Módulo 0:Fechas

1. Desafío: Normalización de API Inconsistente

Escenario: Una API devuelve datos de sensores. A veces, la clave 'readings' contiene un *único diccionario*. Otras veces, una *lista de diccionarios*. Queremos una lista plana de *todas* las lecturas (diccionarios) de *todos* los dispositivos.

```
import json

# Datos de entrada: Una pesadilla de JSON anidado e inconsistente
api_response_string = """
[
  {
    "device_id": "DEV-A-01",
    "location": "Norte",
    "payload": {
      "readings": [
        {"timestamp": 1678886400, "temp_c": 12.0, "status": "OK"},
        {"timestamp": 1678886460, "temp_c": 12.1, "status": "OK"}
      ]
    }
  },
  {
    "device_id": "DEV-B-02",
    "location": "Sur",
    "payload": {
      "readings": {"timestamp": 1678886500, "temp_c": 25.5, "status": "ERROR"}
    }
  },
  {
    "device_id": "DEV-C-03",
    "location": "Norte",
    "payload": {
      "readings": []
    }
  },
  {
    "device_id": "DEV-D-04",
    "location": "Oeste",
    "payload": {
```

```
        "readings": [
          {"timestamp": 1678886600, "temp_c": 19.0, "status": "OK"},
          {"timestamp": 1678886660, "temp_c": null, "status": "MAINT"}
        ]
      }
    }
]
"""

print("--- 1. Desafío: Normalización de API Inconsistente ---")

# --- Solución ---
# 1. Parsear el JSON
data = json.loads(api_response_string)

# 2. Usar una comprehension anidada con lógica de tipos
#    para aplanar la estructura
normalized_readings = [
    # 2c. El 'reading' final que queremos
    reading
    # 2a. Iterar por cada dispositivo en la lista principal
    for device in data
    # 2b. Iterar por la lista interna (que puede ser un dict o una lista)
    #     Usamos una 'list comprehension' ternaria para "normalizar"
    #     la clave 'readings' a *siempre* ser una lista.
    for reading in (
        device['payload']['readings']
        if isinstance(device['payload']['readings'], list)
        else [device['payload']['readings']]
    )
]

print(f"Lecturas normalizadas (planas):\n{json.dumps(normalized_readings, indent=2)}")
```

2. Pipeline de Validación y Partición

Escenario: Tenemos una lista "sucia" de registros de usuarios. Debemos aplicar reglas de validación complejas y *particionar* la lista en dos: 'registros_validos' y 'registros_invalidos'.

```
# Lista de entrada con datos "sucios"
registros_nuevos = [
    {'id': 'usr-001', 'email': 'user@example.com', 'age': 30, 'country': 'ES'},
    {'id': 'usr-002', 'email': 'bad-email.com', 'age': 25, 'country': 'MX'},
```

```
    {'id': 'usr-003', 'email': 'another@web.com', 'age': '17', 'country': 'CO'},
    {'id': 'usr-004', 'email': 'valid@server.org', 'age': 45, 'country': 'ES'},
    {'id': 'usr-005', 'email': 'guest@test.io', 'age': 22, 'country': None},
    {'id': 'invalid', 'email': 'test@test.com', 'age': 50, 'country': 'AR'}
]

# --- Reglas de Validación (definidas como lambdas) ---
# Usamos un diccionario de lambdas para tener un "validador"
# Las lambdas devuelven True si la regla se *rompe* (es inválido)
REGLAS_INVALIDACION = {
    'id_invalido': lambda r: not (isinstance(r.get('id'), str) and len(r.get('id', '')) == 7 and r.get('id',
'').startswith('usr-')),
    'email_invalido': lambda r: not (isinstance(r.get('email'), str) and '@' in r.get('email', '') and '.'
in r.get('email', '').split('@')[-1]),
    'edad_invalida': lambda r: not (isinstance(r.get('age'), int) and r.get('age', 0) >= 18),
    'pais_invalido': lambda r: not (isinstance(r.get('country'), str) and len(r.get('country', '')) == 2)
}

print("\n--- 2. Desafío: Pipeline de Validación y Partición ---")

# --- Solución ---

# 1. Crear una lista de tuplas (registro, [lista_de_errores])
#    Usamos una comprehension anidada
registros_con_errores = [
    (
        registro,
        [nombre_regla for nombre_regla, check in REGLAS_INVALIDACION.items() if
check(registro)]
    )
    for registro in registros_nuevos
]

print("\n--- Registros con sus errores ---")
for reg, err in registros_con_errores:
    print(f"{reg['id']}: {err}")

# 2. Particionar los datos en dos listas (válidos e inválidos)
#    usando comprehensions con filtro
registros_validos = [
    reg for reg, err in registros_con_errores if not err # si la lista de errores está vacía
]
```

```
registros_invalidos = [
    {'registro': reg, 'errores': err}
    for reg, err in registros_con_errores if err # si la lista de errores NO está vacía
]

print(f"\n--- Válidos ({len(registros_validos)}) ---")
print(json.dumps(registros_validos, indent=2))
print(f"\n--- Inválidos ({len(registros_invalidos)}) ---")
print(json.dumps(registros_invalidos, indent=2))
```

3. Correlación de Eventos (START/END)

Escenario: Tenemos una lista de eventos de log desordenados. Cada evento tiene un event_id que agrupa un 'START' y un 'END'. Queremos calcular la duración (en segundos) de cada event_id.

```
# Lista de entrada desordenada
eventos = [
    {'ts': 1678887005, 'id': 'proc-B', 'type': 'START'},
    {'ts': 1678886000, 'id': 'proc-A', 'type': 'START'},
    {'ts': 1678887050, 'id': 'proc-B', 'type': 'END'},
    {'ts': 1678886010, 'id': 'proc-A', 'type': 'END'},
    {'ts': 1678888000, 'id': 'proc-C', 'type': 'START'}, # Proc C nunca termina
    {'ts': 1678889000, 'id': 'proc-D', 'type': 'END'},  # Proc D nunca empieza
]

print("\n--- 3. Desafío: Correlación de Eventos ---")

# --- Solución ---
# 1. Separar 'START' y 'END' en diccionarios para búsqueda rápida (O(1))
#    La clave será el 'id', el valor será el 'ts'
#    Usamos dict comprehensions
starts = {
    e['id']: e['ts'] for e in eventos if e['type'] == 'START'
}
ends = {
    e['id']: e['ts'] for e in eventos if e['type'] == 'END'
}

print(f"Starts: {starts}")
print(f"Ends: {ends}")

# 2. Calcular duraciones usando una comprehension que itera sobre
```

```
#    las claves comunes (intersección de conjuntos de claves)
duraciones = {
    id_proc: ends[id_proc] - starts[id_proc]
    for id_proc in starts.keys() & ends.keys() # Intersección de claves
}

print(f"\nDuraciones calculadas (segundos):\n{duraciones}")

# 3. (Bonus) Encontrar IDs huérfanos
orphaned_starts = starts.keys() - ends.keys()
orphaned_ends = ends.keys() - starts.keys()
print(f"IDs sin 'END' (huérfanos): {orphaned_starts}")
print(f"IDs sin 'START' (huérfanos): {orphaned_ends}")
```

4. "GroupBy" Manual (Agregación de Datos)

Escenario: Tenemos una lista plana de registros de ventas. Queremos agregarla para saber el total de ventas (amount) y el número de transacciones (count) *por cada* product_id. (Esto es lo que pandas.groupby(..).agg(..) hace).

```
ventas = [
    {'product': 'A001', 'amount': 100},
    {'product': 'B002', 'amount': 50},
    {'product': 'A001', 'amount': 120},
    {'product': 'C003', 'amount': 200},
    {'product': 'B002', 'amount': 75},
    {'product': 'A001', 'amount': 90},
]

print("\n--- 4. Desafío: 'GroupBy' Manual ---")

# --- Solución ---

# 1. Obtener los IDs de producto únicos
#    Usamos una set comprehension para eficiencia
product_ids_unicos = {v['product'] for v in ventas}
print(f"Productos únicos: {product_ids_unicos}")

# 2. Construir el diccionario agregado
#    Usamos una dict comprehension que itera sobre los IDs únicos
#    El valor de cada clave es otro diccionario, construido "en línea"
agregado = {
    pid: {
```

```
        'total_amount': sum(v['amount'] for v in ventas if v['product'] == pid),
        'count': sum(1 for v in ventas if v['product'] == pid)
    }
    for pid in product_ids_unicos
}
# NOTA: Esto es O(N*M), (N ventas, M productos).
# Es ineficiente a propósito para demostrar el poder de las comprehensions.
# Una solución O(N) usaría un bucle for y un defaultdict, pero
# esto demuestra la lógica de "pensamiento de conjunto" (set-thinking).

print(f"\nDatos agregados:\n{json.dumps(agregado, indent=2)}")

# --- Solución (Alternativa O(N) con sorted + lambda, más avanzada) ---
# (Esto se acerca más a cómo funciona itertools.groupby)
print("\n--- 4. (Alternativa 'GroupBy' más eficiente, sin itertools) ---")

# 1. Ordenar los datos por la clave de agrupación
ventas_ordenadas = sorted(ventas, key=lambda v: v['product'])

# 2. Agrupar (simulado con un bucle, ya que groupby es un iterador)
#    Esta parte rompe la regla de "solo comprehensions", pero es
#    conceptual... ¿Podemos hacerlo sin un bucle explícito?
#    No fácilmente sin itertools.groupby. El ejemplo O(N*M) de arriba
#    es la respuesta más pura de "Módulo 0".
```