la cadena de  
7     texto de entrada en una lista de diccionarios  
8     # Ordenar la lista por edad  
9     personas_ordenadas = ordenar_personas_por_edad(personas)  
10  
11     # Mostrar el resultado  
12     print(f"La lista ordenada por edad es: {  
13         personas_ordenadas}")
```

Código 10.3: Solución al Ejercicio 10.3.

EJERCICIO 10.4

Escribe un programa en Python que reciba una lista con elementos duplicados y devuelva una lista sin duplicados, manteniendo el orden original de los elementos.

Ejemplo de ejecución esperada

Introduce la lista de elementos: [1, 2, 3, 2, 4, 1, 5]

La lista sin duplicados es: [1, 2, 3, 4, 5]

Solución (girar para leer)

```

1 def eliminar_duplicados(lista):
2     return list(dict.fromkeys(lista)) # también podrí
3     amos haberlo convertido a un set
4     # Solicitar al usuario la lista de elementos
5     entrada = input("Introduce la lista de elementos: ")
6     lista = eval(entrada) # Convertir la cadena de texto
7     de entrada en una lista
8     # Eliminar los duplicados
9     lista_sin_duplicados = eliminar_duplicados(lista)
10
11     # Mostrar el resultado
12     print(f"La lista sin duplicados es: {
13         lista_sin_duplicados}")

```

Código 10.4: Solución al Ejercicio 10.4.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.5

Escribe un programa en Python que reciba una lista de frases y calcule la frecuencia de las palabras en todas las frases combinadas, devolviendo un diccionario donde las claves sean las palabras y los valores su frecuencia.

Ejemplo de ejecución esperada

Introduce las frases: ['Hola mundo', 'Hola Python', 'Adiós mundo']

La frecuencia de las palabras es: 'Hola': 2, 'mundo': 2, 'Python': 1, 'Adiós': 1

Solución (girar para leer)

```
1 def contar_palabras(frases):
2     frecuencia = {}
3     for frase in frases:
4         palabras = frase.split()
5         for palabra in palabras:
6             if palabra in frecuencia:
7                 frecuencia[palabra] += 1
8             else:
9                 frecuencia[palabra] = 1
10    return frecuencia
11
12 # Solicitar al usuario las frases
13 entrada = input("Introduce las frases: ")
14 frases = eval(entrada) # Convertir la cadena de texto
15 # de entrada en una lista de frases
16 # Calcular la frecuencia de las palabras
17 frecuencia_palabras = contar_palabras(frases)
18
19 # Mostrar el resultado
20 print(f"La frecuencia de las palabras es: {
    frecuencia_palabras}")
```

Código 10.5: Solución al Ejercicio 10.5.

EJERCICIO 10.6

Escribe un programa en Python que reciba una lista de números enteros y devuelva una lista de todas las sublistas consecutivas cuyo producto sea mayor que un número dado k . El programa debe devolver la lista de sublistas ordenadas por su longitud en orden ascendente. Considera que el producto de una sublista es el producto de todos sus elementos.

Ejemplo de ejecución esperada

Introduce la lista de números: [1, 2, 3, 4, 5]

Introduce el valor de k : 6

Las sublistas con producto mayor que 6 son: [[3, 4], [4, 5], [2, 3, 4], [3, 4, 5], [1, 2, 3, 4], [2, 3, 4, 5], [1, 2, 3, 4, 5]]

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

Solución (girar para leer)

```

1 def sublistas_consecutivas(numeros, k):
2     # Función para calcular el producto de una sublista
3     def producto_sublista(sublista):
4         producto = 1
5         for num in sublista:
6             producto *= num
7         return producto
8
9     # Lista para almacenar las sublistas que cumplen la
10    condición
11    sublistas_validas = []
12    # Generar todas las sublistas consecutivas
13    for i in range(len(numeros)):
14        for j in range(i+1, len(numeros)+1):
15            sublista = numeros[i:j]
16            if producto_sublista(sublista) > k:
17                sublistas_validas.append(sublista)
18    return sublistas_validas
19
20 # Entrada de datos
21 numeros = list(map(int, input("Introduce la lista de nú
22 meros: ").split(',')))
23 k = int(input("Introduce el valor de k: "))
24 # Llamar a la función
25 sublistas_validas = sublistas_consecutivas(numeros, k)
26 # Mostrar el resultado
27 print(f"Las sublistas con producto mayor que {k} son: {
28 sorted(sublistas_validas, key=lambda x: len(x))}")

```

Código 10.6: Solución al Ejercicio 10.6.

EJERCICIO 10.7

Escribe un programa en Python que reciba un número entero n y genere la n -ésima fila del triángulo de Pascal. El triángulo de Pascal es una figura matemática en la que cada número es la suma de los dos números que se encuentran directamente encima de él en la fila anterior. El triángulo comienza con un 1 en la parte superior, y las filas siguientes se construyen siguiendo esta regla. Cada fila n -ésima del triángulo contiene $n + 1$ elementos.

Por ejemplo, las primeras filas del triángulo de Pascal son:

```
Fila 1: [1]
Fila 2: [1, 1]
Fila 3: [1, 2, 1]
Fila 4: [1, 3, 3, 1]
Fila 5: [1, 4, 6, 4, 1]
```

La fila debe devolverse como una lista de números, donde el primer número es siempre 1 y cada número siguiente es la suma de los dos números directamente encima de él en la fila anterior.

Ejemplo de ejecución esperada

```
Introduce el valor de n: 5
```

```
La fila 5 del triángulo de Pascal es: [1, 4, 6, 4, 1]
```

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

Solución (girar para leer)

```
1 def generar_fila_pascals(n):  
2     fila = [1]  
3     for k in range(1, n):  
4         # Generar el siguiente valor de la fila usando  
5         la fórmula del triángulo de Pascal  
6         siguiente_valor = fila[k - 1] * (n - k) // k  
7         fila.append(siguiente_valor)  
8     return fila  
9 # Solicitar al usuario el valor de n  
10 n = int(input("Introduce el valor de n: "))  
11  
12 # Generar la n-ésima fila del triángulo de Pascal  
13 fila_pascals = generar_fila_pascals(n)  
14  
15 # Mostrar el resultado  
16 print(f"La fila {n} del triángulo de Pascal es: {  
fila_pascals}")
```

Código 10.7: Solución al Ejercicio 10.7.

EJERCICIO 10.8

Escribe un programa en Python que reciba dos listas de números y genere todas las combinaciones posibles de elementos, donde cada combinación consiste en un elemento de cada lista. El programa debe ordenar las combinaciones por la suma de los elementos de cada combinación en orden descendente.

Ejemplo de ejecución esperada

```
Introduce la primera lista: [1, 4, 7]
```

```
Introduce la segunda lista: [2, 5, 8]
```

```
Combinaciones ordenadas por la suma de sus elementos en orden descendente:
```

```
(7, 8)
```

```
(4, 8)
```

```
(7, 5)
```

```
(1, 8)
```

```
(4, 5)
```

```
(7, 2)
```

```
(1, 5)
```

```
(4, 2)
```

```
(1, 2)
```

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

Solución (girar para leer)

```

1 # Paso 1: Recibir las dos listas de números
2 lista1 = list(map(int, input("Introduce la primera
3 lista: ").strip()[1:-1].split(', ')))
4 lista2 = list(map(int, input("Introduce la segunda
5 lista: ").strip()[1:-1].split(', ')))
6 # Paso 2: Generar todas las combinaciones posibles
7 entre los elementos de las dos listas
8 combinaciones = []
9 for i in lista1:
10 for j in lista2:
11 combinaciones.append((i, j))
12 # Paso 3: Ordenar las combinaciones por la suma de sus
13 elementos en orden descendente
14 combinaciones-ordenadas = sorted(combinaciones, key=
15 lambda x: sum(x), reverse=True)
16 # Paso 4: Imprimir las combinaciones ordenadas
17 print("\nCombinaciones ordenadas por la suma de sus
18 elementos en orden descendente:")
19 for combinacion in combinaciones-ordenadas:
20 print(combinacion)

```

EJERCICIO 10.9

Escribe un programa en Python que reciba una lista de enteros representando el peso de varios bloques. El objetivo es dividir la lista en dos sublistas de manera que la suma de los elementos de ambas sublistas sea lo más equilibrada posible. El programa debe devolver las dos sublistas que tienen la suma de los elementos más equilibrada.

Ejemplo de ejecución esperada

Introduce la lista de bloques: `[10, 20, 30, 40, 50]`

Las sublistas equilibradas son: `[10, 20, 40]` `[30, 50]`

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

Solución (girar para leer)

```

1 def obtener_sublistas_equilibradas(bloques):
2     # generar todas las particiones posibles de la
3     lista
4     mejor_sublista1 = []
5     mejor_sublista2 = []
6
7     for i in range(1, 2*len(bloques)): # generar
8         todas las combinaciones de particiones
9         sublista1, sublista2 = particionar_lista(
10            bloques, i)
11         diferencia = abs(sum(sublista1) - sum(sublista2))
12         # calcular la diferencia de las sumas
13         if diferencia < mejor_diferencia: # Comparar
14             la diferencia con la mejor encontrada
15             mejor_diferencia = diferencia
16             mejor_sublista1, mejor_sublista2 =
17             sublista1, sublista2
18         return mejor_sublista1, mejor_sublista2
19
20 def particionar_lista(bloques, mascara):
21     sublista1 = []
22     sublista2 = []
23     for j in range(len(bloques)): # Dividir los
24         elementos según la mascara
25         if (mascara >> j) & 1:
26             sublista1.append(bloques[j])
27         else:
28             sublista2.append(bloques[j])
29     return sublista1, sublista2
30
31 def introducir_lista_de_bloques: """Introduce la lista de
32     bloques = list(map(int, input("Introduce la lista de
33     bloques: ").strip()[1:-1].split(','))) # Entrada
34     sublista1, sublista2 = obtener_sublistas_equilibradas(
35     bloques) # obtener las sublistas equilibradas
36     print("Las sublistas equilibradas son:", sublista1,
37     sublista2)

```

Código 10.9: Solución al Ejercicio 10.9.

EJERCICIO 10.10

Escribe un programa en Python que reciba una lista de números enteros y divida la lista en dos sublistas: una con los números pares y otra con los números impares. El programa debe luego devolver la suma de los números de cada sublista y mostrar ambas sumas.

Ejemplo de ejecución esperada

```
Introduce la lista de números: [1, 2, 3, 4, 5, 6, 7, 8]  
La suma de los números pares es: 20  
La suma de los números impares es: 16
```

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

Solución (girar para leer)

```
1 def separar_pares_impares(lista):  
2     pares = []  
3     impares = []  
4  
5     # Separar los números en pares e impares  
6     for num in lista:  
7         if num % 2 == 0:  
8             pares.append(num)  
9         else:  
10            impares.append(num)  
11  
12     return pares, impares  
13  
14 def suma_lista(lista):  
15     return sum(lista)  
16  
17 # Paso 1: Recibir la lista de números  
18 lista = list(map(int, input("Introduce la lista de nú  
19     ↵  
20     meros: ").strip()[1:-1].split(',')))  
21  
22 # Paso 2: Separar la lista en pares e impares  
23 pares, impares = separar_pares_impares(lista)  
24  
25 # Paso 3: Calcular la suma de los números pares e  
26     ↵  
27     impares  
28 suma_pares = suma_lista(pares)  
29 suma_impares = suma_lista(impares)  
30  
31 # Paso 4: Mostrar los resultados  
32 print(f"La suma de los números pares es: {suma_pares}")  
33 print(f"La suma de los números impares es: {  
34     ↵  
35     suma_impares}")
```

Código 10.10: Solución al Ejercicio 10.10.

EJERCICIO 10.11

Escribe un programa en Python que reciba una lista de palabras separadas por comas y agrupe las palabras que comienzan con la misma letra. El programa debe devolver un diccionario donde las claves sean las primeras letras de las palabras y los valores sean listas de palabras que comienzan con esa letra.

Ejemplo de ejecución esperada

Introduce la lista de palabras: manzana, mango, pera, plátano, kiwi, melon
 El diccionario de agrupación es: {'m': ['manzana', 'mango', 'melon'], 'p': ['pera', 'plátano'], 'k': ['kiwi']}

Solución (girar para leer)

```

1  def agrupar_palabras(palabras):
2      diccionario = {}
3      for palabra in palabras:
4          letra_inicial = palabra[0].lower()
5          if letra_inicial not in diccionario:
6              diccionario[letra_inicial] = []
7          diccionario[letra_inicial].append(palabra)
8      return diccionario
9
10 palabras = input("Introduce la lista de palabras: ")
11 ← strip().split(',')
12 diccionario = agrupar_palabras(palabras)
13 print("El diccionario de agrupación es:", diccionario)

```

Código 10.11: Solución al Ejercicio 10.11.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.12

Escribe un programa en Python que reciba una cadena de texto y cuente la cantidad de veces que cada carácter aparece en la cadena. El programa debe devolver un diccionario donde las claves sean los caracteres y los valores sean su frecuencia de aparición.

Ejemplo de ejecución esperada

Introduce una cadena de texto: `hola mundo`

El diccionario de frecuencias es: `{ 'h': 1, 'o': 2, 'l': 1, 'a': 1, ' ' : 1, 'm': 1, 'u': 1, 'n': 1, 'd': 1 }`

Solución (girar para leer)

```
12 print("El diccionario de frecuencias es:", diccionario)
11 diccionario = contar_caracteres(cadena)
10 cadena = input("Introduce una cadena de texto: ")
9
8 return diccionario
7 diccionario[caracter] = 1
6 else:
5 diccionario[caracter] += 1
4 if caracter in diccionario:
3 for caracter in cadena:
2 diccionario = {}
1 def contar_caracteres(cadena):
```

Código 10.12: Solución al Ejercicio 10.12.

EJERCICIO 10.13

Escribe un programa en Python que reciba un diccionario de empleados con su salario y un porcentaje de aumento. El programa debe aplicar el aumento al salario de cada empleado y devolver el diccionario con los salarios actualizados.

Ejemplo de ejecución esperada

Introduce el diccionario de empleados y salarios: {'Juan': 2000, 'Ana': 3000, 'Luis': 2500}

Introduce el porcentaje de aumento: 10

El diccionario actualizado es: {'Juan': 2200.0, 'Ana': 3300.0, 'Luis': 2750.0}

Solución (girar para leer)

```

1 def actualizar_salarios(diccionario, aumento):
2     for empleado in diccionario:
3         diccionario[empleado] += diccionario[empleado] * (aumento / 100)
4     return diccionario
5
6 diccionario = eval(input("Introduce el diccionario de
7 empleados y salarios: "))
8 aumento = float(input("Introduce el porcentaje de
9 aumento: "))
10 diccionario_actualizado = actualizar_salarios(
11     diccionario, aumento)
12 print("El diccionario actualizado es:",
13     diccionario_actualizado)

```

Código 10.13: Solución al Ejercicio 10.13.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.14

Escribe un programa en Python que reciba un diccionario de claves y valores y cree un nuevo diccionario donde las claves sean los valores del diccionario original y los valores sean las claves. Si hay claves duplicadas en el diccionario original, conserva la última.

Ejemplo de ejecución esperada

Introduce el diccionario: {'a': 1, 'b': 2, 'c': 3, 'd': 2}
El diccionario invertido es: {1: 'a', 2: 'd', 3: 'c'}

Solución (girar para leer)

```
1 def invertir_diccionario(diccionario):  
2     diccionario_invertido = {}  
3     for clave, valor in diccionario.items():  
4         diccionario_invertido[valor] = clave  
5     return diccionario_invertido  
6  
7 diccionario = eval(input("Introduce el diccionario: "))  
8 diccionario_invertido = invertir_diccionario(diccionario)  
9  
10 print("El diccionario invertido es:",  
11       invertir_diccionario(diccionario))
```

Código 10.14: Solución al Ejercicio 10.14.

EJERCICIO 10.15

Escribe un programa en Python que reciba un diccionario con nombres de estudiantes y sus notas, y filtre los estudiantes que tienen una calificación mayor o igual a 6. El programa debe devolver un diccionario solo con los estudiantes aprobados.

Ejemplo de ejecución esperada

Introduce el diccionario de estudiantes y notas: {'Juan': 5, 'Ana': 8, 'Luis': 7}

El diccionario de estudiantes aprobados es: {'Ana': 8, 'Luis': 7}

Solución (girar para leer)

```

1 def filtrar_estudiantes(diccionario):
2     diccionario_aprobados = {}
3     for estudiante, nota in diccionario.items():
4         if nota >= 6:
5             return diccionario_aprobados
6     return diccionario_aprobados
7     print("El diccionario de estudiantes aprobados es: ",
8         diccionario_aprobados)

```

Código 10.15: Solución al Ejercicio 10.15.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.16

Escribe un programa en Python que reciba una lista de tuplas, donde cada tupla contiene un número entero y una cadena. El programa debe buscar la tupla que tenga el mayor número entero y devolver esa tupla.

Ejemplo de ejecución esperada

Introduce la lista de tuplas: [(1, "a"), (4, "b"), (2, "c")]
La tupla con el mayor número entero es: (4, "b")

Solución (girar para leer)

```
1 def buscar_maximo(lista_tuplas):  
2     max_tupla = max(lista_tuplas, key=lambda x: x[0])  
3     return max_tupla  
4  
5 lista_tuplas = eval(input("Introduce la lista de tuplas  
6     : "))  
7     max_tupla = buscar_maximo(lista_tuplas)  
8     print("La tupla con el mayor número entero es:",  
9         max_tupla)
```

Código 10.16: Solución al Ejercicio 10.16.

EJERCICIO 10.17

Escribe un programa en Python que reciba una lista de tuplas, donde cada tupla contiene dos números. El programa debe devolver una nueva lista con las tuplas que tienen el primer número mayor que el segundo.

Ejemplo de ejecución esperada

Introduce la lista de tuplas: [(5, 3), (2, 8), (7, 6), (1, 4)]
 Las tuplas con el primer número mayor que el segundo son: [(5, 3), (7, 6)]

Solución (girar para leer)

```

1  def filtrar_tuplas(lista_tuplas):
2      # Filtra las tuplas donde el primer número es mayor
3      # que el segundo
4      return [tupla for tupla in lista_tuplas if tupla[0] > tupla[1]]
5      # Recibir la entrada del usuario y convertirla en una
6      # lista de tuplas
7      entrada = input("Introduce la lista de tuplas: ")
8      # Convertir la cadena de entrada en una lista de
9      # tuplas de manera segura
10     lista_tuplas = eval(entrada) # Usamos eval solo
11     porque es una entrada controlada por el usuario
12     if isinstance(t, tuple) and len(t) == 2 for t
13     in lista_tuplas): # Verificar que todas las
14     entradas son tuplas de 2 elementos
15     tuplas_filtradas = filtrar_tuplas(lista_tuplas)
16     print("Las tuplas con el primer número mayor
17     que el segundo son:", tuplas_filtradas)
18     else:
19         print("La entrada no es válida. Debes ingresar
20         una lista de tuplas de dos elementos.")
21     except:
22         print("Hubo un error en la entrada. Asegúrate de
23         que la lista esté bien formada.")

```

Código 10.17: Solución al Ejercicio 10.17.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.18

Escribe un programa en Python que reciba una lista de tuplas, donde cada tupla contiene dos números. El programa debe devolver una lista con las sumas de los elementos de cada tupla.

Ejemplo de ejecución esperada

Introduce la lista de tuplas: [(1, 2), (3, 4), (5, 6)]
Las sumas de las tuplas son: [3, 7, 11]

Solución (girar para leer)

```
1 def sumar_tuplas(lista_tuplas):  
2     return [sum(tupla) for tupla in lista_tuplas]  
3  
4 lista_tuplas = eval(input("Introduce la lista de tuplas  
5 : "))  
6 sumas = sumar_tuplas(lista_tuplas)  
7 print("Las sumas de las tuplas son:", sumas)
```

Código 10.18: Solución al Ejercicio 10.18.

EJERCICIO 10.19

Escribe un programa en Python que reciba una lista de tuplas, donde cada tupla contiene dos números. El programa debe ordenar las tuplas por el primer número de cada tupla en orden ascendente y devolver la lista ordenada.

Ejemplo de ejecución esperada

Introduce la lista de tuplas: [(3, 4), (1, 2), (5, 6), (2, 3)]
 La lista ordenada por el primer número es: [(1, 2), (2, 3), (3, 4), (5, 6)]

Solución (girar para leer)

```

1 def ordenar_tuplas(lista_tuplas):
2     return sorted(lista_tuplas, key=lambda x: x[0])
3
4 lista_tuplas = eval(input("Introduce la lista de tuplas
5                               : "))
6 lista_ordenada = ordenar_tuplas(lista_tuplas)
7 print("La lista ordenada por el primer número es:",
8       lista_ordenada)
```

Código 10.19: Solución al Ejercicio 10.19.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.20

Escribe un programa en Python que reciba dos listas de tuplas, donde cada tupla contiene dos números. El programa debe encontrar las tuplas comunes entre ambas listas, pero no sólo comparando las tuplas directamente, sino verificando si la suma de los elementos de una tupla de la primera lista es igual a la suma de los elementos de una tupla de la segunda lista. El programa debe devolver una lista de las tuplas comunes encontradas.

Ejemplo de ejecución esperada

Introduce la primera lista de tuplas: [(1, 2), (3, 4), (5, 6)]

Introduce la segunda lista de tuplas: [(2, 1), (4, 3), (6, 5)]

Las tuplas comunes por la suma de sus elementos son: [(1, 2), (3, 4), (5, 6)]

Solución (girar para leer)

```

1  def interseccion-por-suma(lista1, lista2):
2      # Crear un diccionario para almacenar las tuplas de
3      lista2 con la suma de sus elementos como clave
4      suma_dict = {}
5      for tupla in lista2:
6          suma_dict[tupla] = suma_dict.get(tupla, 0) + 1
7
8      # Filtrar las tuplas de lista1 que tienen la misma
9      suma que las de lista2
10     comunes = []
11     for tupla in lista1:
12         if suma_dict.get(tupla, 0) > 0:
13             comunes.append(tupla)
14             suma_dict[tupla] -= 1
15
16     return comunes
17
18 # Recibir las listas de tuplas
19 lista1 = eval(input("Introduce la primera lista de
20 tuplas: "))
21 lista2 = eval(input("Introduce la segunda lista de
22 tuplas: "))
23 # Obtener las tuplas comunes por la suma de sus
24 elementos
25 comunes = interseccion-por-suma(lista1, lista2)
26 # Mostrar el resultado
27 print("Las tuplas comunes por la suma de sus elementos
28 son:", comunes)

```

Código 10.20: Solución al Ejercicio 10.20.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.21

Escribe un programa en Python que reciba dos conjuntos de números enteros y determine si el primer conjunto es un subconjunto del segundo. El programa debe devolver `True` si el primer conjunto es un subconjunto del segundo, de lo contrario debe devolver `False`.

Ejemplo de ejecución esperada

```
Introduce el primer conjunto: {1, 2, 3}
Introduce el segundo conjunto: {1, 2, 3, 4, 5}
¿Es el primer conjunto un subconjunto del segundo? True
```

Solución (girar para leer)

```
1 def es_subconjunto(conjunto1, conjunto2):
2     # Verificar si conjunto1 es un subconjunto de
3     conjunto1.issubset(conjunto2)
4     # Recibir los conjuntos del usuario
5     conjunto1 = set(map(int, input("Introduce el primer
6     conjunto: ").split(' ')))
7     conjunto2 = set(map(int, input("Introduce el segundo
8     conjunto: ").split(' ')))
9     # Verificar si es un subconjunto del
10    print(f"¿Es el primer conjunto un subconjunto del
11    segundo? {es_subconjunto(conjunto1, conjunto2)}")
```

Código 10.21: Solución al Ejercicio 10.21.

EJERCICIO 10.22

Escribe un programa en Python que reciba una lista de conjuntos de números enteros y devuelva un conjunto con los elementos que aparecen sólo en un único conjunto de la lista (es decir, elementos únicos en comparación con los demás conjuntos).

Ejemplo de ejecución esperada

Introduce la lista de conjuntos: [{1, 2, 3}, {3, 4, 5}, {5, 6, 7}]
 Los elementos únicos son: {1, 2, 4, 6, 7}

Solución (girar para leer)

```

1 def elementos_unicos(conjuntos):
2     # Crear un conjunto para almacenar los elementos u
3     nicos
4     todos = set()
5     comunes = set()
6     # Contamos cuántas veces aparece cada elemento en
7     los conjuntos
8     for conjunto in conjuntos:
9         for elem in conjunto:
10            if elem not in todos:
11                todos.add(elem)
12            elif elem in comunes:
13                comunes.add(elem)
14            # Los elementos únicos son aquellos que están en
15            todos, pero no en comunes
16            return todos - comunes
17        # Recibir la entrada de los conjuntos
18        conjuntos = eval(input("Introduce la lista de conjuntos
19        : "))
20        # Obtener los elementos únicos
21        elementos_unicos = elementos_unicos(conjuntos)
22        # Mostrar el resultado
23        print("Los elementos únicos son:", elementos)

```

Código 10.22: Solución al Ejercicio 10.22.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.23

Escribe un programa en Python que reciba dos conjuntos de números enteros y calcule la suma de todos los elementos de ambos conjuntos sin duplicados. El programa debe devolver el conjunto de los elementos únicos y la suma total.

Ejemplo de ejecución esperada

```
Introduce el primer conjunto: [1, 2, 3, 4]
Introduce el segundo conjunto: [3, 4, 5, 6]
El conjunto de elementos únicos es: {1, 2, 3, 4, 5, 6}
La suma total es: 21
```

Solución (girar para leer)

```
1 def suma_unica(conjunto1, conjunto2):
2     conjunto_unico = conjunto1 | conjunto2
3     return conjunto_unico
4
5 conjunto1 = set(map(int, input("Introduce el primer
6     conjunto: ").split(' ', 1)))
7 conjunto2 = set(map(int, input("Introduce el segundo
8     conjunto de elementos únicos: ").split(' ', 1)))
9     print(f"El conjunto de elementos únicos es: {conjunto_unico}")
10    print(f"La suma total es: {suma}")
```

Código 10.23: Solución al Ejercicio 10.23.

EJERCICIO 10.24

Escribe un programa en Python que reciba una lista de conjuntos de números enteros y devuelva un conjunto que contenga los elementos que son comunes a todos los conjuntos de la lista.

Ejemplo de ejecución esperada

Introduce la lista de conjuntos: `[{1, 2, 3}, {3, 4, 5}, {3, 6, 7}]`
 Los elementos comunes a todos los conjuntos son: `{3}`

Solución (girar para leer)

```

1 def elementos_comunes_a_todos(conjuntos):
2     # Inicializar el conjunto con los elementos del
3     # primer conjunto de la lista
4     comunes = conjuntos[0]
5     # Iterar sobre los demás conjuntos y obtener la
6     # intersección
7     for conjunto in conjuntos[1:]:
8         comunes = comunes.intersection(conjunto)
9     return comunes
10
11 # Recibir la lista de conjuntos del usuario
12 conjuntos = eval(input("Introduce la lista de conjuntos
13 : "))
14 # Obtener los elementos comunes
15 comunes = elementos_comunes_a_todos(conjuntos)
16 # Mostrar el resultado
17 print("Los elementos comunes a todos los conjuntos son:
18     ", comunes)

```

Código 10.24: Solución al Ejercicio 10.24.

10. Listas, diccionarios, tuplas y conjuntos. Funciones de primera clase y orden superior

EJERCICIO 10.25

Escribe un programa en Python que reciba dos conjuntos de números enteros y calcule el índice de similitud entre ellos, el cual se define como la razón entre el tamaño de la intersección de los conjuntos y el tamaño de la unión de los conjuntos. El programa debe de mostrar el índice de similitud con una precisión de seis decimales.

Ejemplo de ejecución esperada

```
Introduce el primer conjunto: [1, 2, 3, 4]
Introduce el segundo conjunto: [3, 4, 5, 6]
El índice de similitud es: 0.333333
```

Solución (girar para leer)

```
1 def indice_similitud(conjunto1, conjunto2):
2     # Calcular la intersección y la unión de los
3     interseccion = conjunto1 & conjunto2
4     union = conjunto1 | conjunto2
5
6     # Calcular el índice de similitud como la razón
7     # entre la intersección y la unión
8     return len(interseccion) / len(union)
9
10 # Recibir los conjuntos del usuario
11 conjunto1 = set(map(int, input("Introduce el primer
12 conjunto: ").strip()[1:-1].split(' ', 1)))
13 conjunto2 = set(map(int, input("Introduce el segundo
14 conjunto: ").strip()[1:-1].split(' ', 1)))
15 # Calcular el índice de similitud
16 print(f"El índice de similitud es: {indice_similitud(conjunto1, conjunto2):.6f}")
```

Código 10.25: Solución al Ejercicio 10.25.

EJERCICIO 10.26

Escribe un programa en Python que reciba dos conjuntos de números enteros y calcule el porcentaje de diferencia entre ellos, que se define como la razón entre el tamaño de la diferencia simétrica y el tamaño de la unión de los conjuntos, multiplicado por 100, mostrándolo con una precisión de cuatro decimales.

Ejemplo de ejecución esperada

Introduce el primer conjunto: [1, 2, 3, 4]
 Introduce el segundo conjunto: [3, 4, 5, 6]
 El porcentaje de diferencia es: 66.6667

Solución (girar para leer)

```

1 def porcentaje_diferencia(conjunto1, conjunto2):
2     # Calcular la diferencia simétrica (elementos que
3     # están en uno de los conjuntos pero no en ambos)
4     # Calcular la unión de los conjuntos
5     union = conjunto1 | conjunto2
6
7     # Calcular el porcentaje de diferencia
8     if len(union) == 0: # Evitar división por cero si
9         # los conjuntos están vacíos
10        return 0.0
11
12    # Recibir los conjuntos del usuario
13    conjunto1 = set(map(int, input("Introduce el primer
14    conjunto: ").strip().split(' ')))
15    conjunto2 = set(map(int, input("Introduce el segundo
16    conjunto: ").strip().split(' ')))
17    # Calcular el porcentaje de diferencia
18    diferencia = porcentaje_diferencia(conjunto1, conjunto2)
19    # Mostrar el resultado con una precisión de cuatro
20    # decimales
21    print(f"El porcentaje de diferencia es: {diferencia:.4f}
  ")

```

Código 10.26: Solución al Ejercicio 10.26.

11

Entrada y salida de datos. Ficheros

En este capítulo veremos cómo Python interactúa con el mundo exterior: leer del usuario, escribir en pantalla, manejar ficheros locales en formatos de texto y binarios, y trabajar con representaciones estructuradas como JSON y CSV. Es un capítulo transversal: la mayor parte de los programas de tratamiento de datos (analizadores de *logs*, procesadores de tablas, conversores de formatos, gestores de configuración) consisten en gran medida en leer ficheros, transformarlos y escribir resultados. La ciberseguridad es uno de los dominios donde estas tareas resultan especialmente frecuentes, y la usaremos como ejemplo de aplicación en varios casos. Como el fichero suele ser el primer punto de contacto del programa con datos cuyo origen no siempre controlamos, dedicaremos también atención a las trampas más habituales y a las buenas prácticas para tratar datos de forma robusta: `eval`, `pickle`, *path traversal* y la gestión de ficheros temporales y permisos.

11.1. Entrada y salida estándar

Las funciones más usadas para interactuar con la consola son `input()` y `print()`. `input()` lee una línea del teclado y la devuelve como cadena (sin el salto de línea final); si necesitamos otro tipo, hay que convertir explícitamente con `int()`, `float()`, etc. `print()` imprime cualquier número de argumentos y admite parámetros como `sep` (separador entre argumentos), `end` (carácter final) y `file` (destino, por defecto `sys.stdout`).

```
nombre = input("Introduce tu nombre: ")
edad = int(input("Introduce tu edad: "))
print(f"Hola, {nombre}. Tienes {edad} años.")
```

```
# Personalizar separador y final
print("a", "b", "c", sep="-", end="!\n") # 'a-b-c!'

# Escribir en stderr
import sys
print("error", file=sys.stderr)
```

↪ A bajo nivel, los flujos estándar son los objetos `sys.stdin`, `sys.stdout` y `sys.stderr`. Trabajamos con ellos cuando, por ejemplo, queremos componer pipelines en línea de comandos.

11.2. La función `eval` y la alternativa segura

`eval(s)` evalúa la cadena `s` como una expresión Python y devuelve su resultado. Es muy potente y, justamente por eso, **muy peligrosa**.

```
print(eval("2 + 3 * 4")) # 14

x = 10
print(eval("x + 5")) # 15
```

Nunca uses `eval` con datos no confiables

`eval` ejecuta **cualquier** expresión Python que se le pase. Si la cadena viene del usuario, de la red, de un fichero o de una base de datos, un atacante puede convertir tu programa en una shell:

```
# Cadena maliciosa que un atacante podría suministrar:
entrada = "__import__('os').system('rm -rf /')"
eval(entrada) # Ejecuta el comando del sistema
```

Si lo que necesitas es interpretar literales (números, listas, diccionarios, cadenas, tuplas, booleanos, `None`), usa `ast.literal_eval`: solo acepta literales y **nunca** ejecuta código arbitrario.

```
import ast

print(ast.literal_eval("[1, 2, 3]")) # [1, 2, 3]
print(ast.literal_eval("{'a': 1}")) # {'a': 1}
ast.literal_eval("__import__('os')") # ValueError:
malformed node
```

Si lo que necesitas es evaluar fórmulas matemáticas con seguridad, considera bibliotecas como `sympy` o restringe el conjunto de operadores que admites con tu propio parser.

11.3. Apertura de ficheros: *open*, modos y codificación

La función `open(path, mode, encoding=...)` abre un fichero y devuelve un objeto sobre el que se realizan las operaciones de lectura/escritura. Los **modos** más utilizados se recogen en la Tabla 1.

Modo	Descripción
'r'	Lectura (por defecto). Falla si no existe.
'w'	Escritura. Trunca el fichero si existe.
'a'	Anexar al final. Crea el fichero si no existe.
'x'	Crear de forma exclusiva : falla si existe.
'r'+	Lectura y escritura.
'b'	Sufijo: modo binario (ej. 'rb', 'wb').
't'	Sufijo: modo texto (por defecto).

Tabla 1: Modos de apertura de ficheros en Python.

11.3.1. Codificación: el parámetro `encoding`

En modo texto, Python decodifica los bytes del fichero a `str` usando una **codificación**. Si no la especificas, Python usa la del sistema, que en macOS y Linux suele ser UTF-8 pero en Windows puede ser CP-1252 u otra. Esto es una fuente recurrente de *bugs* de portabilidad y de *bugs* con caracteres no ASCII.

Especifica siempre `encoding` en modo texto

Salvo que tengas un motivo concreto, usa `encoding="utf-8"` tanto para leer como para escribir:

```
with open("datos.txt", "r", encoding="utf-8") as f:
    contenido = f.read()
```

Si lees datos cuya codificación no controlas, el parámetro `errors` permite definir el comportamiento ante bytes inválidos: `"strict"` (lanza excepción, por defecto), `"replace"` (sustituye por ?), `"ignore"` (descarta). Usar `"ignore"` en datos potencialmente maliciosos es peligroso: puede ocultar contenido o ataques.

11.4. Rutas: `pathlib` y `os.path`

Para manipular nombres de ficheros y directorios, Python ofrece dos APIs:

- `os.path`: API tradicional, basada en cadenas. Sigue siendo común y compatible.

- `pathlib` (Python 3.4+): API orientada a objetos, más legible y portable. Es la opción recomendada hoy.

```
from pathlib import Path

raiz = Path.home() / "datos"          # ~/datos (operador /
↪   como composicion)
fichero = raiz / "logs" / "abr.log"

print(fichero.exists())               # True/False
print(fichero.suffix)                 # '.log'
print(fichero.stem)                   # 'abr'
print(fichero.parent)                 # Path('~ / datos / logs')
↪ print(fichero.read_text(encoding="utf-8")) # Lectura corta
    y comoda

raiz.mkdir(parents=True, exist_ok=True) # mkdir -p
↪ for f in raiz.glob("*.log"):          # Pattern
    print(f.name)
```

`pathlib` es portable: el operador `/` usa el separador correcto en cada sistema (Linux/macOS `/`, Windows `\`).

11.5. Ficheros de texto

11.5.1. Lectura

↪ Los métodos básicos sobre un fichero abierto en modo texto son `f.read([n])`, `f.readline()` y `f.readlines()`, pero la forma más *pythonica* de leer línea a línea es iterar directamente sobre el objeto fichero, lo que evita cargar todo el contenido en memoria:

```
with open("archivo.txt", "r", encoding="utf-8") as f:
    contenido = f.read()              # Lee todo de golpe (cuidado si
↪   es grande)

with open("archivo.txt", "r", encoding="utf-8") as f:
    for linea in f:                   # Iteracion linea a linea,
↪   eficiente
        print(linea, end="")
```

11.5.2. Escritura

Para escribir, se utiliza `f.write(s)` (o `f.writelines(iter)`). El modo `'w'` **trunca** el fichero antes de escribir; el modo `'a'` añade al final.

```
with open("salida.txt", "w", encoding="utf-8") as f:
    f.write("Primera línea.\n")
    f.write("Segunda línea.\n")
```

11.5.3. El bloque `with` y el cierre automático

El bloque `with` es la forma recomendada de manejar ficheros: garantiza que el fichero se cierra al salir del bloque, **incluso si ocurre una excepción**. Sin `with`, habría que llamar manualmente a `f.close()` y envolver todo en un `try/finally` para garantizar el cierre. `with` es más corto y más seguro.

11.5.4. Manejo de errores

Las operaciones con ficheros pueden fallar por muchas razones: el fichero no existe, no tenemos permisos, el disco está lleno, etc. Las excepciones más comunes son `FileNotFoundError`, `PermissionError` e `IsADirectoryError`, todas subclases de `OSError`. Conviene capturarlas explícitamente:

```
try:
    with open("datos.txt", "r", encoding="utf-8") as f:
        contenido = f.read()
except FileNotFoundError:
    print("Error: el archivo no existe.")
except PermissionError:
    print("Error: sin permisos para leer el archivo.")
except OSError as e:
    print(f"Error de E/S: {e}")
```

11.6. Formatos comunes: JSON y CSV

11.6.1. JSON

JSON (*JavaScript Object Notation*) es un formato de texto para intercambio de datos basado en pares clave-valor. Se mapea de forma natural a los tipos básicos de Python: objetos → `dict`, arrays → `list`, strings → `str`, etc. El módulo estándar `json` ofrece cuatro funciones principales:

- `json.dump(obj, f)`: serializa `obj` y lo escribe en el fichero `f`.
- `json.load(f)`: lee del fichero `f` y devuelve el objeto.
- `json.dumps(obj)`: como `dump`, pero devuelve la cadena.
- `json.loads(s)`: como `load`, pero parsea la cadena `s`.

```
1 import json
2
3 datos = {"nombre": "Juan", "edad": 30, "tags": ["admin", "
↪ user"]}
4
5 with open("datos.json", "w", encoding="utf-8") as f:
6     json.dump(datos, f, indent=2, ensure_ascii=False,
↪     sort_keys=True)
7
8 with open("datos.json", "r", encoding="utf-8") as f:
9     cargado = json.load(f)
10    print(cargado)
```

Tres parámetros que conviene conocer:

- `indent=2`: formatea con sangría humanamente legible.
- `ensure_ascii=False`: permite escribir caracteres no ASCII (“ñ”, acentos, emoji) tal cual, sin escaparlos como `\uXXXX`.
- `sort_keys=True`: ordena las claves; útil para diff, control de versiones y hashes reproducibles.

11.6.2. CSV y TSV con `csv`

CSV (*Comma-Separated Values*) y TSV (*Tab-Separated Values*) son ficheros de texto donde cada línea representa una fila y las celdas se separan por un delimitador. Para leerlos y escribirlos correctamente – manejando comillas, escapes, saltos de línea dentro de campos, etc. – conviene usar el módulo estándar `csv`, no hacerlo a mano.

```
1 import csv
2
3 # Escribir
4 with open("salida.csv", "w", newline="", encoding="utf-8") as
↪ f:
5     writer = csv.writer(f)
6     writer.writerow(["Nombre", "Edad"])
7     writer.writerow(["Juan", 30])
8     writer.writerow(["Ana", 25])
9
10 # Leer
11 with open("salida.csv", "r", newline="", encoding="utf-8") as
↪ f:
12     reader = csv.reader(f)
13     cabecera = next(reader)
14     for fila in reader:
15         print(fila)
```


Para trabajar con CSV con cabecera, la API basada en diccionarios es más cómoda:

```
import csv

# DictReader: cada fila es un dict {columna: valor}
with open("usuarios.csv", "r", newline="", encoding="utf-8")
↪ as f:
    for fila in csv.DictReader(f):
        print(fila["Nombre"], fila["Edad"])

# DictWriter
↪ with open("salida.csv", "w", newline="", encoding="utf-8") as
    f:
        writer = csv.DictWriter(f, fieldnames=["Nombre", "Edad"])
        writer.writeheader()
        writer.writerow({"Nombre": "Juan", "Edad": 30})
```

El parámetro `newline=""`

Al abrir ficheros CSV, conviene pasar `newline=""`. Internamente, el módulo `csv` maneja los saltos de línea (que pueden estar dentro de campos entre comillas) y deshabilitar la conversión universal de `open` evita que se dupliquen o pierdan saltos en Windows.

11.6.3. Ejemplo: gestión de contactos

A continuación se ilustra todo lo visto desarrollando un programa que mantiene un fichero de contactos. Dos opciones: añadir registros nuevos o leer y mostrar el contenido. Cada contacto se guarda en una línea con el formato "`<nombre>`, `<teléfono>`, `<email>`
↪ `>`". Se manejan los errores de fichero inexistente y de entrada inválida.

Código 11.1: Código de ejemplo: gestión de contactos.

```
1 import os
2
3 ARCHIVO_CONTACTOS = "contactos.txt"
4
```

```
5 def guardar_contacto():
6     """Solicita información del usuario y la guarda en el
    ↪     archivo de texto."""
7     with open(ARCHIVO_CONTACTOS, 'a') as f: # Abrir en modo
    ↪     'append' para agregar datos
8         while True:
9             nombre = input("Ingrese el nombre del contacto (o
    ↪             presione Enter para terminar): ").strip()
10            if not nombre:
11                break
12            telefono = input("Ingrese el teléfono: ").strip()
13            email = input("Ingrese el correo electrónico: ").
    ↪            strip()
14
15            f.write(f"{nombre}, {telefono}, {email}\n")
16            print("Contacto guardado con éxito.")
17
18 def leer_contactos():
19     """Lee y muestra los contactos almacenados en el archivo
    ↪     de texto."""
20     if not os.path.exists(ARCHIVO_CONTACTOS):
21         print("\nNo hay contactos guardados aún.")
22         return
23
24     print("\nLista de contactos guardados:\n")
25     with open(ARCHIVO_CONTACTOS, 'r') as f:
26         for linea in f:
27             nombre, telefono, email = linea.strip().split(',
    ↪             ')
28             print(f"Nombre: {nombre}, Teléfono: {telefono},
    ↪             Correo: {email}")
29
30 def menu():
31     """Muestra el menú de opciones y ejecuta las acciones
    ↪     correspondientes."""
32     while True:
33         print("\nMenú de opciones:")
34         print("1. Guardar un nuevo contacto")
35         print("2. Mostrar contactos guardados")
36         print("3. Salir")
37
```

```

38     opcion = input("\nSeleccione una opción: ")
39     if opcion == "1":
40         guardar_contacto()
41     elif opcion == "2":
42         leer_contactos()
43     elif opcion == "3":
44         print("\nSaliendo del programa.\n")
45         break
46     else:
47         print("Opción no válida, intente nuevamente.")
48
49 if __name__ == "__main__":
50     menu()

```

11.7. Ficheros binarios

Los ficheros binarios almacenan datos en su representación nativa, sin codificar como texto. Se utilizan para imágenes, vídeos, ejecutables, formatos de protocolo, capturas de tráfico, etc. Se abren con los modos `'rb'`, `'wb'`, `'ab'`, `'rb'+` (no se pasa `encoding` porque trabajamos con bytes, no con `str`).

11.7.1. Lectura y escritura básicas

```

with open("datos.bin", "wb") as f:
    f.write(b"Hola, esto es binario")      # Literal bytes
    f.write("Hola de nuevo".encode())      # Cadena codificada
    f.write(bytes([65, 66, 67]))           # Lista de octetos
    -> b'ABC'

with open("datos.bin", "rb") as f:
    primeros_10 = f.read(10)               # Lee 10 bytes
    resto = f.read()                       # Lee hasta EOF

```

Para procesar ficheros grandes **por bloques** (sin cargar todo en memoria), conviene usar el operador morsa `:=` con un tamaño de bloque típico de 4 KiB o 64 KiB:

```

with open("grande.bin", "rb") as f:
    while bloque := f.read(64 * 1024):    # 64 KiB por
        iteracion
        procesar(bloque)

```

11.7.2. Empaquetado con struct

El módulo `struct` traduce entre tipos de Python y representaciones binarias C. Es indispensable para parsear formatos de fichero y protocolos. `struct.pack(fmt, *args)` produce bytes; `struct.unpack(fmt, data)` los decodifica.

La cadena de formato indica los tipos ('i' entero, 'f' flotante, 'I' unsigned int, 'h' short, 's' bytes, etc.) y, opcionalmente, el **orden de bytes**:

Prefijo	Significado
<	Little-endian (<i>x86, x86-64</i>)
>	Big-endian
!	Network byte order (= big-endian)
=	Nativo, alineamiento estándar
@	Nativo, alineamiento nativo (por defecto)

Tabla 2: Prefijos de orden de bytes en `struct`.

```
1 import struct
2
3 # Empaquetar un entero y un flotante (little-endian explícito
↳ )
4 datos = struct.pack("<if", 42, 3.14159)
5 with open("datos.bin", "wb") as f:
6     f.write(datos)
7
8 # Desempaquetar
9 with open("datos.bin", "rb") as f:
10     contenido = f.read()
11     n, x = struct.unpack("<if", contenido)
12     print(n, x)      # 42 3.141590118408203
```

Especifica siempre el orden de bytes en formatos persistentes

Si guardas datos binarios en un fichero o los envías por la red, **usa siempre** un prefijo explícito (<, > o !). El formato por defecto (@) usa el orden de bytes y alineamiento del sistema, lo que rompe la portabilidad: un fichero generado en una máquina ARM podría no leerse correctamente en x86. Para protocolos de red, el estándar es ! (network byte order).

11.7.3. Hashes y resúmenes con hashlib

El módulo `hashlib` proporciona los algoritmos de hash criptográfico estándar: SHA-256, SHA-512, BLAKE2 y otros. Su patrón habitual con ficheros es leer por bloques y

actualizar el digest:

```
import hashlib

def sha256_de(path):
    h = hashlib.sha256()
    with open(path, "rb") as f:
        while bloque := f.read(64 * 1024):
            h.update(bloque)
    return h.hexdigest()

print(sha256_de("paquete.tar.gz"))
```

No uses hashlib.md5 o sha1 para seguridad

md5 y sha1 están **rotos** desde el punto de vista criptográfico (admiten colisiones prácticas). Aún se admiten en `hashlib` porque siguen siendo útiles como simples *checksums* (verificar integridad sin presencia de adversarios). Para cualquier uso de seguridad – comprobar integridad de ficheros descargados, firmar, etc. – usa sha256 o superior.

11.7.4. El módulo pickle

`pickle` serializa cualquier objeto Python (listas, diccionarios, instancias de clase, funciones, etc.) a una representación binaria, y permite reconstruirlo después. Es muy cómodo, pero conlleva un riesgo de seguridad enorme.

```
import pickle

datos = {"clave": "valor", "edad": 30}

with open("estado.pkl", "wb") as f:
    pickle.dump(datos, f)

with open("estado.pkl", "rb") as f:
    datos = pickle.load(f)
```

¡PELIGRO! `pickle.load` permite ejecución de código arbitrario

`pickle.load` **NO** es seguro frente a datos no confiables. Un fichero `.pkl` controlado por un atacante puede ejecutar código arbitrario en el momento de cargarlo, simplemente definiendo el método especial `__reduce__`. Esto es un **vector de ejecución remota de código** clásico.

Como regla absoluta: **nunca cargues con pickle datos cuyo origen no controles 100 %**. No los descargues de Internet, no los aceptes de un usuario, no los uses como formato de intercambio entre sistemas con confianzas distintas.

Alternativas seguras según el caso de uso:

- Para datos estructurados pero *plenamente serializables* (dict, list, str, int, float, bool, None): `json`.
- Para datos numéricos: `numpy.save` / `numpy.load` (con `allow_pickle=False` ↪).
- Para datos tabulares: `pandas` a `parquet` o `csv`.
- Para datos arbitrarios autenticados: serializa a JSON y **firma** el resultado con HMAC.

11.8. Robustez y buenas prácticas al manejar ficheros

Los ficheros son uno de los puntos más delicados de una aplicación: cualquier ruta o contenido cuyo origen no controlamos conviene tratarlo con cautela. Esta sección recoge las trampas más habituales y cómo evitarlas para escribir código robusto.

11.8.1. Path traversal

Un *path traversal* se produce cuando una ruta proporcionada por el usuario incluye `..` para escapar del directorio previsto y acceder a otros ficheros del sistema. El ejemplo clásico es un servidor web que sirve ficheros desde `«/var/www/files/»` y recibe una petición de `«../../etc/passwd»`.

La defensa estándar es **normalizar** la ruta y comprobar que sigue dentro del directorio permitido:

```
from pathlib import Path

def fichero_seguro(base, nombre_pedido):
    base = Path(base).resolve()
    pedido = (base / nombre_pedido).resolve()
    if base not in pedido.parents and base != pedido:
        raise ValueError("Acceso fuera del directorio")
```

```

↪         permitido")
        return pedido

contenido = fichero_seguro("/var/www/files", "../etc/passwd")
↪         # ValueError

```

11.8.2. Escritura atómica

Si un programa se interrumpe a media escritura, el fichero queda corrupto. La técnica habitual para escribir *atómicamente* es escribir a un fichero temporal y, una vez completado, renombrarlo al destino final con `os.replace` (que en POSIX es atómico):

```

import os
from pathlib import Path

def escribir_atómico(path, contenido):
    path = Path(path)
    tmp = path.with_suffix(path.suffix + ".tmp")
    tmp.write_text(contenido, encoding="utf-8")
    os.replace(tmp, path)    # Atómico en POSIX

```

11.8.3. Ficheros temporales seguros

Crear un fichero temporal con un nombre predecible (por ejemplo, `«/tmp/datos.tmp»`) es vulnerable a *race conditions*: otro proceso podría crear el fichero antes que el tuyo con un enlace simbólico a un fichero sensible. El módulo estándar `tempfile` resuelve el problema: genera nombres aleatorios y abre el fichero con permisos restrictivos:

```

import tempfile

↪ with tempfile.NamedTemporaryFile("w", delete=False, encoding=
    "utf-8") as tmp:
    tmp.write("datos sensibles")
    print(tmp.name)    # Nombre seguro y unico

# Para directorios temporales
↪ with tempfile.TemporaryDirectory() as d:
    print(d)    # Directorio que se autodestruye al salir del
    with

```

11.8.4. Permisos al escribir secretos

En sistemas POSIX, los ficheros se crean por defecto con permisos `0644` (lectura mundial). Si estás escribiendo claves, tokens o datos sensibles, restringe los permisos:

```
import os

# Crear con permisos 0600 (solo el propietario lee/escribe)
fd = os.open("clave.key", os.O_WRONLY | os.O_CREAT | os.
↪ O_TRUNC, 0o600)
with os.fdopen(fd, "w") as f:
    f.write(clave_secreta)
```

11.9. Ejemplo: almacenamiento binario de usuarios

Para cerrar el capítulo, un programa interactivo que guarda y recupera registros de usuarios en un fichero binario. Cada usuario tiene un identificador entero, un nombre de longitud fija (20 bytes) y una edad entera. El menú permite añadir, listar y salir; la entrada termina cuando el identificador introducido es 0. Se manejan errores de entrada y la posible inexistencia del fichero.

Código 11.2: Código de ejemplo: almacenamiento y recuperación de datos en un archivo binario.

```
1 import struct
2 import os
3
4 # Definir el formato de los registros binarios: int (4 bytes)
↪   + string de 20 bytes + int (4 bytes)
5 FORMATO = 'i20si'
6 TAMANYO_REGISTRO = struct.calcsize(FORMATO)
7 ARCHIVO_BINARIO = 'usuarios.bin'
8
```



```

9  def guardar_usuarios():
10     usuarios = []
11     while True:
12         try:
13             id_usuario = int(input("Ingrese ID del usuario (0
↳             para terminar): "))
14             if id_usuario == 0:
15                 break
16             nombre = input("Ingrese nombre del usuario (má
↳             ximo 20 caracteres): ").strip()
17             nombre_bytes = nombre.encode('utf-8').ljust(20, b
↳             '\x00') # Convertir a bytes y rellenar
18             edad = int(input("Ingrese edad del usuario: "))
19             usuarios.append((id_usuario, nombre_bytes, edad))
20         except ValueError:
21             print("Error: Entrada inválida. Intente de nuevo.
↳             ")
22
23     with open(ARCHIVO_BINARIO, 'wb') as f:
24         for usuario in usuarios:
25             f.write(struct.pack(FORMATO, *usuario))
26     print(f"\nDatos guardados en {ARCHIVO_BINARIO}.\n")
27
28 def leer_usuarios():
29     if not os.path.exists(ARCHIVO_BINARIO):
30         print(f"\nNo se encontró el archivo {ARCHIVO_BINARIO
↳         }. Primero guarde datos.\n")
31         return
32
33     print("\nLista de usuarios guardados:\n")
34     with open(ARCHIVO_BINARIO, 'rb') as f:
35         while chunk := f.read(TAMANYO_REGISTRO): # Leer en
↳         bloques de tamaño fijo
36             id_usuario, nombre_bytes, edad = struct.unpack(
↳             FORMATO, chunk)
37             nombre = nombre_bytes.decode('utf-8').strip('\x00
↳             ') # Eliminar padding
38             print(f"ID: {id_usuario}, Nombre: {nombre}, Edad:
↳             {edad}")
39

```

```
40 def menu():
41     while True:
42         print("\nMenú de opciones:")
43         print(f"1. Guardar usuarios en archivo binario (\'{
↪         ARCHIVO_BINARIO}\')")
44         print(f"2. Leer y mostrar usuarios desde archivo
↪         binario (\'{ARCHIVO_BINARIO}\')")
45         print("3. Salir")
46
47         opcion = input("\nSeleccione una opción: ")
48         if opcion == "1":
49             guardar_usuarios()
50         elif opcion == "2":
51             leer_usuarios()
52         elif opcion == "3":
53             print("\nSaliendo del programa.\n")
54             break
55         else:
56             print("Opción no válida, intente nuevamente.")
57
58 if __name__ == "__main__":
59     menu()
```

11.10. Ejercicios

EJERCICIO 11.1

Escribe un programa en Python que procese un archivo de logs de acceso web y determine las 10 IPs más frecuentes que han accedido al sistema. El archivo de logs tiene el siguiente formato:

```
192.168.1.1 - - [10/Mar/2025:12:00:01 +0100] "GET /index.html HTTP/1.1" 200 1234
192.168.1.10 - - [10/Mar/2025:18:48:33 +0100] "GET /login HTTP/1.1" 200 2681
192.168.1.10 - - [12/Mar/2025:07:50:51 +0100] "GET /about.html HTTP/1.1" 404 1962
192.168.1.5 - - [05/Mar/2025:23:56:58 +0100] "DELETE /index.html HTTP/1.1" 500 9110
192.168.1.3 - - [09/Mar/2025:23:40:36 +0100] "GET /index.html HTTP/1.1" 301 4393
192.168.1.9 - - [10/Mar/2025:04:28:46 +0100] "GET /contact.html HTTP/1.1" 200 3286
192.168.1.10 - - [08/Mar/2025:11:46:54 +0100] "DELETE /services.html HTTP/1.1" 301 4174
192.168.1.2 - - [10/Mar/2025:12:00:02 +0100] "POST /login HTTP/1.1" 200 3456
...
```

El programa debe leer el archivo de logs, contar la cantidad de accesos por cada IP y generar un archivo de salida con nombre `top_ips.txt` con las 10 IPs con más accesos, junto con su número de accesos.

Se requiere que el programa use `collections.Counter` para contar las ocurrencias de las IPs, sea capaz de manejar archivos grandes de logs mediante la lectura línea por línea y genere los datos en el archivo de salida de forma ordenada en función de la frecuencia.

Solución (girar para leer)

```

1 import re
2 from collections import Counter
3
4 def procesar_logs(archivo_log, archivo_salida):
5     # Expresión regular para extraer las IPs
6     patron_ip = r'(\d+\.\d+\.\d+\.\d+)'
7
8     # Contador para las IPs
9     contador_ips = Counter()
10
11     with open(archivo_log, 'r') as f:
12         for linea in f:
13             # Buscar la IP en cada línea
14             match = re.match(patron_ip, linea)
15             if match:
16                 ip = match.group(1)
17                 contador_ips[ip] += 1
18
19     # Obtener las 10 IPs con más accesos
20     top_ips = contador_ips.most_common(10)
21
22     # Escribir los resultados en el archivo de salida
23     with open(archivo_salida, 'w') as salida:
24         for ip, cantidad in top_ips:
25             salida.write(f"{ip} {cantidad}\n")
26
27     # Llamar a la programación
28     procesar_logs('access.log', 'top-ips.txt')

```

Código 11.3: Solución al Ejercicio 11.1.

EJERCICIO 11.2

↳ Escribe un programa en Python que realice una copia incremental de archivos grandes, donde sólo se copien aquellos archivos que hayan cambiado desde la última copia de seguridad. El programa debe tomar un directorio de origen (`/origen`) y un directorio de destino (`/backup`) y realizar lo siguiente:

1. Comparar los archivos en `/origen` con los de `/backup`.
2. Si un archivo en `/origen` no está en `/backup` o ha cambiado (comparando los hashes SHA256 de los archivos), debe copiarlo al directorio de destino.

El programa debe generar un archivo de registro `«copia_log.txt»` que contenga los nombres de los archivos copiados, el lugar de destino de la copia, y su fecha de copia.

Se requiere que el programa use el módulo `hashlib` para calcular los hashes SHA256 de los archivos, compare los archivos por su hash y tamaño para determinar si han cambiado y utilice `os.walk()` para recorrer los directorios y archivos.

Solución (girar para leer)

```

1 import os
2 import hashlib
3 import shutil
4 from datetime import datetime
5
6 def calcular_hash(archivo):
7     """Calcula el hash SHA256 de un archivo."""
8     sha256 = hashlib.sha256()
9     with open(archivo, 'rb') as f:
10         while chunk := f.read(8192):
11             sha256.update(chunk)
12     return sha256.hexdigest()
13
14 def copiar_incremental(origen, destino, archivo_log):
15     """Realiza una copia incremental de archivos desde
16     un directorio de origen a un destino."""
17     os.makedirs(destino)
18     with open(archivo_log, 'a') as log:
19         for carpeta, _, archivos in os.walk(origen):
20             for archivo in archivos:
21                 archivo_origen = os.path.join(carpeta,
22                                                 archivo)
23                 archivo_destino = os.path.join(destino,
24                                                 os.path.relpath(archivo_origen,
25                                                                 origen))
26                 # Si el archivo no existe en destino o
27                 # ha cambiado
28                 if not os.path.exists(archivo_destino) or \
29                    calcular_hash(archivo_origen) != \
30                    calcular_hash(archivo_destino):
31                     # Crear la estructura de
32                     # directorios si no existe
33                     os.makedirs(os.path.dirname(
34                         archivo_destino), exist_ok=True)
35                     # Copiar el archivo
36                     shutil.copy2(archivo_origen,
37                                   archivo_destino)
38                     # Registrar la copia
39                     fecha_modificacion = datetime.now()
40                     log.write(f"{archivo_destino} copiado a {
41                               fecha_modificacion.strftime('%Y-%m-%d %H:%M:%S')}
42                               \n")
43                     print(f"Archivo {archivo_destino} copiado.")
44
45 # Llamada a la función
46 copiar_incremental('.', './backup', 'copia_log.txt')

```

EJERCICIO 11.3

Escribe un programa en Python que implemente un sistema de indexación de texto para archivos `.txt`. El programa debe crear un índice invertido, que almacene, para cada palabra en el archivo, la lista de líneas en las que aparece. El archivo de texto de entrada tiene el siguiente formato:

Esto es un archivo de ejemplo.
Este archivo tiene varias líneas de texto.
Cada línea contiene varias palabras.

El programa debe almacenar el índice invertido en un archivo denominado «`indice.json`», donde cada palabra clave es una clave del diccionario y el valor asociado es una lista de números de línea en las que aparece esa palabra.

El programa ha de leer el archivo `.txt` de entrada línea por línea, almacenar el índice en formato JSON, implementar una búsqueda rápida de palabras clave que devuelva las líneas donde aparecen y controlar los posibles errores de ejecución que pudieran suceder.

Solución (girar para leer)

```

1 import json
2 import re
3
4 def crear_indice(archivo_entrada, archivo_salida):
5     indice = {}
6
7     try:
8         with open(archivo_entrada, 'r') as f:
9             for num_linea, linea in enumerate(f, start
10                 =1):
11                 palabras = re.findall(r'\b\w+\b', linea
12                     .lower()) # Extrae palabras y las
13                     pasa a minúsculas
14                 for palabra in palabras:
15                     if palabra not in indice:
16                         indice[palabra] = []
17                 indice[palabra].append(num_linea)
18
19         with open(archivo_salida, 'w') as f_out:
20             json.dump(indice, f_out, indent=4)
21             print(f"Índice invertido guardado en {
22                 archivo_salida}")
23         except FileNotFoundError:
24             print(f"Error: El archivo {archivo_entrada} no
25                 se encontró.")
26         except IOError as e:
27             print(f"Error de entrada/salida: {e}")
28         except Exception as e:
29             print(f"Ocurrió un error inesperado: {e}")
30
31     def buscar_palabra(archivo_indice, palabra):
32         try:
33             with open(archivo_indice, 'r') as f:
34                 indice = json.load(f)
35                 palabra = palabra.lower()
36                 if palabra in indice:
37                     return indice[palabra]
38             else:
39                 return []
40         except FileNotFoundError:
41             print(f"Error: El archivo {archivo_indice} no
42                 se encontró.")
43         return []

```


Solución (girar para leer)

```

39 except json.JSONDecodeError:
40     print(f"Error: El archivo {archivo_indice} no
    ← contiene un formato JSON válido.")
41     return []
42 except Exception as e:
43     print(f"Ocurrió un error inesperado: {e}")
44     return []
45
46 # Llamada a las funciones
47 crear_indice('archivo.txt', 'indice.json')
48
49 # Ejemplo de búsqueda
50 resultado = buscar_palabra('indice.json', 'archivo')
51 if resultado:
52     print(f'Las palabras se encuentran en las líneas: {
    ← resultado}')
53 else:
54     print('Palabra no encontrada o error en la búsqueda
    ← .')

```

Código 11.6: Solución al Ejercicio 11.3 – continuación.

EJERCICIO 11.4

Escribe un programa en Python que procese un archivo CSV con registros de usuarios y realice las siguientes validaciones y transformaciones:

1. Si el campo “Nombre” está vacío, eliminar el registro.
2. Si “Edad” es negativa, establecerla en `None`.
3. Si “Email” no tiene un formato válido, reemplazarlo por “email_invalido”.
4. Si “FechaRegistro” no es válida, establecerla en `None`.

El archivo de entrada «`usuarios.csv`» tiene el siguiente formato:

Ejemplo de ejecución esperada

```
ID,Nombre,Edad,Email,FechaRegistro
1,Juan,25,juan@example.com,2025-03-10
2,,30,ana@ejemplo,2025-02-30
3,Carlos,-5,carlos@example.com,2024-12-01
```

El programa debe generar un nuevo archivo «`usuarios_limpios.csv`» con los datos validados y transformados. El programa ha de usar el módulo `csv` para leer y escribir el archivo CSV, validar el correo electrónico utilizando una expresión regular (módulo `re`) y usar `datetime.strptime()` para validar las fechas.

Solución (girar para leer)

```

1 import csv
2 import re
3 from datetime import datetime
4
5 # Función para validar el formato de email
6 def validar_email(email):
7     patron = r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
8     if re.match(patron, email):
9         return email
10    else:
11        return 'email_invalido'
12
13 # Función para validar la fecha
14 def validar_fecha(fecha):
15     try:
16         return datetime.strptime(fecha, '%Y-%m-%d').
17         strftime('%Y-%m-%d')
18    except ValueError:
19        return None
20
21 # Procesar archivo CSV
22 def procesar_csv(archivo_entrada, archivo_salida):
23     with open(archivo_entrada, 'r', newline='') as infle, open(archivo_salida, 'w', newline='') as outfle:
24         reader = csv.DictReader(infle)
25         writer = csv.DictWriter(outfle, fieldnames=
26             reader.fieldnames)
27         writer.writeheader()
28         for row in reader:
29             # Validar y transformar los datos
30             if not row['Nombre']: # Eliminar registros
31                 con Nombre vacio
32             continue
33             if int(row['Edad']) > 0: # Establecer Edad
34                 en None si es negativa
35                 row['Edad'] = None
36             else:
37                 row['Edad'] = int(row['Edad'])
38             # Validar Email
39             row['Email'] = validar_email(row['Email'])
40             # Validar FechaRegistro
41             row['FechaRegistro'] = validar_fecha(row['
42                 FechaRegistro'])
43             # Escribir la fila validada
44             writer.writerow(row)
45
46 # Llamada a la función
47 procesar_csv('usuarios.csv', 'usuarios_limpios.csv')

```

EJERCICIO 11.5

Escribe un programa en Python que implemente el cifrado y descifrado de archivos utilizando el algoritmo AES en modo CBC. El programa debe realizar las siguientes operaciones:

1. Cifrar un archivo de texto dado con una clave proporcionada por el usuario.
2. Guardar el archivo cifrado en formato binario.
3. Permitir el descifrado del archivo cifrado usando la misma clave.

El archivo de entrada debe ser un archivo de texto, y el archivo de salida debe ser un archivo binario. Como requisitos, en este programa se establecen los siguientes: (I) usar el módulo `pycryptodome` para la implementación de AES; (II) solicitar la clave de cifrado al usuario de forma segura utilizando `getpass` (del módulo `getpass`); y (III) gestionar adecuadamente el relleno de bloques para cifrado en modo CBC.

Solución (girar para leer)

```

1 from Crypto.Cipher import AES
2 from Crypto.Util.Padding import pad, unpad
3 from Crypto.Random import get_random_bytes
4 import getpass
5 import os
6
7 # Función para cifrar un archivo
8 def cifrar_archivo(archivo-entrada, archivo-salida,
9 clave):
10     # Asegurar que la clave tenga una longitud adecuada
11     # (16, 24 o 32 bytes para AES)
12     clave = clave.encode('utf-8')
13     clave = clave.ljust(32, b'\0')[:32] # Ajustar a 32
14     # bytes si es necesario
15     # Generar un IV aleatorio
16     iv = get_random_bytes(AES.block-size)
17     # Crear el cifrador AES en modo CBC
18     cipher = AES.new(clave, AES.MODE_CBC, iv)
19     # Leer el archivo de entrada
20     with open(archivo-entrada, 'rb') as f:
21         datos = f.read()
22     # Rellenar los datos para que su tamaño sea un
23     # múltiplo de 16 bytes
24     datos-rellenados = pad(datos, AES.block-size)
25     # Cifrar los datos
26     datos-cifrados = cipher.encrypt(datos-rellenados)
27     # Guardar el IV y los datos cifrados en el archivo
28     # binario de salida
29     with open(archivo-salida, 'wb') as f:
30         f.write(iv + datos-cifrados)

```

Código 11.8: Solución al Ejercicio 11.5.

Solución (girar para leer)

```

32
33 # Función para descifrar un archivo
34 def descifrar_archivo(archivo-entrada, archivo-salida,
35                        clave):
36     # Asegurar que la clave tenga una longitud adecuada
37     # (16, 24 o 32 bytes para AES)
38     clave = clave.encode('utf-8')
39     clave = clave.ljust(32, b'\0')[:32] # Ajustar a 32
40     bytes si es necesario
41     # Leer el archivo cifrado
42     with open(archivo-entrada, 'rb') as f:
43         iv = f.read(AES.block_size) # Leer el IV
44         datos_cifrados = f.read() # Leer los datos
45     # Crear el cifrador AES en modo CBC
46     cipher = AES.new(clave, AES.MODE_CBC, iv)
47     # Descifrar los datos
48     datos_descifrados = unpad(cipher.decrypt(
49         datos_cifrados), AES.block_size)
50     # Guardar los datos descifrados en el archivo de
51     # salida
52     with open(archivo-salida, 'wb') as f:
53         f.write(datos_descifrados)
54     # Solicitar la clave al usuario de forma segura
55     clave = getpass.getpass("Introduce la clave de cifrado:
56     ")
57     # Llamada a la función de cifrado
58     cifrar_archivo(archivo-entrada, 'archivo-cifrado.bin',
59                   clave)
60     # Llamada a la función de descifrado
61     descifrar_archivo('archivo-cifrado.bin', '
62     archivo-descifrado.txt', clave)
63     print("Cifrado y descifrado completados.")

```

Código 11.9: Solución al Ejercicio 11.5 – continuación.

EJERCICIO 11.6

Escribe un programa en Python que permita detectar archivos maliciosos en un sistema utilizando una lista negra de hashes. El programa debe leer un archivo de logs o directorio con archivos a analizar, calcular el hash SHA256 para cada archivo, comparar el hash calculado con una lista negra de hashes almacenada en un archivo «`hashes_maliciosos.txt`», y, si el hash coincide con alguno de la lista negra, alertar al usuario y registrar el hallazgo en un archivo denominado «`registro_alertas.log`».

El programa ha de usar el módulo `hashlib` para calcular los hashes SHA256.

Solución (girar para leer)

```

1 import hashlib
2 import os
3
4 # Función para calcular el hash SHA256 de un archivo
5 def calcular_hash(archivo):
6     try:
7         sha256 = hashlib.sha256()
8         with open(archivo, 'rb') as f:
9             while chunk := f.read(8192):
10                 sha256.update(chunk)
11         return sha256.hexdigest()
12     except FileNotFoundError:
13         print(f"Error: El archivo {archivo} no se
14             encontró.")
15     except IOError as e:
16         print(f"Error de entrada/salida al leer {
17             archivo}: {e}")
18     except Exception as e:
19         print(f"Ocurrió un error inesperado al calcular
20             el hash de {archivo}: {e}")
21         return None
22
23 # Función para cargar la lista negra de hashes
24 # maliciosos
25 def cargar_lista_negra(archivo_lista):
26     try:
27         with open(archivo_lista, 'r') as f:
28             return set(line.strip() for line in f)
29     except FileNotFoundError:
30         print(f"Error: El archivo {archivo_lista} no se
31             encontró.")
32     except IOError as e:
33         print(f"Error de entrada/salida al leer {
34             archivo_lista}: {e}")
35     except Exception as e:
36         print(f"Ocurrió un error inesperado al cargar
37             la lista negra: {e}")
38         return set()

```

Código 11.10: Solución al Ejercicio 11.6.

Solución (girar para leer)

```

32
33 # Función para procesar los archivos y detectar los
34 def detectar_archivos_maliciosos(directorio,
35                                lista_negra, archivo_registro):
36     try:
37         for carpeta, -, archivos in os.walk(
38             directorio):
39             for archivo in archivos:
40                 archivo_completo = os.path.join(
41                     carpeta, archivo)
42                 hash_calculado = calcular_hash(
43                     archivo_completo)
44                 # Si el hash es válido y está en la
45                 # lista negra
46                 if hash_calculado and
47                 hash_calculado in lista_negra:
48                     alerta = f"ALERTA: Archivo
49                     malicioso detectado: {
50                         archivo_completo}, Hash: {
51                         hash_calculado}\n"
52                     print(alerta)
53                     log.write(alerta)
54                     except Exception as e:
55                         print(f"Ocurrió un error inesperado al procesar
56                             el directorio: {e}")
57                         # Cargar lista negra de hashes maliciosos
58                         lista_negra = cargar_lista_negra('hashes-maliciosos.txt')
59                     # Llamada a la función para detectar archivos
60                     malos
61                     if lista_negra:
62                         detectar_archivos_maliciosos('.', lista_negra,
63                                                         registro_registro.log')
64     print("Detección de archivos maliciosos completada.")

```

Código 11.11: Solución al Ejercicio 11.6 – continuación.

EJERCICIO 11.7

Escribe un programa en Python que escanee una lista de URLs en busca de posibles vulnerabilidades de seguridad, como XSS o SQL Injection. El programa debe tomar una lista de URLs de entrada desde un archivo `urls.txt` y realizar las siguientes verificaciones:

1. Verificar si una URL contiene parámetros susceptibles a SQL Injection (por ejemplo, `id=1`).
2. Verificar si una URL puede estar vulnerable a ataques de XSS, al detectar que los parámetros no están adecuadamente escapados.

El programa debe generar un archivo `resultados_vulnerabilidades.txt` que contenga la URL escaneada y el tipo de vulnerabilidad encontrada. En este caso, se requiere que el programa use expresiones regulares para detectar posibles vulnerabilidades y lea las URLs desde un archivo de texto (llamado `urls.txt`), escribiendo los resultados en un archivo de salida (llamado `resultados_vulnerabilidades.txt`).

Ejemplo de ejecución esperada

```
URL: http://example.com?id=1
```

```
Vulnerabilidad detectada: SQL Injection
```

```
URL: http://example.com?name=<script>alert('XSS')</script>
```

```
Vulnerabilidad detectada: XSS
```

Solución (girar para leer)

```

1 import re
2
3 def verificar_sql_injection(url):
4     try:
5         # Patrón para detectar parámetros susceptibles
6         # a SQL Injection
7         patron_sql_injection = r'[?&]id=\d+'
8         if re.search(patron_sql_injection, url):
9             return "SQL Injection"
10        except Exception as e:
11            print(f"Error en la verificación de SQL
12            Injection para la URL {url}: {e}")
13        return None
14        # Función para verificar vulnerabilidades de XSS
15        def verificar_xss(url):
16            try:
17                # Patrón para detectar parámetros que podrían
18                # ser vulnerables a XSS
19                patron_xss = r'[?&]name=.*<script.*>.*</script
20                >.*'
21                if re.search(patron_xss, url, re.IGNORECASE):
22                    return "XSS"
23                except Exception as e:
24                    print(f"Error en la verificación de XSS para la
25                    URL {url}: {e}")
26                return None
27                # Función para escanear URLs desde un archivo de texto
28                y verificar vulnerabilidades
29                def escanear_urls(archivo_urls, archivo_resultados):
30                    try:
31                        with open(archivo_urls, 'r') as f_in, open(
32                            archivo_resultados, 'w') as f_out:
33                            for url in f_in:
34                                url = url.strip()
35                                vulnerabilidad = None
36                                # Verificar SQL Injection
37                                verificar_sql_injection(url)
38                                if not vulnerabilidad:
39                                    # Verificar XSS si no se detectó
40                                    verificar_xss(url)
41                                vulnerabilidad = verificar_xss(url)
42                                print(f"Vulnerabilidad detectada: {vulnerabilidad} en la URL: {url}")
43                                f_out.write(f"{url}\n{vulnerabilidad}\n")
44                                f_out.flush()
45                    except Exception as e:
46                        print(f"Error al escanear URLs: {e}")
47
48 # Ejemplo de uso
49 if __name__ == '__main__':
50     url = input("Introduce una URL: ")
51     vulnerabilidad = verificar_sql_injection(url)
52     if vulnerabilidad:
53         print(f"Se detectó una vulnerabilidad de tipo {vulnerabilidad} en la URL {url}.")
54     else:
55         print("No se detectó vulnerabilidad en la URL proporcionada.")
56
57 # Licencia: CC BY-NC-SA 4.0
58 # Autor: Ricardo Rodríguez (Universidad de Zaragoza, España)
59 # Fecha: 2024-07-24
60 # Versión: 1.0
61 # Contacto: rrodriguez@zaragoza.es
62 # Repositorio: https://github.com/rrodriguez/seguridad-informatica
63 # Código 11.12: Solución al Ejercicio 11.7.

```

Solución (girar para leer)

```

38 # Si se detecta alguna vulnerabilidad,
39     escribir la en el archivo de
40     resultados
41     if vulnerabilidad:
42         f_out.write(f"URL: {url}\nVulnerabilidad detectada: {vulnerabilidad}\n\n")
43     else:
44         f_out.write(f"URL: {url}\nVulnerabilidad detectada: Ninguna\n\n")
45     except FileNotFoundError as e:
46         print(f"Error: El archivo no se encontró: {e}")
47     except IOError as e:
48         print(f"Error de entrada/salida: {e}")
49     except Exception as e:
50         print(f"Ocurrió un error inesperado: {e}")
51 # Llamada a la función para escanear las URLs
52 try:
53     escanear_urls('urls.txt', 'resultados-vulnerabilidades.txt')
54     print("Escaneo completado. Los resultados se guardaron en 'resultados-vulnerabilidades.txt'.")
55 except Exception as e:
56     print(f"Error al ejecutar el programa: {e}")

```

Código 11.13: Solución al Ejercicio 11.7 – continuación.

EJERCICIO 11.8

Escribe un programa en Python que analice archivos binarios en busca de patrones que podrían indicar la presencia de malware. El programa debe realizar lo siguiente:

1. Leer un archivo binario de entrada.
2. Buscar patrones sospechosos, como cadenas que podrían corresponder a URLs maliciosas, direcciones IP o comandos de sistema.
3. Generar un archivo «`informes_malware.txt`» con los hallazgos encontrados, indicando el patrón y su ubicación en el archivo.

El programa ha de usar expresiones regulares para identificar cadenas de texto sospechosas y guardar los resultados en un archivo de texto (llamado «`informes_malware.txt`»). Respecto a las expresiones regulares, considera las siguientes:

```
r'http://[a-zA-Z0-9./?=&_-]+' # URL maliciosas
r'\b(?:\d{1,3}\.){3}\d{1,3}\b' # Direcciones IP
r'\b(?:rm\s+-rf\s+/\s+shutdown\s+-h\s+now)\b' # Comandos
de sistema peligrosos
```

Ejemplo de ejecución esperada

Patrón detectado: `http://malicious-website.com`

Ubicación: 1024 bytes

Patrón detectado: `192.168.1.1`

Ubicación: 2048 bytes

Solución (girar para leer)

```

1 import re
2
3 # Función para buscar patrones sospechosos en un
4 def buscar_patrones(archivo):
5     patrones = [
6         r'http://[a-zA-Z0-9./?=&~+]', # URL
7         r'\b(?:\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})', # Direcciones
8         r'\b(?:\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})', #
9     ]
10    resultados = []
11
12    try:
13        with open(archivo, 'rb') as f:
14            datos = f.read()
15
16            for patron in patrones:
17                for match in re.finditer(patron, str
18                    datos):
19                    resultados.append((match.group(0),
20                        match.start()))
21
22    except FileNotFoundError:
23        print(f"Error: El archivo {archivo} no se
24            encontró.")
25    except IOError as e:
26        print(f"Error de entrada/salida al leer {
27            archivo}: {e}")
28    except Exception as e:
29        print(f"Ocurrió un error inesperado: {e}")
30
31    return resultados
32
33 # Función para guardar los resultados en un archivo de
34 texto
35 def guardar_resultados(resultados, archivo_resultados):
36    try:
37        with open(archivo_resultados, 'w') as f:
38            for patron, ubicacion in resultados:
39                f.write(f"Patrón detectado: {patron}\
40                    ubicación: {ubicacion}\n\n")
41            print(f"Resultados guardados en {
42                archivo_resultados}")
43
44    except Exception as e:
45        print(f"Error al guardar los resultados: {e}")

```

Solución (girar para leer)

```

36 except IOError as e:
37     print(f"Error al guardar el archivo {
38         archivo_resultados}: {e}")
39     except Exception as e:
40         print(f"Ocurrió un error inesperado al guardar
41             los resultados: {e}")
42         # Llamada a las funciones
43         resultados = buscar_patrones('archivo.bin')
44         if resultados:
45             guardar_resultados(resultados, 'informes_malware.
46                 txt')
47         else:
48             print(f"No se encontraron patrones sospechosos.")

```

Código 11.15: Solución al Ejercicio 11.8 – *continuación*.

EJERCICIO 11.9

Escribe un programa en Python que analice los paquetes de red capturados en un archivo de registro de tráfico (.pcap). El programa debe realizar lo siguiente:

1. Cargar el archivo .pcap y extraer los paquetes de red.
2. Analizar los paquetes para detectar posibles ataques, como DDoS (Denegación de Servicio Distribuida) y escaneos de puertos.
3. Generar un archivo «`analisis_trafico.log`» que contenga los paquetes sospechosos detectados, con detalles como la IP origen, IP destino, protocolo y el tipo de ataque detectado.

El programa ha de usar el módulo `scapy` para cargar y analizar los paquetes de red.

Ejemplo de ejecución esperada

```
IP Origen: 192.168.1.10, IP Destino: 192.168.1.20, Protocolo: TCP,  
Ataque: DDoS  
IP Origen: 192.168.1.10, IP Destino: 192.168.1.30, Protocolo: UDP,  
Ataque: Escaneo de puertos
```


Solución (girar para leer)

```

1  from scapy.all import rdpcap, IP, TCP, UDP
2  from collections import defaultdict
3  import logging
4
5  # Configurar el logger para guardar los resultados en
6  logging.basicConfig(filename='análisis-trafico.log',
7                        level=logging.INFO, format='%(message)s')
8  # Función para detectar DDoS
9  def detectar_ddos(packets, umbral=100):
10     origen_ips = defaultdict(int)
11     for packet in packets:
12         if IP in packet:
13             origen_ips[packet[IP].src] += 1
14
15     ataques_ddos = []
16     for ip, count in origen_ips.items():
17         if count > umbral:
18             return ataques_ddos
19
20     # Función para detectar escaneos de puertos
21     def detectar_escaneo_puertos(packets, umbral=20):
22         puerto_origen = defaultdict(int)
23         for packet in packets:
24             if IP in packet and (TCP in packet or UDP in
25                                   packet):
26                 puerto_origen[packet[IP].src, packet[IP].
27                               dst, packet.sport] += 1
28
29     escaneos = [(ip_src, ip_dst, puerto, count) for
30                  (ip_src, ip_dst, puerto, count) in
31                  puerto_origen.items()]
32     if count > umbral:
33         return escaneos
34
35     # Función para analizar el tráfico
36     def analizar_trafico(archivo_pcap):
37         try:
38             packets = rdpcap(archivo_pcap)
39             except Exception as e:
40                 print(f"Error al leer el archivo {archivo_pcap}
41                       {e}")
42         return

```

Código 11.16: Solución al Ejercicio 11.9.

Solución (girar para leer)

```

35
36 # Detectar DDoS
37 ddos_ips = detectar-ddos(packets)
38 for ip in ddos_ips:
39     logging.info(f"IP Origen: {ip}, Ataque: DDoS")
40
41 # Detectar escaneo de puertos
42 escaneos = detectar-escaneo-puertos(packets)
43 for ip-src, ip-dst, puerto, count in escaneos:
44     logging.info(f"IP Origen: {ip-src}, IP Destino:
45         {ip-dst}, Protocolo: {'TCP' if count > 0
46         else 'UDP'}, Ataque: Escaneo de puertos")
47 # Llamada a la función
48 analizar-trafico('trafico.pcap')
49 print("Análisis completado. Los resultados se guardaron
50     en 'analisis-trafico.log'.")

```

Código 11.17: Solución al Ejercicio 11.9 – continuación.

EJERCICIO 11.10

Escribe un programa en Python que implemente un sistema de autenticación en dos pasos utilizando un código temporal generado por la biblioteca `pyotp`. El sistema debe realizar lo siguiente:

1. Solicitar al usuario su nombre de usuario y contraseña.
2. Verificar las credenciales ingresadas con un archivo «`usuarios.csv`» que contiene las credenciales de los usuarios.
3. Si las credenciales son correctas, generar un código temporal (OTP) con `pyotp` y enviarlo al usuario (simulando el envío por correo o mensaje de texto).
4. Solicitar al usuario el código OTP y verificar si es correcto.

El programa ha de usar `pyotp` para generar el código OTP y asumir que las contraseñas se almacenan de forma segura (mediante un hash, usa el módulo `hashlib` en el programa).

Ejemplo de ejecución esperada

```
Introduce tu nombre de usuario: juan
Introduce tu contraseña: password123
Introduce el código OTP enviado: 123456
Autenticación exitosa.
```

Solución (girar para leer)

```

1  import csv
2  import pyotp
3  import hashlib
4  import getpass
5  import time
6
7  # Función para verificar las credenciales en el archivo
8  def verificar_credenciales(usuario, contraseña,
9                               archivo_usuarios):
10     try:
11         reader = csv.DictReader(f)
12         for row in reader:
13             if row['usuario'] == usuario:
14                 # Comparar la contraseña con el
15                 # hash almacenado
16                 if hashlib.sha256(contraseña.encode(
17                     'utf-8')).hexdigest() == row['
18                     contraseña']:
19                     return True
20                 return False
21     except FileNotFoundError:
22         print(f"Error: El archivo {archivo_usuarios} no
23             se encontró.")
24     except IOError e:
25         print(f"Error de entrada/salida al leer {
26             archivo_usuarios}: {e}")
27     except Exception as e:
28         print(f"Ocurrió un error inesperado al
29             verificar las credenciales: {e}")
30     return False
31
32 # Función para generar y enviar el código OTP
33 def generar_otp():
34     # Usar una clave secreta para generar un código OTP
35     secret = pyotp.random_base32() # Genera una clave
36     secreta_alatoria
37     totp = pyotp.TOTP(secret)
38     print(f"Clave secreta (simulada para ejemplo): {
39         secreta}" # Solo para fines de demostración
40     otp = totp.now() # Generar el OTP
41     return otp, totp, secret

```

Código 11.18: Solución al Ejercicio 11.10.

Solución (girar para leer)

```

34 # Función principal para realizar la autenticación en
35     dos pasos
36 def autenticacion-dos-pasos():
37     try:
38         usuario = input("Introduce tu nombre de usuario
39             : ")
40         contrasena = getpass.getpass("Introduce tu
41             contrasena: ")
42         # Verificar las credenciales con el archivo '
43             usuarios.csv'
44         if not verificar-credenciales(usuario,
45             contrasena, 'usuarios.csv'):
46             print("Credenciales incorrectas.")
47             return
48         # Generar el OTP
49         otp, totp, secret = generar-otp()
50         # Simular el envío del código OTP al usuario
51         print(f"Código OTP enviado (simulado): {otp}")
52         tiempo-expiracion = time.time() + 30 # El OTP
53             es válido por 30 segundos
54         # Solicitar el código OTP
55         otp-usuario = input("Introduce el código OTP
56             enviado: ")
57         # Verificar si el código OTP es correcto y no
58             ha expirado
59         if time.time() > tiempo-expiracion:
60             print("El código OTP ha expirado.")
61         elif totp.verify(otp-usuario):
62             print("Autenticación exitosa.")
63         else:
64             print("Código OTP incorrecto.")
65         except Exception as e:
66             print(f"Ocurrió un error inesperado durante el
67                 proceso de autenticación: {e}")
68     # Llamar a la función de autenticación
69     autenticacion-dos-pasos()

```

Código 11.19: Solución al Ejercicio 11.10 – continuación.

12

Introducción a numpy y pandas

En este capítulo exploraremos cómo utilizar **numpy** y **pandas**, dos de las bibliotecas más esenciales en el ecosistema de Python para la ciencia de datos y el análisis numérico. La biblioteca **numpy** permite trabajar con vectores y matrices multidimensionales de forma eficiente, realizando operaciones matemáticas y estadísticas avanzadas sobre grandes volúmenes de datos. **pandas**, construida sobre **numpy**, facilita la manipulación de datos estructurados (especialmente tabulares) y la importación/exportación de los formatos más habituales (CSV, JSON, Excel, Parquet, etc.). Ambas son herramientas fundamentales en inteligencia artificial, aprendizaje máquina y análisis de datos, y aparecen continuamente en los flujos de trabajo habituales de la ciencia de datos (análisis de registros, detección de anomalías, perfilado de series, etc.). La ciberseguridad es uno de los dominios donde estas técnicas resultan útiles, por ejemplo el análisis de *logs*, la detección de anomalías o el perfilado de tráfico.

12.1. Biblioteca numpy

La biblioteca **numpy** (*Numerical Python*) proporciona una estructura de datos eficiente para trabajar con grandes volúmenes de datos numéricos. Su objeto central es el vector multidimensional **ndarray**. A diferencia de las listas de Python, que son heterogéneas y almacenan referencias a objetos Python (con la sobrecarga que ello supone), los **ndarray** son **homogéneos** (todos los elementos son del mismo tipo) y se almacenan en un bloque contiguo de memoria con la representación nativa C del tipo. Esto permite operaciones **vectorizadas** – que actúan sobre todo el array de una vez – implementadas en C optimizado, lo que es órdenes de magnitud más rápido que un bucle Python equivalente.

12.1.1. El objeto ndarray

Para crear un array a partir de una lista usamos `np.array`:

```
import numpy as np

vector = np.array([1, 2, 3, 4])
matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(vector * 2)          # [2 4 6 8]      operacion vectorizada
print(matriz[1, 2])        # 6              acceso fila 1, columna
↪      2
print(np.mean(vector))     # 2.5
```

Cada `ndarray` tiene tres atributos clave:

- `arr.shape`: tupla con las dimensiones (e.g., (3, 4) para una matriz 3×4).
- `arr.dtype`: tipo de los elementos (`int64`, `float64`, etc.).
- `arr.ndim`: número de dimensiones.

12.1.2. Tipos de datos (dtype)

Los tipos de `numpy` extienden los tipos de Python con anchuras fijas, por motivos de rendimiento y memoria. Esta es la conexión natural con lo discutido en el capítulo 8: los modelos de aprendizaje máquina suelen entrenarse en `float32` o `float16` por motivos de velocidad y memoria.

Tipo	Tamaño	Ejemplos de uso
<code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code>	1, 2, 4, 8 bytes	Contadores, índices, identificadores
<code>uint8</code> , <code>uint16</code> , ...	idem	Bytes (<code>uint8</code>), píxeles, paquetes de red
<code>float16</code> , <code>float32</code> , <code>float64</code>	2, 4, 8 bytes	Cálculo numérico (entrenamiento, inferencia)
<code>complex64</code> , <code>complex128</code>	8, 16 bytes	Procesado de señal, FFT
<code>bool_</code>	1 byte	Máscaras booleanas
<code>datetime64</code> , <code>timedelta64</code>	8 bytes	Series temporales

Tabla 1: Tipos de datos más comunes en `numpy`.

El `dtype` se puede especificar al crear el array:

```
import numpy as np

a = np.array([1, 2, 3], dtype=np.int32)
b = np.zeros(1000, dtype=np.float32)      # 4 KiB en lugar
↪      de 8 KiB
print(a.dtype, a.nbytes)                  # int32 12
print(b.dtype, b.nbytes)                  # float32 4000
```


12.1.3. Operaciones y broadcasting

Las operaciones aritméticas se aplican *elemento a elemento* y devuelven un nuevo array:

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
print(a + b)      # [11 22 33]
print(a * b)      # [10 40 90]
print(a ** 2)     # [1 4 9]
```

El *broadcasting* es la regla que permite combinar arrays de formas distintas sin replicar datos en memoria. El caso más simple es operar un array con un escalar: el escalar se “difunde” a todos los elementos:

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(m + 10)
# [[11 12 13]
#  [14 15 16]
#  [17 18 19]]
```

Cuando los arrays tienen formas distintas, *numpy* alinea las dimensiones por la derecha y, en cada eje, exige que coincidan o que una de ellas valga 1. Si se cumple la regla, la dimensión de tamaño 1 se “expande” virtualmente. Por ejemplo:

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) # (3, 3)
v = np.array([1, 0, -1])                       # (3,)

print(m + v)  # v se aplica a cada fila
# [[2 2 2]
#  [5 5 5]
#  [8 8 8]]

c = np.array([[10], [20], [30]]) # (3, 1)
print(m + c)  # c se aplica como columna
# [[11 12 13]
#  [24 25 26]
#  [37 38 39]]
```

El *broadcasting* no copia datos: el array más pequeño se utiliza con un patrón de *strides* adecuado, lo que es muy eficiente.

12.1.4. Indexación, rebanado y máscaras

Los *ndarray* admiten indexación y rebanado igual que las listas, pero generalizado a múltiples dimensiones. Además soportan dos formas avanzadas: **indexación con arrays de índices** y **máscaras booleanas**.

```
a = np.array([10, 20, 30, 40, 50])

# Indexacion clasica
print(a[1])          # 20
print(a[1:4])        # [20 30 40]
print(a[::-1])       # [50 40 30 20 10]

# Indexacion con array de indices
indices = [0, 2, 4]
print(a[indices])    # [10 30 50]

# Mascara booleana
print(a[a > 25])     # [30 40 50]
```

Para arrays bidimensionales, separamos los índices con coma:

```
m = np.arange(12).reshape(3, 4)  # 3 filas x 4 columnas
print(m[1, 2])                  # 6
print(m[:, 0])                  # primera columna
print(m[0, :])                  # primera fila
print(m[m > 5])                 # [6 7 8 9 10 11]
```

12.1.5. Reorganización: reshape, transpose, flatten

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(m.reshape(9))            # [1 2 3 4 5 6 7 8 9]
print(m.flatten())              # idem (siempre devuelve copia)
print(m.T)                      # transpuesta
print(m.reshape(-1, 3))        # -1 = "deduce esta dimension" ->
↪ (3, 3)
```

Para combinar arrays existen `np.concatenate`, `np.hstack` (horizontal), `np.vstack` (vertical) y `np.stack` (apila por una nueva dimensión):

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(np.concatenate((a, b)))  # [1 2 3 4 5 6]
print(np.vstack((a, b)))       # [[1 2 3]
                                #  [4 5 6]]
```

12.1.6. Funciones vectorizadas útiles

Más allá de las operaciones aritméticas, **numpy** ofrece muchas funciones que evitan bucles explícitos. Las más usadas:

- ↪ ■ Estadísticas: `np.mean`, `np.median`, `np.std`, `np.var`, `np.sum`, `np.prod`, `np.min`, `np.max`, todas con parámetro `axis=` para operar a lo largo de una dimensión.
- Matemáticas: `np.log`, `np.exp`, `np.sqrt`, `np.sin`, `np.cos`, etc.
- Búsqueda: `np.argmax`, `np.argmin` (índices del extremo), `np.argsort` (índices que ordenan), `np.where` (operador condicional vectorizado), `np.unique` (valores únicos, opcionalmente con frecuencias).
- Álgebra lineal en el submódulo `np.linalg`: `inv` (inversa), `det` (determinante), `eig` (autovalores), `solve` (sistemas), `norm` (normas).

```

a = np.array([3, 1, 4, 1, 5, 9, 2, 6])

print(np.argmax(a))                # 5 (índice del 9)
print(np.unique(a, return_counts=True))
# (array([1, 2, 3, 4, 5, 6, 9]), array([2, 1, 1, 1, 1, 1, 1]))
↪

print(np.where(a > 3, a, 0))        # ceros donde no, valor
↪   original donde si
# [0 0 4 0 5 9 0 6]

```

Vectoriza siempre que puedas

`np.apply_along_axis(funcion_python, eje, arr)` existe pero ejecuta la función en Python una vez por fila/columna, lo que pierde casi toda la ventaja de *numpy*. Antes de recurrir a él, comprueba si tu cálculo se puede expresar con operaciones vectorizadas, `np.where`, máscaras o álgebra lineal: la diferencia de rendimiento puede ser de uno o dos órdenes de magnitud.

12.1.7. Eficiencia: vistas vs copias

Algunas operaciones devuelven una **vista** (los datos no se duplican; modificar la vista modifica el original); otras devuelven una **copia**. Es importante saber distinguirlas:

```

a = np.array([1, 2, 3, 4, 5])

vista = a[1:4]          # Slicing basico -> VISTA
vista[0] = 99
print(a)                # [ 1 99  3  4  5]    <- afectado!

copia = a[[0, 2, 4]]    # Indexacion con array -> COPIA
copia[0] = -1
print(a)                # [ 1 99  3  4  5]    <- no afectado

```

Si necesitas una copia explícita: `a.copy()`. Si no estás seguro, comprueba con `arr.base is None` (es `None` si `arr` es propietario de los datos).

12.1.8. Generación de números pseudoaleatorios

numpy ofrece la API moderna basada en `np.random.default_rng`, que es la recomendada desde numpy 1.17. Crea un *generador* con una semilla y se usan sus métodos para producir números:

```
import numpy as np

rng = np.random.default_rng(seed=42)    # Generador
reproducible

print(rng.random(5))                    # 5 floats en [0, 1)
print(rng.integers(0, 100, size=5))      # 5 enteros en [0,
100)
print(rng.normal(loc=5, scale=2, size=5)) # normal mu=5
sigma=2
print(rng.choice([10, 20, 30], size=4)) # muestreo con
reemplazo
```

La API antigua (`np.random.seed`, `np.random.rand`, `np.random.randint`, `np.random.normal`) sigue funcionando pero no se recomienda en código nuevo: la nueva API tiene mejor calidad estadística (PCG64 vs Mersenne Twister) y aísla el estado en el objeto `rng` (importante en programas concurrentes).

numpy.random no es criptográficamente seguro

Igual que el módulo `random` del cap. 7, `numpy.random` usa generadores deterministas pensados para simulación, no para criptografía. Para claves, tokens o cualquier valor que deba permanecer impredecible, usa `secrets`.

12.1.9. Interoperabilidad con bytes y ficheros binarios

Dado que a menudo trabajamos con datos textuales y registros, conviene mencionar tres funciones de `numpy` que conectan con el cap. 11:

- `np.frombuffer(data, dtype=...)`: interpreta un objeto `bytes` como un `ndarray` del tipo dado, **sin copiar**. Ideal para parsear paquetes de red u otras estructuras binarias.
- `np.fromfile(path, dtype=..., count=...)`: lee directamente un fichero binario.
- `arr.tobytes()`: serializa un array a una cadena de bytes.

```
# Interpretar 4 bytes como 4 muestras uint8
datos = b"\x01\x02\x03\x04"
muestras = np.frombuffer(datos, dtype=np.uint8)
print(muestras)           # [1 2 3 4]

# Volver a bytes
print(muestras.tobytes()) # b'\x01\x02\x03\x04'
```

Para guardar y cargar arrays preservando el `dtype` y la forma, `np.save` y `np.load` usan un formato propio (`«.npy»`) más seguro y eficiente que `pickle` (recuerda la advertencia del cap. 11: pasa siempre `allow_pickle=False` a `np.load` si los datos no son tuyos).

12.1.10. Otros tipos: complejos, fechas, estructurados

Números complejos. Soportados nativamente en `numpy` con sintaxis a `bj+`:

```
z = np.array([1 + 2j, 3 + 4j, 5 + 6j])
print(np.real(z))    # [1. 3. 5.]
print(np.imag(z))    # [2. 4. 6.]
print(np.abs(z))     # [2.236..., 5., 7.810...]    modulo
print(np.angle(z))   # argumento (radianes)
```

```
fechas = np.array(['2025-03-01', '2025-03-02', '2025-03-03'],
                  dtype='datetime64[D]')
print(fechas + np.timedelta64(10, 'D'))
# ['2025-03-11' '2025-03-12' '2025-03-13']
```

Arrays estructurados. Cada elemento es un registro con varios campos nombrados, cada uno con su propio `dtype`:

```
↪ tipo = np.dtype([('nombre', 'U10'), ('edad', 'i4'), ('altura',
↪      , 'f4')])
personas = np.array([('Alice', 25, 1.70), ('Bob', 30, 1.80)],
                    dtype=tipo)

print(personas['nombre']) # ['Alice' 'Bob']
print(personas['edad'])  # [25 30]
```

Para datos verdaderamente tabulares y heterogéneos, `pandas` es más cómodo y potente, como vemos a continuación.

12.2. Biblioteca pandas

La biblioteca `pandas` está construida sobre `numpy` y proporciona dos estructuras principales:

- **Series**: “lista etiquetada” unidimensional. Como un `ndarray` 1D, pero con un *índice* explícito.
- **DataFrame**: tabla bidimensional con filas (índice) y columnas (cada una de un `dtype` propio). Cada columna es una **Series**.

Vamos a verlas en orden.

12.2.1. El tipo Series

Creación

```
import pandas as pd

s = pd.Series([10, 20, 30, 40])           # Índice
    automatico 0..3
s = pd.Series([10, 20, 30, 40],
              index=['a', 'b', 'c', 'd']) # Índice
    personalizado
s = pd.Series([1.5, 'Python', True])     # Heterogenea ->
    dtype=object
```

Acceso, operaciones y funciones

```
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])

print(s['b'])           # 20 por etiqueta
print(s.iloc[1])        # 20 por posición
print(s + 5)            # vectorizado
print(s.mean(), s.sum(), s.max())
print(s[s > 20])        # Filtrado por máscara

# Aplicar una función a cada elemento
print(s.apply(lambda x: x ** 2))
```

Frecuencias: `value_counts`

Para contar ocurrencias de los valores de una **Series**, `value_counts()` es más natural que `collections.Counter`:

```

↪ s = pd.Series(['admin', 'user', 'admin', 'guest', 'user', 'admin'])
print(s.value_counts())
# admin      3
# user       2
# guest      1
# Name: count, dtype: int64

```

Fechas y cadenas

pandas ofrece accesores `.dt` y `.str` para operar vectorizadamente sobre series de fechas y de cadenas:

```

↪ fechas = pd.to_datetime(pd.Series(['2025-01-01', '2025-02-15', '2025-03-31']))
print(fechas.dt.month)           # 1, 2, 3
print(fechas.dt.day_name())      # Wednesday, Saturday, Monday

texto = pd.Series(['apple', 'BANANA', 'Cherry'])
print(texto.str.upper())         # APPLE, BANANA, CHERRY
print(texto.str.contains('an', case=False)) # True, True, False
↪

```

Alineación de índices

Cuando se operan dos *Series*, *pandas* alinea por índice antes de operar. Los índices que no coincidan dan *NaN*:

```

s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
s2 = pd.Series([4, 5, 6], index=['b', 'c', 'd'])
print(s1 + s2)
# a      NaN
# b      6.0
# c      8.0
# d      NaN

```

Valores nulos: *NaN*

Las series suelen contener valores ausentes (*NaN*, *Not a Number*). Las operaciones más útiles:

- `s.isna()`: máscara booleana de nulos.
- `s.dropna()`: descarta los nulos.

- `s.fillna(valor)`: rellena con un valor fijo.
- `s.ffill()` / `s.bfill()`: rellenan con el último valor anterior/siguiente válido (útil en series temporales con huecos).

```
s = pd.Series([1, 2, None, 4, 5])
print(s.isna())           # mascara
print(s.fillna(0))        # rellena con 0
print(s.fillna(s.mean())) # rellena con la media
```

12.2.2. El tipo DataFrame

Un `DataFrame` es una tabla con filas (índice) y columnas (cada una con su `dtype`).

Creación

```
1 import pandas as pd
2
3 df = pd.DataFrame({
4     'Nombre': ['Alice', 'Bob', 'Charlie'],
5     'Edad': [25, 30, 35],
6     'Ciudad': ['New York', 'Los Angeles', 'Chicago']
7 })
8 print(df)
9
10 #      Nombre  Edad      Ciudad
11 # 0    Alice    25    New York
12 # 1     Bob    30  Los Angeles
13 # 2  Charlie    35    Chicago
```

Lectura y escritura

`pandas` ofrece funciones `pd.read_*` y métodos `df.to_*` para los formatos más habituales: CSV, JSON, Excel, Parquet, HDF5, SQL... Las opciones más útiles de `read_csv` son:

- `dtype={...}`: forzar el tipo de cada columna (evita inferencias incorrectas).
- `parse_dates=[...]`: convierte directamente columnas de fechas.
- `chunksize=N`: itera sobre el fichero en bloques de N filas (para ficheros que no caben en memoria).


```
import pandas as pd

# Lectura
df = pd.read_csv('usuarios.csv', dtype={'ID': 'int32'},
                 parse_dates=['FechaRegistro'])

print(df.head())    # primeras 5 filas

# Escritura
df.to_csv('salida.csv', index=False)
df.to_parquet('salida.parquet')    # mas eficiente que CSV
df.to_json('salida.json', orient='records', force_ascii=False)
→ )
```

Otros formatos. `read_excel`, `read_json`, `read_sql`, `read_parquet` leen formatos correspondientes. *Parquet* es un formato columnar binario muy eficiente para datasets grandes – es lo que se usa habitualmente en el ecosistema de datos hoy.

Selección: `loc`, `iloc`, máscaras

- `df['col']` o `df.col`: selecciona una columna como `Series`.
- `df[['col1', 'col2']]`: selecciona varias columnas como `DataFrame`.
- `df.iloc[i]`, `df.iloc[i, j]`: por posición entera.
- `df.loc[etiqueta]`, `df.loc[etiqueta, 'col']`: por etiquetas del índice.
- `df[mascara_booleana]`: filtra filas.

```
df = pd.DataFrame({'Nombre': ['Alice', 'Bob', 'Charlie'],
                  'Edad': [25, 30, 35],
                  'Ciudad': ['NY', 'LA', 'Chicago']})

print(df['Edad'])    # Serie
print(df.iloc[0])    # Primera fila
print(df.loc[df['Edad'] > 28, 'Nombre']) # Nombres con edad >
→ 28
```

SettingWithCopyWarning: usa loc para asignar

Asignar a una selección encadenada **no funciona de forma fiable**:

```
df[df['Edad'] > 28]['Ciudad'] = 'desconocida' # MAL:  
puede no escribir
```

Esto suele lanzar `SettingWithCopyWarning` porque `df[df['Edad'] > 28]` devuelve, según el caso, una vista o una copia, y la siguiente asignación no garantiza modificar el original. La forma correcta es asignar a `.loc` con la condición y la columna en una sola operación:

```
df.loc[df['Edad'] > 28, 'Ciudad'] = 'desconocida' #  
BIEN
```

Modificación y eliminación

```
df['Edad'] = df['Edad'] + 1 # Recalcular  
columna  
df['Categoria'] = df['Edad'].apply( # Crear columna  
nueva  
lambda x: 'Joven' if x < 30 else 'Mayor')  
  
df.drop('Categoria', axis=1, inplace=True) # Eliminar columna  
df.drop(0, axis=0, inplace=True) # Eliminar fila  
por etiqueta
```

Resumen estadístico: describe

`df.describe()` devuelve un resumen estadístico (count, mean, std, min, max, percentiles 25/50/75) de las columnas numéricas. Con `include='all'` extiende a todas las columnas.

```
df = pd.DataFrame({'Edad': [25, 30, 35, 40, 45],  
                  'Sesiones': [10, 15, 12, 18, 20]})  
print(df.describe())
```

Valores nulos

Igual que en Series: `isna`, `dropna`, `fillna`, `ffill`, `bfill`.

```

df = pd.DataFrame({
    'Nombre': ['Alice', 'Bob', None, 'Charlie'],
    'Edad': [25, None, 35, 40],
    'Ciudad': ['NY', 'LA', None, 'Chicago']
})

print(df.isna().sum())      # Conteo de nulos por columna
df.dropna(axis=0)           # Filas sin nulos
df.fillna({'Nombre': 'Desconocido', 'Edad': df['Edad'].mean()
})

# Forward / backward fill (modernos: metodo, no parametro
# method=)
df.ffill()
df.bfill()

```

Agrupación: groupby

`groupby` divide el `DataFrame` en grupos y aplica una función de agregación. Es una de las herramientas más útiles de *pandas*:

```

df = pd.DataFrame({
    'Nombre': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Edad': [25, 30, 35, 40, 28],
    'Ciudad': ['NY', 'LA', 'NY', 'Chicago', 'Chicago'],
    'Departamento': ['Ventas', 'TI', 'Ventas', 'TI', 'Ventas']
})

# Edad media por ciudad
print(df.groupby('Ciudad')['Edad'].mean())

# Múltiples agregaciones a la vez
print(df.groupby('Ciudad')['Edad'].agg(['mean', 'std', 'count'])

# Agrupación jerárquica
print(df.groupby(['Ciudad', 'Departamento'])['Edad'].mean())

```

Combinación de tablas: merge

`pd.merge` realiza un *join* estilo SQL entre dos `DataFrame`. El parámetro `on=` indica la columna(s) común(es); `how=` controla el tipo (`'inner'`, `'left'`, `'right'`, `'outer'`).

```
usuarios = pd.DataFrame({
    'id': [1, 2, 3],
    'nombre': ['Alice', 'Bob', 'Charlie']
})

eventos = pd.DataFrame({
    'usuario_id': [1, 1, 2, 4],
    'evento': ['login', 'logout', 'login', 'login']
})

# Join: une eventos con datos del usuario
combinado = pd.merge(eventos, usuarios,
                     left_on='usuario_id', right_on='id', how=
                     'left')
↪ print(combinado)
```

	usuario_id	evento	id	nombre
# 0	1	login	1.0	Alice
# 1	1	logout	1.0	Alice
# 2	2	login	2.0	Bob
# 3	4	login	NaN	NaN

Reorganización: pivot y melt

`pivot` pasa de “formato largo” a “formato ancho”; `melt` es la operación inversa. Es habitual en análisis de datos cuando los datos vienen como series largas (una fila por observación) y se quieren mostrar como tabla.

```
ventas = pd.DataFrame({
    'fecha': ['2025-01', '2025-01', '2025-02', '2025-02'],
    'producto': ['A', 'B', 'A', 'B'],
    'unidades': [10, 5, 12, 7]
})

# Largo -> ancho: filas=fechas, columnas=productos, valores=
↪ unidades
↪ print(ventas.pivot(index='fecha', columns='producto', values=
    'unidades'))
```

	A	B
# fecha		
# 2025-01	10	5
# 2025-02	12	7

```

# Ancho -> largo (operacion inversa)
ancho = ventas.pivot(index='fecha', columns='producto',
    ↪     values='unidades')
largo = ancho.reset_index().melt(id_vars='fecha',
    ↪                               var_name='producto',
                               value_name='unidades')

```

Series temporales

pandas destaca en series temporales. Si el índice es de tipo `datetime64`, se desbloquean operaciones como `resample` (cambio de frecuencia), `shift` (desplazamiento) y `rolling` (ventana móvil).

```

data = pd.DataFrame({
    ↪     'fecha': pd.date_range('2025-01-01', periods=6, freq='D')
    ,
    'valor': [10, 20, 30, 40, 50, 60]
})
data = data.set_index('fecha')

# Remuestreo diario -> mensual (sumar)
print(data.resample('ME').sum())    # 'ME' = Month End (pandas
    ↪    2.x)

# Diferencia con el día anterior
data['delta'] = data['valor'] - data['valor'].shift(1)

# Media movil de 3 días
data['mm3'] = data['valor'].rolling(3).mean()
print(data)

```

Componentes de la fecha mediante el accesor `.dt`:

```

df = pd.DataFrame({'fecha': pd.date_range('2025-01-01',
    ↪     periods=3, freq='D')})
df['anyo'] = df['fecha'].dt.year
df['mes'] = df['fecha'].dt.month
df['dia'] = df['fecha'].dt.day
df['dia_semana'] = df['fecha'].dt.day_name()

```

12.3. Aplicación: análisis de registros de actividad

Cerramos el capítulo ilustrando cómo se combinan `numpy` y `pandas` en escenarios típicos de seguridad y análisis de *logs*. Los ejemplos son intencionadamente compactos;

en producción, querríamos añadir robustez (manejo de errores, validación de entradas, etc.) pero las ideas son las mismas.

12.3.1. Detección de anomalías por z-score

Un patrón muy común para detectar valores atípicos es el *z-score*: para cada observación, calcular cuántas desviaciones estándar la separan de la media. Valores con $|z| > 3$ se consideran sospechosos:

```
import numpy as np
import pandas as pd

# Tiempos de respuesta de un servicio (ms)
tiempos = pd.Series(np.random.default_rng(0).normal(50, 10,
    1000))
tiempos.iloc[42] = 500    # Inyectamos un outlier

z = (tiempos - tiempos.mean()) / tiempos.std()
anomalias = tiempos[z.abs() > 3]
print(f"Detectadas {len(anomalias)} anomalias")
```

12.3.2. Top de IPs en un fichero de logs

Combinando pandas y `value_counts`, sustituimos el código manual con `collections.Counter` del cap. 10 por una sola línea:

```
import pandas as pd
import re

ip_re = re.compile(r"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$")

with open("access.log", encoding="utf-8") as f:
    ips = pd.Series(m.group(1) for line in f if (m := ip_re.
        match(line)))

print(ips.value_counts().head(10))    # Top 10 de IPs mas
    activas
```

12.3.3. Eventos por hora con resample

Para detectar picos de actividad sospechosa, agrupamos eventos por hora y los contamos. La media móvil ayuda a localizar desviaciones:

```

eventos = pd.read_csv("eventos.csv", parse_dates=['timestamp'])
eventos = eventos.set_index('timestamp')

por_hora = eventos.resample('h').size()           # eventos por
hora                                           hora
mm = por_hora.rolling(24).mean()               # media movil
de 24h
picos = por_hora[por_hora > mm * 3]             # horas con 3
x la media
print(picos)

```

12.3.4. Cruzar IPs con una lista negra

`merge` permite cruzar dos tablas por una clave común; con `how='inner'` obtenemos solo las coincidencias. Útil para enriquecer eventos con información de *threat intelligence*:

```

import pandas as pd

eventos = pd.read_csv("eventos.csv")           # con
columna 'ip'
malas = pd.read_csv("ioc_malas.csv")           # con
columnas 'ip', 'campaign'

sospechosas = pd.merge(eventos, malas, on='ip', how='inner')
print(f"Eventos coincidentes con IOCs: {len(sospechosas)}")
print(sospechosas.groupby('campaign').size().sort_values(
ascending=False))

```

12.3.5. Análisis de frecuencias para criptoanálisis clásico

Combinando `numpy` y `pandas`, contar frecuencias de letras en un texto cifrado es trivial:

```

texto = "WKLV LV D VHFUHW PHVVDJH HQFRGHG ZLWK FDHVDU"
letras = pd.Series([c for c in texto if c.isalpha()])
print(letras.value_counts(normalize=True).head(5))

```

Con esto cerramos los fundamentos de manipulación de datos. En el capítulo siguiente (13) veremos cómo presentar visualmente lo que hemos analizado, con `matplotlib` y `seaborn`, y en el capítulo 14 daremos el salto al aprendizaje automático con `scikit-learn`.

12.4. Ejercicios

EJERCICIO 12.1

Escribe un programa en Python que, usando `numpy`, simule 1000 registros de tráfico de red, cada uno con tres características: (1) duración del paquete (entre 1 y 5000 ms), (2) tamaño del paquete (entre 64 y 1500 bytes) e (3) intervalo entre paquetes (entre 1 y 3000 ms). El programa debe identificar los paquetes anómalos, definidos como aquellos cuyos valores en alguna de las tres características estén a más de una vez y media desviaciones estándar por encima o por debajo de la media.

Ejemplo de salida esperada

```
Total de paquetes anómalos detectados: 339
```

```
Primeros 5 paquetes anómalos:
```

```
[[3093  138 1693]
```

```
 [2920  410  106]
```

```
 [ 131  703 1444]
```

```
 [2392  939    2]
```

```
 [4118   88 1287]]
```


Solución (girar para leer)

```

1 import numpy as np
2
3 # Fijar semilla para reproducibilidad
4 np.random.seed(42)
5
6 # Generar 1000 paquetes de red con 3 características
7 # duración (ms), tamaño (bytes), intervalo (ms)
8 trafico = np.column_stack([
9     np.random.randint(1, 5000, 1000),
10    np.random.randint(64, 1500, 1000),
11    np.random.randint(1, 3000, 1000)
12 ])
13
14 # Calcular media y desviación estándar por columna
15 media = np.mean(trafico, axis=0)
16 desviacion = np.std(trafico, axis=0)
17
18 # Definir umbrales superior e inferior (1.5
19 # desviaciones)
20 umbral_superior = media + 1.5 * desviacion
21 umbral_inferior = media - 1.5 * desviacion
22 # Detectar anomalías
23 anomalos = (trafico > umbral_inferior) | (trafico >
24    umbral_superior)
25 paquetes_anomalos = np.any(anomalos, axis=1)
26 # Mostrar resultados
27 print(f"Total de paquetes anomalos detectados: {np.sum(
28    paquetes_anomalos)}")
29 print("Primeros 5 paquetes anomalos:\n", trafico[
30    paquetes_anomalos[:5]])

```

Código 12.1: Solución al Ejercicio 12.1.

13

Visualización de datos con matplotlib y seaborn

En este capítulo veremos cómo representar gráficamente los datos que hemos aprendido a manipular en el capítulo 12. La visualización es un pilar del análisis exploratorio: una buena gráfica permite detectar patrones, valores atípicos, distribuciones inesperadas y errores en los datos que pasarían inadvertidos en una tabla. En cualquier proyecto de análisis de datos, las representaciones visuales (mapas de calor de actividad, distribuciones de valores, series temporales, curvas de evaluación de modelos) son a menudo la forma más eficaz de comunicar resultados a un equipo o de descubrir comportamientos inesperados. La ciberseguridad es uno de los dominios donde esto resulta útil, por ejemplo al representar tamaños de paquete, series temporales de eventos o curvas ROC de detectores. Trabajaremos con dos bibliotecas complementarias: `matplotlib`, la *lingua franca* de la visualización en Python, y `seaborn`, una capa de alto nivel construida sobre `matplotlib` que produce gráficos estadísticos elegantes con muy poco código.

13.1. Biblioteca matplotlib

`matplotlib` es la biblioteca de visualización más usada en Python. Su API es muy flexible – permite controlar cualquier detalle del gráfico – pero también puede resultar verbosa. La forma habitual de utilizarla es importando el submódulo `pyplot` con el alias estándar `plt`:

```
import matplotlib.pyplot as plt
```

13.1.1. Anatomía de una figura

Una figura de `matplotlib` tiene dos elementos principales:

- La **Figure** (figura): el lienzo completo donde se dibuja.
- Los **Axes** (ejes): cada subgráfico dentro de la figura. Una figura puede tener uno o varios **Axes** organizados en una rejilla.

Hay dos estilos de uso habituales:

- **Estilo *state-machine*** (heredado de Matlab): se llama directamente a `plt.plot(...)`, `plt.title(...)`, etc. Es cómodo para gráficos rápidos.
↪
- **Estilo orientado a objetos**: se obtiene la figura y los ejes con `fig, ax = plt.subplots()` y se invocan los métodos sobre `ax`. Es más verboso, pero más flexible y recomendable para gráficos complejos. Es el estilo que usaremos en este capítulo.
↪

13.1.2. Primer gráfico: una línea

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 2 * np.pi, 100)
5 y = np.sin(x)
6
7 fig, ax = plt.subplots(figsize=(7, 4))
8 ax.plot(x, y, label="sin(x)")
9 ax.set_xlabel("x")
10 ax.set_ylabel("sin(x)")
11 ax.set_title("Función seno")
12 ax.legend()
13 ax.grid(True)
14 plt.show()
```

Algunos comentarios:

- `plt.subplots` devuelve una tupla (`figura`, `ejes`) que desempaquetamos.
- `figsize=(ancho, alto)` se mide en pulgadas. Es un buen parámetro al que prestar atención: las figuras demasiado pequeñas son ilegibles.
- `ax.plot` acepta opciones de estilo: color (`color="red"`), grosor (`linewidth=2`), tipo de línea (`linestyle="--"`), marcador (`marker="o"`).
- `plt.show()` abre una ventana interactiva. En Jupyter no es necesario; en scripts sí.

13.1.3. Tipos de gráfico habituales

```
rng = np.random.default_rng(0)
x = rng.normal(size=200)
y = 0.5 * x + rng.normal(scale=0.3, size=200)

fig, ax = plt.subplots()
ax.scatter(x, y, s=20, alpha=0.6)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Dispersión x vs y")
```

El parámetro `alpha` da transparencia, útil cuando los puntos se solapan.

Histograma. Para visualizar la distribución de una variable:

```
muestras = rng.normal(loc=50, scale=10, size=10000)

fig, ax = plt.subplots()
ax.hist(muestras, bins=40, edgecolor="black")
ax.set_xlabel("Valor")
ax.set_ylabel("Frecuencia")
ax.set_title("Histograma de muestras normales")
```

El parámetro `bins` controla el número de barras: pocos *bins* ocultan estructura, demasiados producen ruido. Un buen punto de partida es la regla de Sturges (`int(np.log2(n)) + 1`).

Gráfico de barras. Útil para variables categóricas:

```
categorias = ["GET", "POST", "PUT", "DELETE"]
conteos = [1245, 320, 80, 25]

fig, ax = plt.subplots()
ax.bar(categorias, conteos, color="steelblue")
ax.set_ylabel("Peticiones")
ax.set_title("Distribución de métodos HTTP")
```

Boxplot. Resume cinco estadísticos (mínimo, primer cuartil, mediana, tercer cuartil, máximo) y resalta valores atípicos. Muy útil para comparar distribuciones:

```
g1 = rng.normal(50, 5, 200)
g2 = rng.normal(55, 8, 200)
g3 = rng.normal(48, 12, 200)
```

```
fig, ax = plt.subplots()
ax.boxplot([g1, g2, g3], tick_labels=["Servicio A", "Servicio
↪      B", "Servicio C"])
ax.set_ylabel("Latencia (ms)")
ax.set_title("Distribución de latencias por servicio")
```

Mapa de calor (heatmap). Para matrices 2D, donde el color codifica la magnitud:

```
matriz = rng.random((10, 12))

fig, ax = plt.subplots()
im = ax.imshow(matriz, cmap="viridis", aspect="auto")
fig.colorbar(im, ax=ax, label="Intensidad")
ax.set_title("Matriz de actividad")
```

13.1.4. Múltiples subgráficos

`plt.subplots(filas, cols)` devuelve una matriz de `Axes` con la que componer varios gráficos en una misma figura:

```
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

axes[0].plot(x, np.sin(x))
axes[0].set_title("sin(x)")

axes[1].plot(x, np.cos(x), color="red")
axes[1].set_title("cos(x)")

fig.tight_layout()    # Ajusta espacios automáticamente
```

13.1.5. Personalización: ejes, leyenda, anotaciones

- `ax.set_xlim(a, b)` / `ax.set_ylim(a, b)`: rango de los ejes.
- `ax.set_xticks(...)` / `ax.set_yticks(...)`: marcas concretas.
- `ax.set_xscale("log")`: escala logarítmica (muy útil cuando los datos abarcan varios órdenes de magnitud, como tamaños de fichero o tiempos de respuesta).
- `ax.legend(loc="best")`: leyenda automática con las etiquetas `label=` pasadas a `plot/scatter`.
- `ax.annotate("texto", xy=(x, y), ...)`: anotación con flecha en un punto concreto.
- `ax.axvline(x, color="red", linestyle="--")`: línea vertical (útil para marcar umbrales).

13.1.6. Guardar la figura

`fig.savefig("salida.png", dpi=150, bbox_inches="tight")` guarda la figura en disco. Formatos típicos: `.png` (mapa de bits, web), `.pdf` y `.svg` (vectoriales, ideales para publicación). El parámetro `dpi` controla la resolución; `bbox_inches="tight"` recorta los márgenes en blanco.

Reproducibilidad: cierra siempre la figura

En scripts que generan muchas figuras (en bucle, por ejemplo), `matplotlib` acumula todas en memoria. Tras `savefig`, llama a `plt.close(fig)` para liberar la memoria de esa figura.

13.2. Biblioteca *seaborn*

seaborn se construye sobre `matplotlib` y aporta dos cosas: estilos visuales más cuidados por defecto, y una API de alto nivel pensada para datos tabulares (`DataFrame` de *pandas*). Se importa convencionalmente como `sns`:

```
import seaborn as sns
```

13.2.1. Filosofía y datos en formato largo

Casi todas las funciones de *seaborn* aceptan un `DataFrame` y los nombres de las columnas que actúan como ejes, color, tamaño, etc. Para sacarle partido, los datos deben estar en *formato largo*: una fila por observación, con columnas para cada variable. Si tus datos están en formato ancho (una columna por categoría), conviértelos antes con `pd.melt` (cap. 12).

13.2.2. Gráficos estadísticos esenciales

`histplot` y `kdeplot`. Histograma con opción de superponer una estimación de densidad por núcleo (KDE):

```
import seaborn as sns
import numpy as np
import pandas as pd

rng = np.random.default_rng(0)
df = pd.DataFrame({
    "latencia": np.concatenate([rng.normal(50, 5, 1000),
                                rng.normal(70, 10, 500)]),
    "servicio": ["A"] * 1000 + ["B"] * 500
})
```

```
↪ sns.histplot(data=df, x="latencia", hue="servicio", kde=True, bins=40)
```

scatterplot y regplot. Dispersión, opcionalmente con ajuste por regresión:

```
df = pd.DataFrame({
    "x": rng.normal(size=200),
})
df["y"] = 0.5 * df["x"] + rng.normal(scale=0.3, size=200)
sns.regplot(data=df, x="x", y="y", line_kws={"color": "red"})
```

boxplot y violinplot. Comparan distribuciones por categoría. violinplot añade una estimación de densidad lateral:

```
↪ sns.boxplot(data=df_eventos, x="servicio", y="latencia", hue="entorno")
sns.violinplot(data=df_eventos, x="servicio", y="latencia")
```

heatmap. Especialmente útil para matrices de correlación:

```
corr = df.select_dtypes(include="number").corr()
sns.heatmap(corr, annot=True, cmap="coolwarm", center=0)
```

pairplot. Matriz de gráficos de dispersión entre todos los pares de variables numéricas. Muy útil en análisis exploratorio:

```
sns.pairplot(df, hue="clase", diag_kind="kde")
```

13.2.3. Estilos y paletas

seaborn ofrece estilos globales con `sns.set_theme(style=..., palette=...)`. Algunos estilos comunes son `"whitegrid"`, `"darkgrid"` y `"ticks"`. Para paletas de color: `"deep"`, `"colorblind"` (especialmente recomendable en publicaciones por accesibilidad), `"viridis"` (perceptualmente uniforme, ideal para escalas continuas).

```
sns.set_theme(style="whitegrid", palette="colorblind")
```

13.3. Aplicación: visualización de registros de actividad

Mostramos a continuación cuatro patrones recurrentes de visualización en análisis de seguridad. Todos presuponen que los datos están en un `DataFrame` pandas – la salida natural del capítulo anterior.

13.3.1. Distribución de tamaños de paquete

Una distribución muy sesgada o con varios picos suele revelar una mezcla de tráfico heterogéneos (HTTP corto + transferencias largas, por ejemplo). Un histograma en escala logarítmica facilita la lectura:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(7, 4))
ax.hist(df_paquetes["tamaño"], bins=80)
ax.set_xscale("log")
ax.set_xlabel("Tamaño del paquete (bytes, escala log)")
ax.set_ylabel("Frecuencia")
ax.set_title("Distribución de tamaños de paquete")
```

13.3.2. Mapa de calor de actividad por hora y día

Combinando `pandas.pivot` con `seaborn.heatmap` generamos un mapa que revela patrones horarios y semanales (jornadas laborales, actividad fuera de horario, etc.):

```
import seaborn as sns

df_eventos["hora"] = df_eventos["timestamp"].dt.hour
df_eventos["dia"] = df_eventos["timestamp"].dt.day_name()

tabla = (df_eventos.groupby(["dia", "hora"]).size()
         .unstack(fill_value=0))

fig, ax = plt.subplots(figsize=(10, 4))
sns.heatmap(tabla, cmap="YlOrRd", ax=ax)
ax.set_title("Eventos por día y hora")
```

13.3.3. Serie temporal con umbrales

Para detectar visualmente picos en una serie temporal, conviene superponer la media móvil y un umbral (e.g., $\mu + 3\sigma$):

```
ts = df_eventos.set_index("timestamp").resample("h").size()
mm = ts.rolling(24).mean()
umbral = mm + 3 * ts.rolling(24).std()
```

```
fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(ts.index, ts, label="eventos/hora", linewidth=0.8)
ax.plot(mm.index, mm, label="media móvil 24h", color="orange",
↪      )
↪ ax.plot(umbral.index, umbral, label="mu+3sigma", color="red",
      linestyle="--")
ax.legend()
ax.set_title("Eventos por hora con detección de picos")
```

13.3.4. Matriz de correlación de features

Antes de entrenar un modelo de aprendizaje (capítulo siguiente), es muy útil inspeccionar las correlaciones entre las variables de entrada para detectar redundancia:

```
sns.set_theme(style="white")
fig, ax = plt.subplots(figsize=(7, 6))
sns.heatmap(df_features.corr(), annot=True, fmt=".2f",
            cmap="coolwarm", center=0, ax=ax)
ax.set_title("Correlación entre variables")
```

13.4. Ejercicios

EJERCICIO 13.1

Escribe un programa en Python que cargue un fichero CSV con tamaños de paquete (columna `tamano`, en bytes) y muestre dos gráficas en una sola figura: a la izquierda, un histograma con escala lineal; a la derecha, el mismo histograma con escala logarítmica en el eje X. Las dos gráficas deben tener título, etiquetas en los ejes y un número razonable de *bins*.

El programa debe usar `matplotlib.pyplot.subplots` con 1, 2 paneles y guardar la figura resultante en un fichero «`tamanos.png`» con resolución 150 dpi.

Solución (girar para leer)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Datos sintéticos: tamaños de paquete en bytes (cola
5   larga)
6 rng = np.random.default_rng(0)
7 tamaño = rng.lognormal(mean=6.0, sigma=1.2, size=5000)
8 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
9
10 # Panel izquierdo: histograma en escala lineal
11 ax1.hist(tamaño, bins=40, color="steelblue", edgecolor="
12   white")
13 ax1.set_title("Escala lineal")
14 ax1.set_xlabel("Tamaño (bytes)")
15 ax1.set_ylabel("Frecuencia")
16 # Panel derecho: mismo histograma con eje X logarítmico
17 bins_log = np.logspace(np.log10(tamaño.min()), np.log10
18   (tamaño.max()), 40)
19 ax2.hist(tamaño, bins=bins_log, color="darkorange",
20   edgecolor="white")
21 ax2.set_xscale("log")
22 ax2.set_title("Escala logarítmica")
23 ax2.set_xlabel("Tamaño (bytes)")
24 ax2.set_ylabel("Frecuencia")
25 fig.tight_layout()
26 fig.savefig("tamanos.png", dpi=150)

```

Código 13.1: Solución al Ejercicio 13.1.

EJERCICIO 13.2

Dado un fichero CSV de eventos con una columna de timestamp (formato ISO) y un identificador de usuario, escribe un programa que genere un mapa de calor de actividad con días de la semana en el eje Y y horas del día (0–23) en el eje X. Las celdas deben mostrar el número total de eventos para esa combinación.

Usa `pandas` para agrupar y `seaborn.heatmap` para la visualización. La figura debe guardarse como «heatmap.pdf».

Solución (girar para leer)

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5
6 # Datos sintéticos: eventos con timestamp ISO e id de usuario
7 rng = np.random.default_rng(0)
8 n = 20000
9 inicio = pd.Timestamp("2024-01-01")
10 desplaz = rng.integers(0, 30 * 24 * 60, size=n) # minutos sobre un mes
11 df = pd.DataFrame({
12     "timestamp": inicio + pd.to_timedelta(desplaz, unit="m"),
13     "usuario": rng.integers(1, 50, size=n),
14 })
15
16 # Extraer día de la semana (0=Lunes) y hora del día
17 df["día"] = df["timestamp"].dt.dayofweek
18 df["hora"] = df["timestamp"].dt.hour
19
20 # Agrupar contando eventos por (día, hora)
21 tabla = df.groupby(["día", "hora"]).size().unstack()
22 días = ["Lun", "Mar", "Mié", "Jue", "Vie", "Sab", "Dom"]
23 tabla.index = días
24
25 fig, ax = plt.subplots(figsize=(10, 4))
26 sns.heatmap(tabla, cmap="viridis", ax=ax, cbar_kws={"label": "Eventos"})
27 ax.set_xlabel("Hora del día")
28 ax.set_ylabel("Día de la semana")
29 ax.set_title("Mapa de calor de actividad")
30 fig.tight_layout()
31 fig.savefig("heatmap.pdf")

```

Código 13.2: Solución al Ejercicio 13.2.

EJERCICIO 13.3

A partir de un fichero CSV con dos columnas (**servicio** y **latencia_ms**), escribe un programa que dibuje un *boxplot* comparando las distribuciones de latencias por servicio. Identifica visualmente los servicios con mayor mediana y los que tienen más valores atípicos.

Genera la figura tanto con **matplotlib** como con **seaborn** (en paneles distintos) para apreciar la diferencia de estilo por defecto.

Solución (girar para leer)

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5
6 # Datos sintéticos: latencia (ms) por servicio, con
  valores atípicos
7 rng = np.random.default_rng(0)
8 servicios = ["auth", "api", "db", "cache"]
9 medias = {"auth": 40, "api": 80, "db": 120, "cache":
  20}
10 filas = []
11 for s in servicios:
12     base = rng.normal(medias[s], 10, size=200)
13     atip = rng.normal(medias[s] * 4, 20, size=8) #
  outliers
14     for v in np.concatenate([base, atip]):
15         filas.append((s, v))
16     df = pd.DataFrame(filas, columns=["servicio", "
  latencia-ms"])
17
18 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 4))
19
20 # Panel izquierdo: boxplot con matplotlib
  datos = [df[df.servicio == s].latencia-ms.values for s
  in servicios]
21
22 ax1.boxplot(datos, tick_labels=servicios)
23 ax1.set_title("matplotlib")
24 ax1.set_xlabel("Servicio")
25 ax1.set_ylabel("Latencia (ms)")
26
27 # Panel derecho: boxplot con seaborn
  sns.boxplot(data=df, x="servicio", y="latencia-ms", ax=
  ax2)
28
29 ax2.set_title("seaborn")
30 ax2.set_xlabel("Servicio")
31 ax2.set_ylabel("Latencia (ms)")
32
33 fig.tight_layout()
34 fig.savefig("boxplot.png", dpi=150)

```

Código 13.3: Solución al Ejercicio 13.3.

EJERCICIO 13.4

Dado un fichero CSV con una columna de timestamp (granularidad de minutos) y una columna numérica **eventos**, escribe un programa que:

1. Remuestree los datos a frecuencia horaria sumando.
2. Calcule la media móvil de 24 horas y el umbral $\mu + 3\sigma$ (con σ también en ventana móvil de 24 h).
3. Dibuje la serie original, la media móvil y el umbral en la misma gráfica.
4. Marque con un círculo rojo los puntos que superan el umbral.

Solución (girar para leer)

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 # Datos sintéticos: eventos por minuto durante 20 días,
  con picos
6 rng = np.random.default_rng(0)
7 idx = pd.date_range("2024-01-01", periods=20 * 24 * 60,
  freq="min")
8 base = rng.poisson(5, size=len(idx))
9 base[5000:5060] += 80 # ráfaga anómala
10 serie = pd.Series(base, index=idx, name="eventos")
11
12 # 1. Remuestrear a frecuencia horaria sumando
  hora = serie.resample("h").sum()
13
14 # 2. Media móvil y umbral mu + 3 sigma (ventana de 24 h
  media = hora.rolling(24, min_periods=1).mean()
15 desv = hora.rolling(24, min_periods=1).std().fillna(0)
16 umbral = media + 3 * desv
17
18 # 3-4. Dibujar serie, media móvil, umbral y marcar
  anomalías
19 fig, ax = plt.subplots(figsize=(11, 4))
20 ax.plot(hora.index, hora, label="Eventos", color=
  "steelblue")
21 ax.plot(media.index, media, label="Media móvil 24 h",
  color="green")
22 ax.plot(umbral.index, umbral, label="Umbral mu+3sigma",
  color="orange", ls="--")
23 anom = hora[hora > umbral]
24 ax.scatter(anom.index, anom, color="red", zorder=5,
  label="Anomalías")
25 ax.set_xlabel("Tiempo")
26 ax.set_ylabel("Eventos por hora")
27 ax.set_title("Detección de anomalías en serie temporal")
28
29 ax.legend()
30 fig.tight_layout()
31 fig.savefig("anomalías.png", dpi=150)

```

Código 13.4: Solución al Ejercicio 13.4.

EJERCICIO 13.5

Carga un fichero CSV con varias columnas numéricas y una columna categórica **clase** (con 2 o 3 valores). Genera un pairplot de seaborn coloreando los puntos por la clase, y guárdalo como «pairplot.png». ¿Se aprecia visualmente alguna combinación de variables que separe bien las clases?

Solución (girar para leer)

```

1 import numpy as np
2 import pandas as pd
3 import seaborn as sns
4
5 # Datos sintéticos: 3 clases con medias distintas en 4
6   variables
7 rng = np.random.default_rng(0)
8   centros = {"normal": [0, 0, 0, 0], "sospechoso": [3, 3,
9   "malicioso": [6, 1, 4, 2]}
10   filas = []
11   for clase, c in centros.items():
12     muestras = rng.normal(c, 1.0, size=(150, 4))
13     for m in muestras:
14       filas.append([*m, clase])
15   df = pd.DataFrame(filas, columns=["v1", "v2", "v3", "v4",
16   "clase"])
17   # Pairplot coloreado por la clase
18   g = sns.pairplot(df, hue="clase", diag_kind="kde",
19   corner=False)
20   g.fig.suptitle("Relaciones entre variables por clase",
21   y=1.02)
22   g.savefig("pairplot.png", dpi=150)

```

Código 13.5: Solución al Ejercicio 13.5.

EJERCICIO 13.6

Dado un fichero CSV con $n \geq 6$ columnas numéricas, calcula la matriz de correlación de Pearson entre todas las columnas y dibújala como un mapa de calor anotado con los valores numéricos. La diagonal debe quedar resaltada. Identifica los pares de variables con correlación absoluta superior a 0.8.

Solución (girar para leer)

```

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Datos sintéticos: 6 variables, algunas fuertemente
# correladas
rng = np.random.default_rng(0)
n = 500
a = rng.normal(size=n)
b = rng.normal(size=n)
df = pd.DataFrame({
    "v1": a,
    "v2": a * 0.95 + rng.normal(0, 0.1, n), # muy
    "v3": b,
    "v4": b * -0.9 + rng.normal(0, 0.1, n), #
    "v5": rng.normal(size=n),
    "v6": rng.normal(size=n),
})

# Matriz de correlacion de Pearson
corr = df.corr()

# Mapa de calor anotado con diagonal resaltada
fig, ax = plt.subplots(figsize=(7, 6))
sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm",
            center=0,
            linewidths=0.5, linecolor="white", ax=ax)
for i in range(len(corr)):
    ax.add_patch(plt.Rectangle((i, i), 1, 1, fill=False))
    edgcolor="black", lw=2)
ax.set_title("Matriz de correlacion de Pearson")
fig.tight_layout()
fig.savefig("correlacion.png", dpi=150)

# Pares con correlacion absoluta superior a 0.8
for i in range(len(corr)):
    for j in range(i + 1, len(corr)):
        if abs(corr.iloc[i, j]) > 0.8:
            print(f"{corr.columns[i]} - {corr.columns[j]}: {corr.iloc[i, j]:.2f}")

```

EJERCICIO 13.7

Combina varias técnicas para construir un *dashboard* compacto en una única figura, con cuatro paneles dispuestos en una rejilla 2×2:

1. Histograma de la duración de las sesiones.
2. Boxplot de tráfico (bytes) por categoría de usuario.
3. Serie temporal de eventos por hora del día.
4. Mapa de calor de actividad por día y hora.

Usa `plt.subplots(2, 2)` y `fig.tight_layout()` para que los paneles no se solapen. Guarda el resultado como «`dashboard.pdf`».

Solución (girar para leer)

```

7  rng = np.random.default_rng(0)
8  n = 4000
9  ts = pd.Timestamp("2024-01-01") + pd.to_timedelta(
10     rng.integers(0, 7 * 24 * 60, n), unit="m")
11  df = pd.DataFrame({
12     "duracion": rng.lognormal(3, 0.6, n),
13     "bytes": rng.lognormal(8, 1.0, n),
14     "categoria": rng.choice(["nuevo", "habitual", "
15     "timestamp": ts,
16     })
17  df["dia"] = df["timestamp"].dt.dayofweek
18  df["hora"] = df["timestamp"].dt.hour
19
20  fig, axs = plt.subplots(2, 2, figsize=(11, 8))
21
22  # 1. Histograma de duracion de sesiones
23  axs[0, 0].hist(df["duracion"], bins=40, color="
24     steelblue")
25  axs[0, 0].set_title("Duracion de sesiones")
26  axs[0, 0].set_xlabel("Segundos")
27  axs[0, 0].set_ylabel("Frecuencia")
28
29  # 2. Boxplot de trafico por categoria
30  sns.boxplot(data=df, x="categoria", y="bytes", ax=axs
31     [0, 1])
32  axs[0, 1].set_title("Trafico por categoria")
33
34  # 3. Serie de eventos por hora del dia
35  por_hora = df.groupby("hora").size()
36  axs[1, 0].plot(por_hora.index, por_hora.values, marker=
37     "o")
38  axs[1, 0].set_title("Eventos por hora")
39  axs[1, 0].set_xlabel("Hora")
40  axs[1, 0].set_ylabel("Eventos")
41
42  # 4. Mapa de calor de actividad por dia y hora
43  tabla = df.groupby(["dia", "hora"]).size().unstack(
44     fill_value=0)
45  sns.heatmap(tabla, cmap="viridis", ax=axs[1, 1])
46  axs[1, 1].set_title("Actividad dia/hora")
47  fig.tight_layout()
48  fig.savefig("dashboard.pdf")

```

14

Introducción al aprendizaje automático con scikit-learn

Llegamos al final del libro con los primeros pasos en **aprendizaje automático**. En este capítulo introduciremos los conceptos fundamentales del *aprendizaje automático* (*machine learning*, ML) y los aplicaremos a algunos problemas concretos, varios de ellos tomados del ámbito de la seguridad: clasificar correos como *spam* o legítimos, distinguir tráfico benigno de malicioso, detectar conexiones anómalas en una red. Trabajaremos con **scikit-learn**, la biblioteca de referencia para ML clásico en Python: ofrece decenas de algoritmos con una API uniforme, herramientas de preprocesado, validación y evaluación, y una documentación excelente. Al construirse sobre **numpy** y **pandas**, todo lo aprendido en los dos capítulos anteriores es directamente reutilizable.

No es nuestro objetivo cubrir el ML en profundidad: el lector interesado encontrará tratados completos en libros y cursos dedicados. Lo que queremos es dar una primera vista **operativa** del flujo de trabajo (carga, preprocesado, entrenamiento, evaluación, predicción) y mostrar cómo se manifiesta este flujo en escenarios reales, varios de ellos del ámbito de la ciberseguridad.

14.1. Conceptos fundamentales

14.1.1. Aprendizaje supervisado y no supervisado

Los problemas de ML se agrupan en dos grandes familias:

- **Aprendizaje supervisado:** disponemos de pares (*entrada, salida correcta*). Un

algoritmo busca una función que prediga la salida a partir de la entrada. Ejemplos: clasificar un correo como *spam* (clase $\in \{0, 1\}$) o predecir el número de visitas de mañana (regresión, salida continua).

- **Aprendizaje no supervisado:** solo disponemos de entradas, sin etiquetas. El algoritmo busca estructura: agrupar observaciones similares (*clustering*), detectar anomalías o reducir dimensionalidad. Es muy frecuente en seguridad, donde rara vez tenemos etiquetas masivas y bien curadas.

Entre ambos extremos hay variantes (*semi-supervisado*, *auto-supervisado*, *aprendizaje por refuerzo*) que no abordaremos aquí.

14.1.2. Variables, features y vectores de datos

En ML se trabaja típicamente con dos estructuras:

- **X:** matriz de *features* (entradas) de forma (**n_muestras**, **n_features**). Cada fila es una observación, cada columna una variable.
- **y:** vector de etiquetas (salidas) de longitud **n_muestras**.

En *scikit-learn*, **X** e **y** son *ndarray* de *numpy* o *DataFrame*/*Series* de *pandas* – ambos funcionan.

14.1.3. Entrenamiento, validación y prueba

El error que un modelo comete *sobre los datos con los que se entrenó* es siempre demasiado optimista: el modelo puede haber memorizado peculiaridades del conjunto de entrenamiento (*sobreajuste* o *overfitting*). Para estimar el rendimiento real, se reserva una parte de los datos como conjunto de **prueba** (*test*) que el modelo nunca ve durante el entrenamiento. Una división típica es 70 %-80 % para entrenamiento y 20 %-30 % para prueba.

Cuando se quiere ajustar *hiperparámetros* (parámetros que no aprende el algoritmo, sino que debemos elegir nosotros), se reserva además un conjunto de **validación** o se utiliza *validación cruzada* sobre el entrenamiento.

14.1.4. Métricas de evaluación

Para clasificación binaria, las métricas más utilizadas son:

- **Exactitud** (*accuracy*): proporción de aciertos. Engaña en problemas desbalanceados (si el 99 % del tráfico es benigno, predecir siempre “benigno” acierta el 99 %).
- **Precisión** (*precision*): de las predicciones positivas, cuántas son correctas. Importa cuando un falso positivo es costoso (cerrar una cuenta legítima por sospecha).

- **Sensibilidad** (*recall*): de los casos positivos reales, cuántos detectamos. Importa cuando un falso negativo es peligroso (no detectar un *malware*).
- **F1**: media armónica de precisión y sensibilidad. Balancea ambas.
- **Matriz de confusión**: tabla 2×2 con verdaderos/falsos positivos/negativos.
- **Curva ROC y AUC**: relación entre tasa de verdaderos positivos y de falsos positivos al variar el umbral de decisión. Útil para comparar modelos independientemente del umbral.

En ciberseguridad, el desbalance es la norma

Los datasets de seguridad están casi siempre desbalanceados: hay muchísimas más observaciones benignas que maliciosas. La *accuracy* no es una métrica adecuada en estos contextos: prefiere *precision*, *recall*, F1 y curvas ROC/PR (*precision-recall*). Y cuando entrenes, considera técnicas como ponderar las clases (`class_weight="balanced"`) o submuestrear/sobremuestrear.

14.2. La API de *scikit-learn*

Una de las virtudes de *scikit-learn* es su API uniforme. Casi todos los algoritmos siguen el mismo patrón, con tres métodos principales:

- `modelo.fit(X, y)`: ajusta (entrena) el modelo a los datos.
- `modelo.predict(X)`: produce predicciones para nuevas entradas.
- `modelo.score(X, y)`: evalúa el rendimiento (depende del estimador).

Esto significa que cambiar de algoritmo es esencialmente cambiar la línea `modelo = ...`; el resto del código sigue igual.

14.2.1. Un ejemplo mínimo de clasificación

Vamos a clasificar muestras del clásico dataset Iris (no es de seguridad, pero sirve como “Hola, mundo” de ML):

```
1 from sklearn.datasets import load_iris
2 from sklearn.model_selection import train_test_split
3 from sklearn.linear_model import LogisticRegression
4 from sklearn.metrics import accuracy_score,
  ↳ classification_report
5
```

```
6 # 1. Cargar datos
7 iris = load_iris()
8 X, y = iris.data, iris.target
9
10 # 2. Dividir en entrenamiento / prueba
11 X_train, X_test, y_train, y_test = train_test_split(
12     X, y, test_size=0.3, random_state=42, stratify=y)
13
14 # 3. Entrenar
15 modelo = LogisticRegression(max_iter=1000)
16 modelo.fit(X_train, y_train)
17
18 # 4. Predecir y evaluar
19 y_pred = modelo.predict(X_test)
20 print("Accuracy:", accuracy_score(y_test, y_pred))
21 print(classification_report(y_test, y_pred,
22                             target_names=iris.target_names))
```

Algunos puntos clave:

- `train_test_split` con `stratify=y` preserva la proporción de clases entre los dos subconjuntos. Importante en problemas desbalanceados.
- `random_state=42` fija la semilla de la división para reproducibilidad.
- `classification_report` devuelve precisión, sensibilidad y F1 por clase, que es mucho más informativo que la *accuracy* sola.

14.2.2. Preprocesado: StandardScaler y LabelEncoder

Muchos algoritmos asumen que las variables están en escalas comparables (regresión logística, SVM, redes neuronales) o requieren entradas numéricas (no categóricas). Las dos herramientas más usadas son:

- **StandardScaler**: estandariza cada variable restando la media y dividiendo por la desviación típica, dejándola con $\mu = 0$ y $\sigma = 1$.
- **OneHotEncoder**: convierte una variable categórica de k valores en k columnas binarias.

```
↪ from sklearn.preprocessing import StandardScaler,
    OneHotEncoder

scaler = StandardScaler()
X_train_esc = scaler.fit_transform(X_train)
X_test_esc = scaler.transform(X_test)
```

Ajusta el preprocesado solo con *train*

Es un error muy común aplicar `fit_transform` sobre todos los datos antes de dividir. Esto introduce *data leakage*: el conjunto de prueba influye en la media y la desviación con que se escala el entrenamiento, y las métricas de prueba quedan optimistas. La regla es: `fit_transform` solo con `X_train` y `transform` (sin `fit`) con `X_test`.

14.2.3. Pipelines

Para no tener que recordar el orden de las transformaciones, *scikit-learn* ofrece la clase `Pipeline`, que encadena pasos de preprocesado y un estimador final en un único objeto. Es la forma recomendada de trabajar:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("clf", LogisticRegression(max_iter=1000)),
])
pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

`Pipeline` aplica `fit_transform` en cada paso al entrenar y `transform` al predecir, evitando el *leakage* y manteniendo el código limpio.

14.2.4. Validación cruzada

Para estimar el rendimiento de forma más robusta que con una única división, se usa *k-fold cross-validation*: se parte el conjunto de entrenamiento en *k* pliegues, se entrena *k* veces dejando uno fuera y se promedian las métricas:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(pipe, X_train, y_train, cv=5,
    ↪      scoring="f1_macro")
print(f"F1 medio: {scores.mean():.3f} +- {scores.std():.3f}")
```

14.2.5. Búsqueda de hiperparámetros: GridSearchCV

`GridSearchCV` explora combinaciones de hiperparámetros con validación cruzada y devuelve la mejor:

```
from sklearn.model_selection import GridSearchCV

grid = {
    "clf__C": [0.1, 1.0, 10.0],
    "clf__penalty": ["l2"],
}
busqueda = GridSearchCV(pipe, grid, cv=5, scoring="f1_macro",
    ↪ n_jobs=-1)
busqueda.fit(X_train, y_train)
print("Mejores parámetros:", busqueda.best_params_)
print("Mejor F1:", busqueda.best_score_)
```

La notación `clf__C` accede al hiperparámetro `C` del paso `clf` del pipeline.

14.3. Algoritmos clásicos

scikit-learn implementa decenas de algoritmos. Recogemos los más utilizados, agrupados por familia.

14.3.1. Clasificación

- `LogisticRegression`: regresión logística. Lineal, interpretable, base de comparación natural.
- `KNeighborsClassifier`: k vecinos más cercanos. No paramétrico; sensible a la escala (estandariza siempre).
- `DecisionTreeClassifier`: árbol de decisión. Interpretable, propenso al sobreajuste si no se poda.
- `RandomForestClassifier`: bosque aleatorio. Robusto, buen punto de partida.
- `GradientBoostingClassifier` y `HistGradientBoostingClassifier`: *boosting*. Suelen dar el mejor rendimiento en problemas tabulares (a menudo `xgboost` / `lightgbm` si se necesita aún más rendimiento).
↪
- `SVC`: máquinas de vector de soporte. Potentes pero costosas en datasets grandes.
- `MLPClassifier`: perceptrón multicapa. Para redes serias, salta a `pytorch` o `tensorflow`.

14.3.2. Regresión

Para predecir valores continuos: `LinearRegression`, `Ridge`, `Lasso`, `RandomForestRegressor`, `GradientBoostingRegressor`, `SVR`.

14.3.3. Clustering

Para agrupar sin etiquetas: KMeans, DBSCAN, AgglomerativeClustering, GaussianMixture.

14.3.4. Detección de anomalías

Específicamente útil en seguridad: IsolationForest, OneClassSVM, LocalOutlierFactor. Aprenden la “normalidad” y marcan como anómalo lo que se aparta.

14.3.5. Reducción de dimensionalidad

Para visualizar o comprimir: PCA (lineal, rápido), TSNE (no lineal, costoso, ideal para 2D/3D), UMAP (similar a t-SNE pero más rápido; en biblioteca aparte).

14.4. Aplicación: detección de spam

Como primer caso práctico, construimos un clasificador de *spam* a partir de texto. Es un problema clásico de seguridad y de procesamiento de lenguaje natural.

```
1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.feature_extraction.text import TfidfVectorizer
4 from sklearn.linear_model import LogisticRegression
5 from sklearn.pipeline import Pipeline
6 from sklearn.metrics import classification_report,
↪ confusion_matrix
7
8 # Dataset: cada fila tiene 'texto' y 'etiqueta' (0=ham, 1=
↪ spam)
9 df = pd.read_csv("correos.csv")
10 X = df["texto"]
11 y = df["etiqueta"]
12
13 X_train, X_test, y_train, y_test = train_test_split(
14     X, y, test_size=0.25, random_state=42, stratify=y)
15
```

```
16 pipe = Pipeline([
17     ("tfidf", TfidfVectorizer(min_df=2, ngram_range=(1, 2),
18                             stop_words="english")),
19     ("clf", LogisticRegression(max_iter=1000, class_weight=
    ↪     "balanced")),
20 ])
21 pipe.fit(X_train, y_train)
22 y_pred = pipe.predict(X_test)
23
24 print(classification_report(y_test, y_pred, target_names=["
    ↪     ham", "spam"]))
25 print(confusion_matrix(y_test, y_pred))
```

Qué está pasando.

- `TfidfVectorizer` convierte cada texto en un vector numérico ponderando los términos por frecuencia inversa de documento (TF-IDF). `ngram_range=(1, 2)` usa palabras y bigramas; `min_df=2` ignora términos que aparecen solo en un correo.
- `class_weight="balanced"` pondera las clases inversamente a su frecuencia, mitigando el desbalance.
- El pipeline aplica TF-IDF sobre el entrenamiento y la *misma* transformación al test.
- Mostramos `classification_report` y la matriz de confusión, donde se distinguen los falsos positivos (ham marcados como spam) y falsos negativos (spam que se nos cuela).

14.5. Aplicación: clasificación de tráfico de red

A partir de *features* numéricas extraídas de capturas de tráfico (duración de conexión, bytes enviados/recibidos, número de paquetes, flags TCP...), entrenamos un clasificador para distinguir tráfico benigno de ataques. Usamos `RandomForestClassifier`, que tolera bien las variables sin escalar y suele ser un punto de partida competitivo:

```
1 import pandas as pd
2 from sklearn.model_selection import train_test_split,
    ↪     cross_val_score
3 from sklearn.ensemble import RandomForestClassifier
4 from sklearn.metrics import classification_report,
    ↪     roc_auc_score
5
```

```

6 df = pd.read_csv("trafico.csv")      # con columna 'etiqueta'
↪   (0=benigno, 1=ataque)
7 X = df.drop(columns=["etiqueta"])
8 y = df["etiqueta"]
9
10 X_train, X_test, y_train, y_test = train_test_split(
11     X, y, test_size=0.25, random_state=42, stratify=y)
12
13 clf = RandomForestClassifier(
14     n_estimators=300, max_depth=None,
15     class_weight="balanced", random_state=42, n_jobs=-1)
16 clf.fit(X_train, y_train)
17
18 y_pred = clf.predict(X_test)
19 y_proba = clf.predict_proba(X_test)[: , 1]
20
21 print(classification_report(y_test, y_pred, target_names=["
↪     benigno", "ataque"]))
22 print(f"AUC ROC: {roc_auc_score(y_test, y_proba):.3f}")
23
24 # Importancia de las variables
25 importancia = pd.Series(clf.feature_importances_, index=X.
↪     columns)
26 print(importancia.sort_values(ascending=False).head(10))

```

Notas.

- `predict_proba` devuelve la probabilidad estimada de cada clase. La AUC ROC se calcula con la probabilidad de la clase positiva.
- `feature_importances_` permite identificar qué variables influyen más en la decisión, lo que ayuda a entender el modelo y a detectar *features* potencialmente fugadas (*data leakage*: variables que codifican accidentalmente la etiqueta).

14.6. Aplicación: detección de anomalías sin etiquetas

A menudo no tenemos etiquetas: solo sabemos que la mayoría del tráfico es “normal” y queremos descubrir lo distinto. `IsolationForest` es un detector de anomalías eficaz y rápido que asigna a cada muestra una puntuación; las muestras con puntuación más baja son las más anómalas:

```

1 import pandas as pd
2 from sklearn.ensemble import IsolationForest
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.pipeline import Pipeline
5
6 df = pd.read_csv("conexiones.csv")
7 X = df[["duracion", "bytes_enviados", "bytes_recibidos", "
↪ paquetes"]]
8
9 modelo = Pipeline([
10     ("scaler", StandardScaler()),
11     ("if", IsolationForest(contamination=0.01, random_state
↪ =42)),
12 ])
13 modelo.fit(X)
14
15 # +1 = normal, -1 = anomalia
16 df["anomalia"] = modelo.predict(X) == -1
17 df["score"] = -modelo["if"].score_samples(modelo["scaler"
↪ ].transform(X))
18
19 # Top 10 mas anomalas
20 print(df.sort_values("score", ascending=False).head(10))

```

Contamination y validación

contamination es la fracción esperada de anomalías. Si lo desconoces, prueba con varios valores y revisa manualmente los resultados; en producción, conviene comparar puntuaciones contra un umbral fijo en lugar de la decisión binaria `predict`. Sin etiquetas, validar un detector de anomalías es difícil: lo habitual es combinar inspección manual con reglas conocidas.

14.7. Cierre del libro

↪ Hemos recorrido un camino largo: del primer `print("Hola, mundo desde Python")` al entrenamiento de un clasificador con un bosque aleatorio, aplicado por ejemplo a la detección de tráfico malicioso. Por el camino hemos visto los tipos básicos del lenguaje, las estructuras compuestas, la entrada/salida, las dos bibliotecas que dan acceso al ecosistema de datos (`numpy` y `pandas`), las herramientas de visualización (`matplotlib` y `seaborn`) y, finalmente, los conceptos básicos del aprendizaje automático con `scikit-learn`.

↪ Por supuesto, esto es solo un punto de partida. El aprendizaje automático y, en general, la ciencia de datos forman un campo en evolución muy rápida, donde conviven

técnicas clásicas como las que hemos visto con redes neuronales profundas, modelos de lenguaje y agentes autónomos. La ciberseguridad es uno de los dominios donde estas técnicas encuentran aplicación. Algunas direcciones naturales para seguir aprendiendo:

- **Redes neuronales profundas:** `pytorch` o `tensorflow` son los dos *frameworks* dominantes. Para texto, los modelos de tipo *transformer* (a través de `transformers` de Hugging Face) son hoy el estándar.
- **Procesado de lenguaje natural:** análisis de logs, descripciones de vulnerabilidades, correos de *phishing*.
- **Modelos generativos:** detección de contenido sintético, simulación de adversarios.
- **Aprendizaje por refuerzo:** agentes que aprenden a explorar o defender entornos.
- **Adversarial machine learning:** cómo atacar y defender modelos.

Esperamos que el libro te haya servido como base sólida. El resto depende de tu curiosidad y de los problemas concretos en los que decidas embarcarte. ¡Buena suerte!

14.8. Ejercicios

EJERCICIO 14.1

Carga el dataset Iris de `sklearn.datasets` y compara cuatro clasificadores: `LogisticRegression`, `KNeighborsClassifier` (con $k = 5$), `DecisionTreeClassifier` y `RandomForestClassifier`. Para cada uno, usa un `Pipeline` con `StandardScaler` y reporta la *accuracy* media y desviación típica obtenidas con validación cruzada de 10 pliegues. ¿Cuál gana? ¿Justifica el coste extra?

Solución (girar para leer)

```

1 from sklearn.datasets import load_iris
2 from sklearn.model_selection import cross_val_score
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.pipeline import Pipeline
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.neighbors import KNeighborsClassifier
7 from sklearn.tree import DecisionTreeClassifier
8 from sklearn.ensemble import RandomForestClassifier
9 X, y = load_iris(return_X_y=True)
10 # Cuatro clasificadores a comparar
11 modelos = {
12     "LogisticRegression": LogisticRegression(max_iter
13         =1000, random_state=0),
14     "KNN(k=5)": KNeighborsClassifier(
15         n_neighbors=5),
16     "DecisionTree": DecisionTreeClassifier(
17         random_state=0),
18     "RandomForest": RandomForestClassifier(
19         n_estimators=100, random_state=0),
20 }
21 # Validacion cruzada de 10 pliegues con escalado dentro
22     del pipeline
23     for nombre, clf in modelos.items():
24         pipe = Pipeline([("scaler", StandardScaler()), ("
25             clf", clf)])
26         scores = cross_val_score(pipe, X, y, cv=10, scoring
27             = "accuracy")
28         print(f"{nombre:20s} accuracy = {scores.mean():.3f}
29             +/- {scores.std():.3f}")

```

Código 14.1: Solución al Ejercicio 14.1.

EJERCICIO 14.2

Implementa el clasificador de *spam* de la sección 14.4 y experimenta con tres variaciones: (I) cambia `TfidfVectorizer` por `CountVectorizer`; (II) prueba un `RandomForestClassifier` en lugar de la regresión logística; (III) ajusta `ngram_range` a (1, 3). Compara las matrices de confusión y los F1 obtenidos.

Como dataset puedes utilizar el clásico *SMS Spam Collection* disponible públicamente (también puedes generar uno sintético si prefieres).

Solución (girar para leer)

```

1 import numpy as np
2 from sklearn.model_selection import train_test_split
3 from sklearn.feature_extraction.text import TfidfVectorizer
4 from sklearn.linear_model import LogisticRegression
5 from sklearn.ensemble import RandomForestClassifier
6 from sklearn.pipeline import Pipeline
7 from sklearn.metrics import f1_score, confusion_matrix
8
9 np.random.seed(0)
10
11 # Generar correos sintéticos: ham y spam con
12     vocabulario parcialmente solapado
13     ham = "reunion equipo informe proyecto calendario".
14     spam = "gratis premio dinero oferta urgente gana clic".
15     split() + comun
16     def gen(pal, n):
17         return [" ".join(np.random.choice(pal, 8)) for _ in
18             range(n)]
19     X = gen(ham, 300) + gen(spam, 300)
20     y = np.array([0] * 300 + [1] * 300)
21     Xtr, Xte, ytr, yte = train_test_split(X, y, test_size
22         =0.25, random_state=0, stratify=y)
23     # Tres variaciones del clasificador de spam
24     {
25         "Count + LogReg": Pipeline([("v",
26             CountVectorizer()), ("c", LogisticRegression(
27                 max_iter=1000))]),
28         "Tfidf + RandomForest": Pipeline([("v",
29             TfidfVectorizer()), ("c", RandomForestClassifier
30                 (n_estimators=100, random_state=0))]),
31         "Tfidf(1,3) + LogReg": Pipeline([("v",
32             TfidfVectorizer(ngram_range=(1, 3)), ("c",
33                 LogisticRegression(max_iter=1000))])],
34     }
35     for nombre, pipe in variantes.items():
36         pipe.fit(Xtr, ytr)
37         pred = pipe.predict(Xte)
38         print(f"== {nombre} == F1 = {f1_score(yte, pred)}
39             :.3f}")
40     print(confusion_matrix(yte, pred))

```

EJERCICIO 14.3

A partir del clasificador de tráfico de la sección 14.5, usa `GridSearchCV` para buscar los mejores valores de `n_estimators` ($\in \{100, 300, 500\}$), `max_depth` ($\in \{5, 10, \text{None}\}$) y `min_samples_split` ($\in \{2, 5, 10\}$) optimizando `f1_macro`. Reporta los mejores hiperparámetros y compara las métricas con el modelo por defecto.

Solución (girar para leer)

```

1 from sklearn.datasets import make_classification
2 from sklearn.model_selection import train_test_split,
GridSearchCV
3 from sklearn.ensemble import RandomForestClassifier
4 from sklearn.metrics import f1_score
5
6 # Dataset de tráfico sintético (benigno vs ataque)
7 X, y = make_classification(n_samples=1000, n_features
=12, n_informative=6,
8 weights=[0.7, 0.3],
9 random_state=0)
10 Xtr, Xte, ytr, yte = train_test_split(X, y, test_size
=0.25, random_state=0, stratify=y)
11 # Modelo por defecto como referencia
12 base = RandomForestClassifier(random_state=0)
13 base.fit(Xtr, ytr)
14 print(f"F1 macro por defecto: {f1_score(yte, base.
predict(Xte), average='macro'):.3f}")
15
16 # Búsqueda de hiperparámetros optimizando f1_macro
17 rejilla = {
18     "n_estimators": [100, 300, 500],
19     "max_depth": [5, 10, None],
20     "min_samples_split": [2, 5, 10],
21 }
22 gs = GridSearchCV(RandomForestClassifier(random_state
=0), rejilla,
23 scoring="f1_macro", cv=5, n_jobs=-1)
24 gs.fit(Xtr, ytr)
25 print(f"Mejores hiperparámetros: ", gs.best_params_)
26 print(f"F1 macro ajustado: {f1_score(yte, gs.predict
(Xte), average='macro'):.3f}")

```

Código 14.3: Solución al Ejercicio 14.3.

EJERCICIO 14.4

Genera un dataset sintético con `numpy`: 1000 muestras de tres dimensiones procedentes de una normal multivariante con $\mu = (0,0,0)$ y matriz de covarianza identidad, y mezcla 20 anomalías cuyas coordenadas son uniformes en $[-5,5]$. Entrena un `IsolationForest` con `contamination=0.02` y reporta cuántas de las anomalías inyectadas detecta correctamente y cuántos falsos positivos produce.

Solución (girar para leer)

```

1 import numpy as np
2 from sklearn.ensemble import IsolationForest
3
4 np.random.seed(0)
5
6 # 1000 muestras normales en 3D con media cero y
7   covarianza identidad
8   normales = np.random.multivariate_normal([0, 0, 0], np.
9     eye(3), size=1000)
10 # 20 anomalías uniformes en [-5, 5]
11 anomalías = np.random.uniform(-5, 5, size=(20, 3))
12 X = np.vstack([normales, anomalías])
13 # Verdad: 1 = normal, -1 = anomalía (convención de
14   IsolationForest)
15 y-real = np.array([1] * 1000 + [-1] * 20)
16 clf = IsolationForest(contamination=0.02, random-state
17   =0)
18 pred = clf.fit_predict(X)
19 # Anomalías inyectadas detectadas y falsos positivos
20 detectadas = np.sum((y-real == -1) & (pred == -1))
21 falsos-pos = np.sum((y-real == 1) & (pred == -1))
22 print(f"Anomalías inyectadas: {np.sum(y-real ==
23   -1)}")
24 print(f"Anomalías detectadas: {detectadas}")
25 print(f"Falsos positivos (normales): {falsos-pos}")

```

Código 14.4: Solución al Ejercicio 14.4.

EJERCICIO 14.5

Dado un fichero CSV con *features* numéricas extraídas de eventos de log (sin etiquetas), aplica **KMeans** con k entre 2 y 10 y dibuja la curva del “codo” (*inertia* vs k) para elegir un buen número de clústeres. Una vez elegido, visualiza los clústeres en 2D proyectando con **PCA** y comenta si los grupos parecen razonables (uniformes, bien separados, etc.).

Solución (girar para leer)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.datasets import make_blobs
4 from sklearn.preprocessing import StandardScaler
5 from sklearn.cluster import KMeans
6 from sklearn.decomposition import PCA
7
8 np.random.seed(0)
9
10 # Features numéricas sintéticas de eventos de log (sin
    etiquetas)
11 X, _ = make_blobs(n_samples=600, centers=4, n_features
    =6,
12 cluster_std=1.5, random_state=0)
13 X = StandardScaler().fit_transform(X)
14
15 # Curva del código: inercia frente a k
16 ks = range(2, 11)
17 inercias = [KMeans(n_clusters=k, n_init=10,
    random_state=0).fit(X).inertia_ for k in ks]
18
19 # Numero de clusters elegido tras inspeccionar el código
20 k_elegido = 4
21 etiquetas = KMeans(n_clusters=k_elegido, n_init=10,
    random_state=0).fit_predict(X)
22 Xp = PCA(n_components=2, random_state=0).fit_transform(
    X)
23
24 fig, (a1, a2) = plt.subplots(1, 2, figsize=(11, 4))
25 a1.plot(list(ks), inercias, "o-"); a1.set_xlabel("k");
    a1.set_ylabel("Inercia")
26 a1.set_title("Curva del código")
27 a2.scatter(Xp[:, 0], Xp[:, 1], c=etiquetas, cmap="tab10",
    s=12)
28 a2.set_title(f"Clusters (k={k_elegido}) en PCA 2D")
29 fig.tight_layout()
30 fig.savefig("kmeans-logs.png", dpi=120)
31 print(f"K elegido = {k_elegido}; figura guardada en
    kmeans-logs.png")

```

Código 14.5: Solución al Ejercicio 14.5.

EJERCICIO 14.6

Construye un pipeline que combine: (I) `ColumnTransformer` para aplicar `StandardScaler` a las columnas numéricas y `OneHotEncoder` a las categóricas; y (II) un `RandomForestClassifier` como clasificador final. Aplícalo a un dataset con columnas de ambos tipos (puedes usar el *Adult* o cualquier CSV con *features* mixtas) y entrena/valida con `cross_val_score`.

Solución (girar para leer)

```

1 import numpy as np
2 import pandas as pd
3 from sklearn.compose import ColumnTransformer
4 from sklearn.preprocessing import StandardScaler,
5 OneHotEncoder
6 from sklearn.ensemble import RandomForestClassifier
7 from sklearn.pipeline import Pipeline
8 from sklearn.model_selection import cross_val_score
9 np.random.seed(0)
10 n = 600
11
12 # Dataset con features numéricas y categóricas
13 df = pd.DataFrame({
14     "edad": np.random.randint(18, 70, n),
15     "ingresos": np.random.normal(30000, 9000, n),
16     "pais": np.random.choice(["ES", "FR", "DE"], n),
17     "rol": np.random.choice(["admin", "user", "
18     "guest"], n),
19 })
20 # Etitqueta dependiente de variables numéricas y
21 y = (df["ingresos"] > 30000) ~ (df["rol"] == "admin")
22     .astype(int)
23 num = ["edad", "ingresos"]
24 cat = ["pais", "rol"]
25 pre = ColumnTransformer([
26     ("num", StandardScaler(), num),
27     ("cat", OneHotEncoder(handle_unknown="ignore"), cat
28     ),
29 ])
30 pipe = Pipeline([
31     ("pre", pre),
32     ("clf", RandomForestClassifier(n_estimators=100,
33     random_state=0)),
34 ])
35 scores = cross_val_score(pipe, df, y, cv=5, scoring="
36 accuracy")
37 print(f"Accuracy CV: {scores.mean():.3f} +/- {scores.
38 std():.3f}")

```

Código 14.6: Solución al Ejercicio 14.6.

EJERCICIO 14.7

Entrena dos modelos sobre un mismo problema de clasificación binaria (e.g., `LogisticRegression` y `RandomForestClassifier`). Calcula la curva ROC de cada uno con `sklearn.metrics.roc_curve` y dibújalas en una sola figura con `matplotlib`, mostrando también la diagonal de azar. Anota en la leyenda el AUC de cada modelo.

Código 14.7: Solución al Ejercicio 14.7.

```

1 import matplotlib.pyplot as plt
2 from sklearn.datasets import make_classification
3 from sklearn.model_selection import train_test_split
4 from sklearn.linear_model import LogisticRegression
5 from sklearn.ensemble import RandomForestClassifier
6 from sklearn.metrics import roc_curve, roc_auc_score
7
8 # Problema de clasificación binaria
9 X, y = make_classification(n_samples=1000, n_features
10                             =10, n_informative=5,
11                             random_state=0)
12 Xtr, Xte, ytr, yte = train_test_split(X, y, test_size
13                                       =0.3, random_state=0)
14 modelos = {
15     "LogisticRegression": LogisticRegression(max_iter
16                                               =1000, random_state=0),
17     "RandomForest": RandomForestClassifier(
18         n_estimators=100, random_state=0),
19 }
20 plt.figure(figsize=(6, 5))
21 for nombre, clf in modelos.items():
22     clf.fit(Xtr, ytr)
23     proba = clf.predict_proba(Xte)[:, 1]
24     fpr, tpr, _ = roc_curve(yte, proba)
25     auc = roc_auc_score(yte, proba)
26     plt.plot(fpr, tpr, label=f"{nombre} (AUC = {auc:.3f})")
27
28 plt.plot([0, 1], [0, 1], "k--", label="azar") #
29
30 plt.xlabel("FPR"); plt.ylabel("TPR"); plt.title("Curvas
31 ROC")
32 plt.legend(); plt.tight_layout()
33 plt.savefig("roc-comparativa.png", dpi=120)
34 print("Figura guardada en roc-comparativa.png")

```

EJERCICIO 14.8

Combina lo aprendido en los caps. 12, 13 y este capítulo: a partir de un fichero CSV con tráfico de red, detecta anomalías con `IsolationForest`, exporta las muestras anómalas a otro CSV y dibuja un *scatterplot* en 2D (proyección con PCA) coloreando puntos normales vs anómalos. El programa debe ser un *script* ejecutable con un argumento de línea de comandos para el fichero de entrada.

Solución (girar para leer)

```

1 import sys
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 from sklearn.ensemble import IsolationForest
6 from sklearn.decomposition import PCA
7
8 np.random.seed(0)
9
10 # Si no se pasa CSV, se genera uno sintético de ejemplo
11 if len(sys.argv) > 1:
12     df = pd.read_csv(sys.argv[1])
13 else:
14     normal = np.random.normal(0, 1, (480, 4))
15     anom = np.random.uniform(-6, 6, (20, 4))
16     df = pd.DataFrame(np.vstack([normal, anom]),
17                        columns=["duracion", "bytes-env",
18                               "bytes-rec", "paquetes"])
19 X = df.select_dtypes("number")
20 clf = IsolationForest(contamination=0.04, random_state=0)
21 pred = clf.fit_predict(X)
22 # -1 = anormal,
23     I = normal
24     df["anormal"] = pred == -1
25 # Exportar nuestras anomalías a otro CSV
26 df[df["anormal"]].to_csv("anomalías.csv", index=False)
27 print(f"Anomalías detectadas: {df['anormal'].sum()}")
28 # Scatterplot 2D con proyección PCA, coloreando normal
29     us anormalo
30 Xp = PCA(n_components=2, random_state=0).fit_transform(X)
31 plt.scatter(Xp[df["anormal"], 0], Xp[df["anormal"], 1],
32             s=30, c="red", label="anormalo")
33 plt.legend(); plt.title("Anomalías de tráfico (PCA 2D)")
34 plt.tight_layout()
35 plt.savefig("anomalías-tráfico.png", dpi=120)

```

Código 14.8: Solución al Ejercicio 14.8.

Epílogo

Has recorrido el camino que va desde el concepto de algoritmo hasta una primera aproximación al aprendizaje automático. Por el medio quedan los tipos de datos y los operadores, las estructuras de control, las funciones y el desarrollo modular, el manejo de ficheros y, ya en el terreno de la ciencia de datos, las bibliotecas `numpy` y `pandas`, la visualización con `matplotlib` y `seaborn`, y los primeros modelos con `scikit-learn`. Cada pieza se apoya en la anterior: sin una base firme en el lenguaje no es posible analizar datos con criterio, y sin entender los datos no se puede aplicar un modelo con sentido.

Conviene recordar que un libro de introducción solo abre puertas. Lo que aquí se ha presentado es suficiente para empezar a resolver problemas reales, pero no agota ninguno de los temas tratados. La forma natural de continuar es profundizar en aquello que el lector vaya necesitando: la documentación oficial de cada biblioteca, proyectos propios que obliguen a buscar lo que no se sabe, y la lectura de código ajeno bien escrito. La programación se aprende programando, y la ciencia de datos, trabajando con datos.

Las técnicas de este libro son independientes del dominio de aplicación. A lo largo del texto la ciberseguridad ha servido de ejemplo recurrente, con la detección de tráfico de red anómalo, la clasificación de programas maliciosos o el análisis de registros de actividad, pero las mismas herramientas se aplican en cualquier disciplina que genere datos. Esa es, quizá, la idea que conviene retener: no se trata de memorizar recetas, sino de adquirir un modo de pensar que permita abordar problemas nuevos.

Gracias por haber llegado hasta aquí. Si el libro te ha resultado útil, o si encuentras erratas y aspectos que mejorar, te ruego que me lo comuniques. El aprendizaje no termina con la última página; en realidad, empieza ahí.

