Conferencia 6: Estructuras de datos II - Diccionarios y Conjuntos

Introducción

En sesiones anteriores, hemos trabajado principalmente con estructuras de datos secuenciales como las listas y las tuplas. Estas estructuras son útiles cuando el orden de los elementos es importante y accedemos a ellos por su posición o índice (0, 1, 2, etc.).

Hoy, abordaremos dos problemas:

1. ¿Cómo almacenamos datos donde la posición no importa, sino una "etiqueta" o "nombre" específico?
2. ¿Cómo manejamos colecciones de elementos donde no queremos duplicados?

Para esto, Python nos ofrece los **diccionarios** y los **conjuntos**. Estas estructuras no se basan en el *orden*, sino en la *pertenencia* y la *unicidad*, y son extremadamente eficientes.

Luego, veremos cómo la sintaxis de estas estructuras se expande para crear funciones mucho más flexibles.

Parte 1: Diccionarios (dict)

Un diccionario es una estructura de datos que almacena elementos en pares de **clave-valor** (`key: value`). En lugar de usar un índice numérico, usamos una clave única para acceder a su valor asociado. Son la implementación en Python de las "tablas hash" o "mapas".

1. Creación y Reglas de Claves

Se crean usando llaves `{}`.

```
# Un diccionario vacío
configuracion_vacia = {}

# Un diccionario con datos
alumno = {
    'nombre': 'Ana',
    'id_matricula': 12345,
    'curso_actual': 'Programación I',
    'activa': True
}
```

- `'nombre'` es una clave, `'Ana'` es su valor.
- Las claves deben ser únicas. Si se repite una clave, prevalece el último valor.

¿Qué puede ser una Clave? Las claves *deben* ser de un tipo de dato **inmutable**.

- **Sí pueden ser claves:** Strings (`'hola'`), números (`123`, `4.5`), `True`/`False` y tuplas `(1, 2)`.
- **No pueden ser claves:** Listas (`[1, 2]`), otros diccionarios (`{...}`) o conjuntos (`{1, 2}`).

Intentar usar una lista como clave generará un `TypeError`.

2. Operaciones Fundamentales

Acceder a un Valor (Lectura) Se accede usando corchetes `[]` con la clave.

```
print(alumno['nombre'])       # Imprime: 'Ana'
```

- **Error Común:** Si intentamos acceder a una clave que no existe, el programa lanzará un `KeyError`. `# print(alumno['apellido'])  # <-- Esto genera un KeyError`

Acceso Seguro (Lectura) Para evitar errores, usamos el método `.get()`. Acepta un valor por defecto (opcional) si la clave no se encuentra.

```
# Si no encuentra 'apellido', devuelve None (por defecto)
apellido = alumno.get('apellido')
print(apellido)  # Imprime: None

# Si no encuentra 'universidad', devuelve 'No especificada'
universidad = alumno.get('universidad', 'No especificada')
print(universidad) # Imprime: 'No especificada'
```

Añadir y Modificar (Creación/Actualización) La sintaxis `[]` sirve para crear o actualizar. Si la clave existe, se actualiza. Si no existe, se crea.

```
# Modificar un valor existente
alumno['curso_actual'] = 'Programación II'

# Añadir un nuevo par clave-valor
alumno['creditos'] = 24
```

Añadir/Fusionar Múltiples Pares (Actualización) Para fusionar otro diccionario o añadir múltiples pares, usamos `.update()`.

```
# Añadir múltiples valores
alumno.update({'campus': 'Central', 'promedio': 9.5})
print(alumno)

# Fusionar otro diccionario (sobrescribe claves si ya existen)
config_adicional = {'id_matricula': 55555, 'beca': True}
alumno.update(config_adicional)
# 'id_matricula' ahora es 55555 y se ha añadido 'beca': True
```

Eliminar un Par Clave-Valor Existen dos formas principales:

1. `del`: Rápido, pero da `KeyError` si la clave no existe. `python  # del alumno['activa'] # Elimina el par, pero falla si no existe`
2. `.pop()` (Recomendado): Es más seguro. Elimina la clave, **devuelve el valor** y permite un valor por defecto para evitar errores.

   ```
   # Elimina 'creditos' y guarda el valor 24 en la variable
   creditos_eliminados = alumno.pop('creditos')

   # Intenta eliminar 'activa'. Como no existe, no da error
   # y devuelve el valor por defecto None.
   valor_activo = alumno.pop('activa', None)
   print(valor_activo) # Imprime: None
   ```

Comprobar la Existencia de una Clave Usamos el operador `in`. Esta operación es **extremadamente rápida** en diccionarios.

```
print('nombre' in alumno)       # Imprime: True
print('apellido' in alumno)     # Imprime: False
```

- **Importante:** El operador `in` comprueba la existencia de **claves**, no de valores.

3. Iteración sobre Diccionarios

Esta es la operación más común. Hay tres formas de recorrerlos:

```
# 1. Iterar sobre las claves (por defecto)
print("--- Claves ---")
for clave in alumno:
    print(clave)
    # Podemos usar la clave para obtener el valor
    # print(f"{clave}: {alumno[clave]}")

# 2. Iterar sobre los valores
print("--- Valores ---")
for valor in alumno.values():
    print(valor)

# 3. Iterar sobre pares clave-valor (la más útil)
print("--- Clave y Valor ---")
for clave, valor in alumno.items():
    print(f"[{clave}] -> {valor}")
```

Parte 2: Conjuntos (set)

Un conjunto es una colección de elementos **sin orden** y **sin elementos duplicados**.

Son útiles principalmente para:

1. Eliminar duplicados de una lista de forma rápida.
2. Comprobar pertenencia (usando `in`) muy rápidamente.
3. Realizar operaciones matemáticas de conjuntos (unión, intersección, etc.).

1. Creación de Conjuntos

Se pueden crear con `{}` (si no están vacíos) o con la función `set()`.

```
# Python elimina automáticamente los duplicados
numeros_primos = {2, 3, 5, 7, 11, 2, 3}
print(numeros_primos)  # Imprime: {2, 3, 5, 7, 11} (El orden puede variar)

# Crear un conjunto a partir de una lista (elimina duplicados)
lista_con_duplicados = [1, 1, 2, 2, 3, 4, 1]
conjunto_numeros = set(lista_con_duplicados)
print(conjunto_numeros)  # Imprime: {1, 2, 3, 4}
```

Importante: Para crear un conjunto **vacío**, deben usar `set()`. Usar `{}` crea un diccionario vacío.

```
mi_conjunto_vacio = set()
mi_diccionario_vacio = {}
```

2. Operaciones Básicas (Añadir, Quitar, Comprobar)

```
mi_conjunto = {'rojo', 'verde', 'azul'}

# Añadir un elemento
mi_conjunto.add('amarillo') # {'rojo', 'verde', 'azul', 'amarillo'}

# Si añadimos un elemento que ya existe, el conjunto no cambia
mi_conjunto.add('rojo') # Sigue igual

# Eliminar un elemento (da KeyError si no existe)
mi_conjunto.remove('verde')

# Eliminar un elemento (NO da error si no existe)
mi_conjunto.discard('negro') # No hace nada, no hay error

# Comprobar Membresía (muy rápida)
print('rojo' in mi_conjunto)     # Imprime: True
```

3. Operaciones Matemáticas de Conjuntos

Aquí radica el verdadero poder de los conjuntos.

```
set_A = {1, 2, 3, 4}
set_B = {3, 4, 5, 6}
```

Usando Operadores (Solo entre conjuntos)

```
# Unión (|): Todos los elementos de A y B, sin duplicados
union = set_A | set_B         # {1, 2, 3, 4, 5, 6}

# Intersección (&): Solo los elementos que están en AMBOS
interseccion = set_A & set_B  # {3, 4}

# Diferencia (-): Elementos en A, pero NO en B
diferencia = set_A - set_B    # {1, 2}

# Diferencia Simétrica (^): Elementos que están en A o B, pero NO en ambos
simetrica = set_A ^ set_B     # {1, 2, 5, 6}
```

Usando Métodos (Más flexibles) Los métodos (ej. `.union()`) pueden operar con *cualquier iterable* (como una lista), mientras que los operadores (ej. `|`) *requieren* que ambos operandos sean conjuntos.

```
lista_C = [5, 6, 7, 8]

# Unión con una lista (funciona)
union_metodo = set_A.union(lista_C)
```

```
print(union_metodo) # {1, 2, 3, 4, 5, 6, 7, 8}

# Intersección con una lista (funciona)
inter_metodo = set_B.intersection(lista_C)
print(inter_metodo) # {5, 6}

# Esto daría un TypeError:
# union_op = set_A | lista_C # ERROR
```

4. Comprobación de Subconjuntos

Podemos verificar si un conjunto está contenido en otro.

```
set_pequeno = {1, 2}
set_grande = {1, 2, 3, 4}

# ¿Es 'pequeno' un subconjunto de 'grande'?
print(set_pequeno.issubset(set_grande)) # True
print(set_pequeno <= set_grande)        # True (sintaxis alternativa)

# ¿Es 'grande' un superconjunto de 'pequeno'?
print(set_grande.issuperset(set_pequeno)) # True
print(set_grande >= set_pequeno)          # True
```

5. Conjuntos Inmutables (frozenset)

Un `set` normal es mutable (puedes `.add()` o `.remove()`). Esto significa que **no puede ser usado como clave de diccionario**.

Si necesitamos usar un conjunto como clave, debemos usar un `frozenset`, que es su versión inmutable.

```
conjunto_mutable = {1, 2}
conjunto_inmutable = frozenset([1, 2])

# Esto es válido:
mi_dict = {conjunto_inmutable: 'Valor'}

# Esto da TypeError:
# mi_dict_error = {conjunto_mutable: 'Valor'}
```

Parte 3: Empaquetado de Argumentos (`*args` y `**kwargs`)

Ahora conectaremos estas estructuras con la definición de funciones.

1. `*args` (Argumentos Posicionales)

`*args` en la definición de una función permite pasar un número variable de argumentos *posicionales*. Python los agrupa automáticamente en una **tupla**.

```python
def funcion_con_args(parametro_fijo, *args):
    print(f"Fijo: {parametro_fijo}")
    print(f"Args (Tupla): {args}")

funcion_con_args('Hola', 1, 2, 3)
# Fijo: Hola
# Args (Tupla): (1, 2, 3)
```

2. `**kwargs` (Argumentos de Palabra Clave)

`**kwargs` (Keywords Arguments) permite pasar un número variable de argumentos **nombrados**. Python los agrupa automáticamente en un **diccionario**.

```python
def configurar_sistema(**opciones):
    # 'opciones' es un diccionario, podemos usar .get()
    modo = opciones.get('modo', 'produccion')
    version = opciones.get('version', 1.0)
    debug = opciones.get('debug', False)

    print(f"Modo: {modo}, Versión: {version}, Debug: {debug}")

# Llamada completa
configurar_sistema(modo='desarrollo', debug=True)
# Modo: desarrollo, Versión: 1.0, Debug: True

# Llamada parcial
configurar_sistema(version=2.5)
# Modo: produccion, Versión: 2.5, Debug: False
```

3. Orden de Parámetros y Keyword-Only Args

El orden estricto al definir una función es:

1. Argumentos estándar.
2. `*args`
3. Argumentos de solo-palabra-clave (Keyword-Only).
4. `**kwargs`

Los **argumentos de solo-palabra-clave** son argumentos que *deben* ser pasados por nombre (no por posición). Se definen después de `*args` o de un asterisco `*` solo.

```python
# 'nombre' es estándar
# 'permisos' es keyword-only
# 'opciones' captura el resto (kwargs)
def crear_usuario(nombre, *, permisos, **opciones):
    print(f"Usuario: {nombre}")
    print(f"Permisos: {permisos}")
    print(f"Opciones: {opciones}")

# Correcto
crear_usuario('Ana', permisos='admin', activa=True, email='ana@mail.com')
# Usuario: Ana
# Permisos: admin
```

```
# Opciones: {'activa': True, 'email': 'ana@mail.com'}

# Incorrecto (TypeError)
# 'permisos' no puede pasarse posicionalmente
# crear_usuario('Ana', 'admin')
```

Parte 4: Sintaxis Generalizada * y ** (Desempaquetado)

Hemos visto * y ** al *definir* funciones (empaquetar). Ahora los veremos al *llamar* funciones y al *crear literales* (desempaquetar).

1. Desempaquetado en Llamadas a Funciones

Con * (Listas/Tuplas) Si una función espera argumentos separados, y los tenemos en una lista, * los "desempaqueta".

```
def suma(a, b, c):
    return a + b + c

numeros = [10, 20, 30]

# Desempaquetamos la lista en argumentos
resultado = suma(*numeros) # Equivale a: suma(10, 20, 30)
print(resultado) # 60
```

Con ** (Diccionarios) Si una función espera argumentos nombrados, y los tenemos en un diccionario, ** los desempaqueta.

```
def crear_usuario(nombre, email, es_admin=False):
    print(f"Creando usuario: {nombre}, Email: {email}, Admin: {es_admin}")

datos_admin = {
    'nombre': 'Sara',
    'email': 'sara.admin@email.com',
    'es_admin': True
}

# Desempaquetamos el diccionario en argumentos nombrados
crear_usuario(**datos_admin)
# Equivale a:
# crear_usuario(nombre='Sara', email='sara.admin@email.com', es_admin=True)
```

2. Desempaquetado para Fusión de Colecciones (En literales)

Desde Python 3.5+, podemos usar * y ** directamente al crear listas y diccionarios para fusionarlos.

Fusión de Listas

```
lista_A = [1, 2]
lista_B = [10, 11]
lista_total = [*lista_A, 5, *lista_B]
print(lista_total) # [1, 2, 5, 10, 11]
```

Fusión de Diccionarios Esto es extremadamente útil para manejar configuraciones. Si una clave se repite, **el último valor "gana"**.

```
config_default = {'modo': 'produccion', 'debug': False}
config_usuario = {'modo': 'desarrollo', 'version': 2.0}

# config_usuario sobrescribe las claves de config_default
config_final = {**config_default, **config_usuario}

print(config_final)
# Imprime: {'modo': 'desarrollo', 'debug': False, 'version': 2.0}
```

Resumen

1. **Diccionarios (`dict`)**: Colecciones `clave:valor`. Se accede por clave `d['k']` (da error si no existe) o `d.get('k', def)` (seguro). Se itera con `.keys()`, `.values()` e `.items()`. Se fusionan con `.update()` o `{**d1, **d2}`. Se elimina de forma segura con `.pop('k', def)`.
2. **Conjuntos (`set`)**: Elementos **únicos** y sin orden. Útiles para eliminar duplicados y operaciones matemáticas.
   - **Operadores** (`|`, `&`): Requieren que ambos sean `set`.
   - **Métodos** (`.union()`, `.intersection()`): Aceptan cualquier iterable.
   - `frozenset`: Versión inmutable que SÍ puede ser clave de diccionario.
3. **`*args` y `**kwargs`** (En definición): Empaquetan argumentos variables en una `tupla` y un `diccionario`, respectivamente.
4. **`*` y `**`** (En llamada/literales): Desempaquetan iterables y diccionarios, ya sea para pasarlos a funciones (`func(*lista)`) o para fusionar colecciones (`lista_total = [*l1, *l2]`).