

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: {  
17     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().split(' ')))
4
5 # Paso 2: Generar todas las combinaciones posibles
6 entre los elementos de las dos listas
7 combinaciones = []
8 for i in lista1:
9     for j in lista2:
10         combinaciones.append((i, j))
11
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: x[0] + x[1], reverse=True)
16
17 # Paso 4: Imprimir las combinaciones ordenadas
18 print("\nCombinaciones ordenadas por la suma de sus
19 elementos en orden descendente:")
20 for combinacion in combinaciones_ordenadas:
21     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 bloques = list(map(int, input("Introduce la lista de
32 bloques: ").strip()[1:-1].split(','))) # Entrada
33 sublista1, sublista2 = obtener_sublistas_equilibradas(
34     bloques) # obtener las sublistas equilibradas
35 print("Las sublistas equilibradas son:", sublista1,
36       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 suma_pares = suma_lista(pares)  
26 suma_impares = suma_lista(impares)  
27  
28 # Paso 4: Mostrar los resultados  
29 print(f"La suma de los números pares es: {suma_pares}")  
30 print(f"La suma de los números impares es: {suma_impares}")  
31 ↵
```

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)

```
1  def contar_caracteres(cadena):
2      diccionario = {}
3      for caracter in cadena:
4          if caracter in diccionario:
5              diccionario[caracter] += 1
6          else:
7              diccionario[caracter] = 1
8      return diccionario
9
10     cadena = input("Introduce una cadena de texto: ")
11     diccionario = contar_caracteres(cadena)
12     print("El diccionario de frecuencias es:", diccionario)
```

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]
4         * (aumento / 100)
5     return diccionario
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 sumar_tuplas(lista_tuplas)
7
8 print("Las sumas de las tuplas son:", sumar_tuplas)
```

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     lista_ordenada = ordenar_tuplas(lista_tuplas)
6     print("La lista ordenada por el primer número es:",
7         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 = sum(tupla)
7         if tupla in suma_dict:
8             suma_dict[tupla] += 1
9         else:
10            suma_dict[tupla] = 1
11
12 # Recibir las listas de tuplas
13 lista1 = eval(input("Introduce la primera lista de
14 tuplas: "))
15 lista2 = eval(input("Introduce la segunda lista de
16 tuplas: "))
17
18 # Obtener las tuplas comunes por la suma de sus
19 elementos
20 comunes = interseccion_por_suma(lista1, lista2)
21
22 # Mostrar el resultado
23 print("Las tuplas comunes por la suma de sus elementos
24 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 conjunto1 es un subconjunto de
10    conjunto2
11    return conjunto1.issubset(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      todos = set()
4      comunes = set()
5      # Contamos cuántas veces aparece cada elemento en
6      los conjuntos
7      for conjunto in conjuntos:
8          for elem in conjunto:
9              if elem not in todos:
10                 todos.add(elem)
11             else:
12                 comunes.add(elem)
13
14         # Los elementos únicos son aquellos que están en
15         todos, pero no en comunes
16         return todos - comunes
17
18     # Recibir la entrada de los conjuntos
19     conjuntos = eval(input("Introduce la lista de conjuntos
20     : "))
21     # Obtener los elementos únicos
22     elementos_unicos = elementos_unicos(conjuntos)
23     # Mostrar el resultado
24     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(' ')))  
7     conjunto2 = set(map(int, input("Introduce el segundo  
8     conjunto de elementos únicos: ").split(' ')))  
9     suma = suma_unica(conjunto1, conjunto2)  
10    print(f"El conjunto de elementos únicos es: {conjunto_unico}")  
11    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: ").split(' ')))  
13 conjunto2 = set(map(int, input("Introduce el segundo  
14 conjunto: ").split(' ')))  
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     # Calcular el porcentaje de diferencia
7     if len(union) == 0: # Evitar división por cero si
8         los conjuntos están vacíos
9         return 0.0
10    return (len(diferencia_simetrica) / len(union)) *
11    100
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.