Los Diccionarios en Python

1. Introducción a los Diccionarios

Los diccionarios en Python son estructuras de datos que almacenan pares clave-valor. Son mutables, no ordenados (hasta Python 3.6, desde 3.7+ mantienen orden de inserción) y muy eficientes para búsquedas. Se definen usando llaves {} y los elementos se separan por comas.

Características principales:

- **Mutables**: Se pueden modificar después de su creación
- **Acceso por clave**: No se accede por índice, sino por clave
- **Claves únicas**: No puede haber claves duplicadas
- **Claves inmutables**: Las claves deben ser de tipos inmutables (str, int, tuple)
- **Dinámicos**: Pueden crecer y decrecer dinámicamente
- **Eficientes**: Búsqueda O(1) en promedio usando tabla hash

2. Creación y Acceso a Diccionarios

Creación de diccionarios:

```
# Diccionario vacío
dict_vacio = {}
dict_vacio2 = dict()

# Diccionario con elementos
persona = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}

# Diccionario desde constructor
dict_constructor = dict(nombre="Carlos", edad=30)

# Diccionario desde lista de tuplas
pares = [("a", 1), ("b", 2), ("c", 3)]
dict_tuplas = dict(pares)

# Diccionario con fromkeys
claves = ["x", "y", "z"]
dict_fromkeys = dict.fromkeys(claves, 0)
# {"x": 0, "y": 0, "z": 0}
```

Acceso a elementos:

```
alumno = {
    "nombre": "Juan",
    "edad": 20,
    "notas": [8, 9, 7],
    "activo": True
}

# Acceso con corchetes
print(alumno["nombre"])  # Juan
print(alumno["notas"])   # [8, 9, 7]

# Acceso con get() - más seguro
print(alumno.get("edad"))        # 20
print(alumno.get("email"))       # None
print(alumno.get("email", "N/A")) # N/A

# Modificar valores
alumno["edad"] = 21

# Añadir nuevas claves
alumno["email"] = "juan@email.com"
```

Si pones algo que no está, te da error

Si pones algo que no está, da None

Not available, lo añade al diccionario.

Ejercicios - Creación y Acceso:

Ejercicio 1 (Básico): Crear un diccionario con información personal y acceder a sus elementos.

```
# Crear diccionario de información personal
def crear_perfil():
    perfil = {
        "nombre": input("Nombre: "),
        "edad": int(input("Edad: ")),
        "hobbies": input("Hobbies (separados por coma): ").split(","),
        "contacto": {
            "email": input("Email: "),
            "telefono": input("Teléfono: ")
        }
    }
    return perfil
```

```
# Mostrar información
def mostrar_perfil(perfil):
    print(f"\nPerfil de {perfil['nombre']}")
    print(f"Edad: {perfil['edad']} años")
    print(f"Hobbies: {', '.join(perfil['hobbies'])}")
    print(f"Email: {perfil['contacto']['email']}")
```

Ejercicio 2 (Intermedio): Implementar un contador de frecuencias de palabras.

```
def contador_palabras(texto):
    """Cuenta la frecuencia de cada palabra en un texto"""
    # Limpiar y dividir el texto
    palabras = texto.lower().replace(',', '').replace('.', '').split()

    # Contar frecuencias
    frecuencias = {}
    for palabra in palabras:
        if palabra in frecuencias:
            frecuencias[palabra] += 1
        else:
            frecuencias[palabra] = 1

    # Alternativa más pythónica con get()
    # for palabra in palabras:
    #     frecuencias[palabra] = frecuencias.get(palabra, 0) + 1

    return frecuencias

# Prueba
texto = "el gato subió al tejado el tejado era rojo"
frecuencias = contador_palabras(texto)
print(frecuencias)  # {"el": 2, "gato": 1, ...}
```

Ejercicio 3 (Avanzado): Crear un sistema de caché con expiración de tiempo.

```
import time
from datetime import datetime, timedelta

class CacheConExpiracion:
    """Sistema de caché con tiempo de expiración"""
```

```
        def __init__(self, ttl_segundos=60):
            self.cache = {}
            self.ttl = ttl_segundos

        def set(self, clave, valor, ttl_custom=None):
            """Almacena un valor con tiempo de expiración"""
            ttl = ttl_custom if ttl_custom else self.ttl
            expiracion = datetime.now() + timedelta(seconds=ttl)
            self.cache[clave] = {
                'valor': valor,
                'expiracion': expiracion
            }

        def get(self, clave):
            """Obtiene un valor si no ha expirado"""
            if clave not in self.cache:
                return None

            item = self.cache[clave]
            if datetime.now() > item['expiracion']:
                del self.cache[clave]
                return None

            return item['valor']
```

3. Métodos Principales de los Diccionarios

Métodos de acceso y obtención:

```
diccionario = {"a": 1, "b": 2, "c": 3}

# keys() - Retorna las claves
claves = diccionario.keys()  # dict_keys(["a", "b", "c"])
print(list(claves))  # ['a', 'b', 'c']

# values() - Retorna los valores
valores = diccionario.values()  # dict_values([1, 2, 3])
print(list(valores))  # [1, 2, 3]

# items() - Retorna pares clave-valor
items = diccionario.items()
print(list(items))  # [('a', 1), ('b', 2), ('c', 3)]

# get() - Obtiene valor con valor por defecto
valor = diccionario.get("a", 0)  # 1
valor = diccionario.get("z", 0)  # 0 (no existe)

# setdefault() - Obtiene valor o lo crea si no existe
valor = diccionario.setdefault("d", 4)  # 4 (lo crea)
print(diccionario)  # {"a": 1, "b": 2, "c": 3, "d": 4}
```

Devuelve los formatos clava valor en formato tupla

Métodos de modificación:

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

# update() - Actualiza con otro diccionario
dict1.update(dict2)
print(dict1)  # {"a": 1, "b": 3, "c": 4}

# update() con kwargs
dict1.update(d=5, e=6)
print(dict1)  # {"a": 1, "b": 3, "c": 4, "d": 5, "e": 6}

# pop() - Elimina y retorna valor
```

```
valor = dict1.pop("a")  # 1
valor = dict1.pop("z", "No existe")  # "No existe"

# popitem() - Elimina y retorna último par
ultimo = dict1.popitem()  # ('e', 6)

# clear() - Vacía el diccionario
dict1.clear()
print(dict1)  # {}
```

4. Operaciones con Diccionarios

Operaciones básicas:

```
# Iteración sobre diccionarios
diccionario = {"a": 1, "b": 2, "c": 3}

# Iterar sobre claves
for clave in diccionario:
    print(clave)  # a, b, c

# Iterar sobre valores
for valor in diccionario.values():
    print(valor)  # 1, 2, 3

# Iterar sobre pares clave-valor
for clave, valor in diccionario.items():
    print(f"{clave}: {valor}")

# Operador de pertenencia
"a" in diccionario  # True (busca en claves)
1 in diccionario.values()  # True

# Longitud
len(diccionario)  # 3
```

5. Diccionarios por Comprensión

Los diccionarios por comprensión ofrecen una forma concisa y pythónica de crear diccionarios.

Sintaxis básica:

```
# Sintaxis: {clave: valor for elemento in iterable if condición}

# Básica
cuadrados = {x: x**2 for x in range(5)}
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Con condición
```

```python
pares = {x: x**2 for x in range(10) if x % 2 == 0}
# {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

# Desde dos listas
claves = ["a", "b", "c"]
valores = [1, 2, 3]
diccionario = {k: v for k, v in zip(claves, valores)}
# {"a": 1, "b": 2, "c": 3}

# Transformar diccionario existente
original = {"a": 1, "b": 2, "c": 3}
doble = {k: v*2 for k, v in original.items()}
# {"a": 2, "b": 4, "c": 6}
```

6. Ejercicios Integradores

Ejercicio Final: Sistema de Gestión de Inventario

Resumen

Los diccionarios son estructuras de datos fundamentales en Python que permiten almacenar y acceder eficientemente a datos mediante pares clave-valor. Su implementación basada en tablas hash proporciona acceso O(1) en promedio, haciéndolos ideales para búsquedas rápidas, caché, índices y muchas otras aplicaciones.

Conceptos clave a recordar:

- Los diccionarios son mutables pero las claves deben ser inmutables
- Desde Python 3.7+, los diccionarios mantienen el orden de inserción
- El método get() es más seguro que el acceso directo con corchetes
- Los métodos items(), keys() y values() facilitan la iteración
- Las comprensiones de diccionarios ofrecen sintaxis concisa y eficiente
- Los diccionarios son ideales para implementar estructuras de datos complejas

Mejores prácticas:

- Usar get() con valor por defecto para evitar KeyError
- Preferir setdefault() para inicializar valores
- Utilizar defaultdict para estructuras anidadas
- Considerar Counter para contar frecuencias
- Aprovechar las comprensiones para transformaciones
- Documentar el esquema esperado en diccionarios complejos

El dominio de los diccionarios es esencial para escribir código Python eficiente y elegante. Su versatilidad los convierte en una herramienta indispensable para resolver problemas de programación del mundo real, desde simples contadores hasta complejos sistemas de gestión de datos.