Estructuras de Datos Avanzadas en Python para Aprendizaje por Refuerzo

Dr. Darío Ezequiel Díaz

Marzo 2025

Contents

1 Listas en Python

Las listas constituyen una estructura fundamental en Python, caracterizadas por su versatilidad y flexibilidad. A diferencia de otros lenguajes de programación donde los arrays tienen limitaciones significativas, las listas en Python destacan mediante características distintivas que facilitan considerablemente la manipulación de datos.

1.1 Características principales

- **Colecciones ordenadas**: Mantienen el orden de inserción de los elementos
- **Heterogeneidad**: Pueden almacenar diferentes tipos de datos simultáneamente
- **Mutabilidad**: Sus elementos pueden modificarse después de la creación
- **Indexación**: Acceso directo por posición (desde 0)
- **Flexibilidad**: Pueden crecer o disminuir dinámicamente

1.2 Creación de listas

```
# Lista con números enteros
numeros = [1, 2, 3, 4, 5]

# Lista con diferentes tipos de datos (heterogénea)
datos_varios = [42, "hola", 3.1416, False, [10, 20]] # Contiene hasta
    otra lista

# Lista vacía
lista_vacia = []
```

1.3 Acceso a elementos

La indexación funciona mediante un esquema intuitivo donde cada elemento ocupa una posición numérica. Python ofrece dos formas complementarias de acceder a estos elementos:

```
colores = ["rojo", "verde", "azul", "amarillo"]

# Índice positivo: comienza desde 0 hasta n-1 (siendo n la cantidad de
    elementos)
primer_color = colores[0] # "rojo"

# Índice negativo: comienza desde -1 (último elemento) hacia atrás
ultimo_color = colores[-1] # "amarillo"
```

```
# Equivalente con índice positivo
ultimo_color_con_indice_positivo = colores[3] # "amarillo"
```

1.4 Modificación de listas

La naturaleza mutable de las listas permite diversas operaciones que transforman su contenido:

```
numeros = [10, 20, 30, 40, 50]

# Modificar un elemento existente
numeros[2] = 35 # Ahora numeros es [10, 20, 35, 40, 50]

# Añadir elementos
lista = [1, 2, 3]
lista.append("hola") # Ahora lista es [1, 2, 3, "hola"]

# Insertar en posición específica
lista.insert(1, 1.5) # Ahora lista es [1, 1.5, 2, 3, "hola"]

# Eliminar elementos por valor
lista.remove(1.5) # Ahora lista es [1, 2, 3, "hola"]

# Eliminar por índice
del lista[0] # Ahora lista es [2, 3, "hola"]

# Extraer elemento con retorno
elemento = lista.pop(1) # Extrae y devuelve el elemento en posición 1
# elemento = "hola", lista queda como [2, 3]
```

1.5 Métodos avanzados

Las listas ofrecen funcionalidades sofisticadas para operaciones comunes:

```
numeros = [42, 13, 7, 24, 36, 5]

# Ordenamiento in-place (modifica la lista original)
numeros.sort() # Ahora numeros es [5, 7, 13, 24, 36, 42]

# Ordenamiento inverso
numeros.sort(reverse=True) # Ahora numeros es [42, 36, 24, 13, 7, 5]

# Ordenamiento de listas heterogéneas
lista_mixta = [42, "cadena", True, 3.14159]
# Para ordenar elementos de diferentes tipos se necesita una "clave" de
    comparación
lista_mixta.sort(key=str) # Convierte cada elemento a string para
    comparar
```

```
# Ordenamiento sin modificar la original (devuelve una nueva lista)
lista_ordenada = sorted(lista_mixta, key=str)

# Inversión
lista = [1, 2, 3, 4, 5]
lista.reverse() # Ahora lista es [5, 4, 3, 2, 1]
```

2 Tuplas en Python

Las tuplas representan una estructura de datos inmutable, proporcionando un mecanismo para almacenar colecciones ordenadas que no deben cambiar durante la ejecución del programa.

2.1 Características principales

- **Inmutabilidad**: Una vez creadas, no pueden modificarse
- **Ordenadas**: Mantienen el orden de inserción
- **Eficiencia**: Más rápidas y consumen menos memoria que las listas
- **Hashables**: Pueden usarse como claves en diccionarios (a diferencia de las listas)

2.2 Creación de tuplas

```
# Creación con paréntesis
coordenadas = (10, 20)
persona = ("Ana", 28, "Ingeniera")

# Tupla con un solo elemento (requiere coma)
singleton = (42,)

# Creación implícita (sin paréntesis)
tupla_implicita = 1, 2, 3 # Equivalente a (1, 2, 3)
```

2.3 Inmutabilidad

La característica distintiva de las tuplas radica en su inmutabilidad:

```
tupla = (10, 20, 30)
# tupla[0] = 99 # Esto produciría un TypeError
```

Aunque no podemos modificar una tupla directamente, podemos crear nuevas tuplas basadas en existentes:

```
tupla_original = (1, 2, 3)
nueva_tupla = tupla_original + (4, 5) # (1, 2, 3, 4, 5)
subtupla = tupla_original[0:2] # (1, 2)
tupla_repetida = tupla_original * 2 # (1, 2, 3, 1, 2, 3)
```

2.4 Desempaquetado de tuplas

Una operación particularmente útil con tuplas es el desempaquetado, que permite asignar los elementos individuales a variables:

```
tupla = (100, 200, 300)
x, y, z = tupla # x=100, y=200, z=300

# Aplicación práctica: intercambio de variables
a = 10
b = 20
a, b = b, a # Ahora a=20, b=10
```

3 Diccionarios en Python

Los diccionarios constituyen estructuras asociativas que establecen mapeos entre claves y valores, facilitando búsquedas extremadamente eficientes.

3.1 Características principales

- **Mapeo clave-valor**: Cada elemento asocia una clave con un valor
- **Claves únicas**: No permite duplicados en las claves
- **Claves inmutables**: Las claves deben ser objetos hashables (strings, números, tuplas)
- **Valores flexibles**: Los valores pueden ser cualquier tipo de dato
- **Acceso O(1)**: Búsqueda por clave extremadamente eficiente

3.2 Creación de diccionarios

```
# Creación mediante llaves
estudiante = {
   "nombre": "Ana García",
   "edad": 22,
   "carrera": "Ingeniería Informática",
   "promedio": 8.7
}

# Diccionario vacío
diccionario_vacio = {}
```

```
# Creación mediante la función dict()
coordenadas = dict(x=10, y=20, z=30)

# Desde secuencia de pares clave-valor
items = [("a", 1), ("b", 2), ("c", 3)]
letras_numeros = dict(items)
```

3.3 Operaciones fundamentales

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

# Acceso por clave
valor_b = diccionario["b"] # 2

# Acceso seguro (para claves que quizás no existan)
valor_d = diccionario.get("d", 0) # Devuelve 0 si "d" no existe

# Insertar o modificar
diccionario["d"] = 4 # Añade nuevo par
diccionario["a"] = 10 # Modifica valor existente

# Eliminar pares
del diccionario["b"] # Elimina el par con clave "b"

# Vaciar diccionario
diccionario.clear() # Elimina todos los pares
```

3.4 Verificación y recuperación

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

# Verificar existencia de clave
existe_b = "b" in diccionario # True
existe_x = "x" in diccionario # False

# Obtener colecciones de claves, valores o pares
claves = diccionario.keys() # dict_keys(['a', 'b', 'c'])
valores = diccionario.values() # dict_values([1, 2, 3])
items = diccionario.items() # dict_items([('a', 1), ('b', 2), ('c', 3)])
```

3.5 Diccionarios anidados

Los diccionarios pueden contener estructuras más complejas como valores, incluyendo otros diccionarios:

```
empleados = {
   "E001": {
       "nombre": "Ana García",
       "departamento": "Ingeniería",
       "salario": 50000
   },
   "E002": {
       "nombre": "Carlos López",
       "departamento": "Marketing",
       "salario": 45000
   }
}

# Acceso a datos anidados
nombre_e001 = empleados["E001"]["nombre"] # "Ana García"

# Modificación de datos anidados
empleados["E002"]["salario"] = 48000
```

4 Sets (Conjuntos) en Python

Los conjuntos implementan el concepto matemático de conjunto: colecciones desordenadas de elementos únicos, ideales para operaciones de pertenencia y eliminación de duplicados.

4.1 Características principales

- **Elementos únicos**: No permite duplicados
- **Desordenados**: No mantienen orden de inserción
- **Elementos hashables**: Solo pueden contener objetos inmutables
- **Mutabilidad**: Se pueden modificar (añadir/eliminar elementos)
- **Eficiencia**: Comprobación de pertenencia muy rápida

4.2 Creación de conjuntos

```
# Creación mediante llaves
primos = {2, 3, 5, 7, 11, 13}
vocales = {"a", "e", "i", "o", "u"}

# Conjunto vacío (no se puede usar {})
conjunto_vacio = set()

# Creación a partir de iterables
caracteres = set("Hola") # {'H', 'o', 'l', 'a'}
```

```
# Los duplicados se eliminan automáticamente
numeros = {1, 2, 2, 3, 3, 3} # {1, 2, 3}
```

4.3 Modificación de conjuntos

```
conjunto = {1, 2, 3}

# Añadir elementos
conjunto.add(4) # {1, 2, 3, 4}

# Añadir múltiples elementos
conjunto.update([4, 5, 6]) # {1, 2, 3, 4, 5, 6}

# Eliminar elementos
conjunto.remove(3) # Genera error si no existe
conjunto.discard(3) # No genera error si no existe
```

4.4 Operaciones entre conjuntos

Los conjuntos soportan operaciones matemáticas:

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# Unión (A ∪B)
union = A | B # {1, 2, 3, 4, 5, 6, 7, 8}
union_alt = A.union(B) # Método alternativo

# Intersección (A ∩B)
interseccion = A & B # {4, 5}
interseccion_alt = A.intersection(B)

# Diferencia (A - B)
diferencia = A - B # {1, 2, 3}
diferencia_alt = A.difference(B)

# Diferencia simétrica (A △B)
dif_simetrica = A ^ B # {1, 2, 3, 6, 7, 8}
dif_simetrica_alt = A.symmetric_difference(B)
```

4.5 Conjuntos inmutables (frozenset)

Para situaciones donde necesitamos inmutabilidad:

```
# Creación de un conjunto inmutable
vocales_inmutables = frozenset({"a", "e", "i", "o", "u"})
```

```
# Uso como clave en diccionarios
grupos_fonéticos = {
   frozenset({"b", "p", "m"}): "Bilabiales",
   frozenset({"d", "t", "n"}): "Dentales"
}
```

5 Comparación de Estructuras de Datos

Estructura	Mutabilidad	Ordenamiento	Duplicados	Indexación	Uso Principal
Lista	Mutable	Ordenada	Permitidos	Por índice	Secuencias modificables
Tupla	Inmutable	Ordenada	Permitidos	Por índice	Datos inmutables, claves compuestas
Diccionario	Mutable	No ordenado*	Claves únicas	Por clave	Mapeos, búsquedas rápidas
Set	Mutable	No ordenado	No permitidos	No	Eliminar duplicados, pertenencia
Frozenset	Inmutable	No ordenado	No permitidos	No	Sets inmutables, claves de diccionario

Table 1: Comparación de estructuras de datos en Python

** A partir de Python 3.7, los diccionarios mantienen orden de inserción como una característica de implementación, formalizada en Python 3.8.*

6 Aplicaciones en Aprendizaje por Refuerzo

Estas estructuras resultan fundamentales para implementar algoritmos de Aprendizaje por Refuerzo:

- **Listas**: Almacenamiento de secuencias de estados, acciones y recompensas durante episodios.
- **Tuplas**: Representación inmutable de estados, transiciones (estado, acción, recompensa, siguiente_estado).
- **Diccionarios**:
  - Implementación de tablas Q (mapeo estado-acción a valores)
  - Almacenamiento de configuraciones de entornos
  - Caching de resultados para optimización
- **Sets**:

- Seguimiento de estados visitados
- Implementación de espacios de acciones discretos
- Eliminación eficiente de duplicados en experiencias

7 Ejemplo práctico para Aprendizaje por Refuerzo

A continuación, un ejemplo de cómo estas estructuras se utilizan juntas en un algoritmo Q-learning simplificado:

```
import numpy as np
import random

# Diccionario para almacenar valores Q
q_table = {} # Formato: {(estado): [valor_accion1, valor_accion2, ...]}

# Parámetros de aprendizaje
alpha = 0.1 # Tasa de aprendizaje
gamma = 0.9 # Factor de descuento
epsilon = 0.1 # Exploración/explotación

# Posibles acciones representadas como conjunto
acciones = {0, 1, 2, 3} # Norte, Este, Sur, Oeste

# Episodios de entrenamiento
for episodio in range(1000):
   # Inicializar estado
   estado_actual = (0, 0) # Representado como tupla (x, y)
   terminado = False

   # Lista para almacenar la trayectoria del episodio
   trayectoria = []

   while not terminado:
       # Si el estado no está en la tabla Q, añadirlo
       if estado_actual not in q_table:
           q_table[estado_actual] = [0, 0, 0, 0]

       # Política epsilon-greedy para selección de acción
       if random.random() < epsilon:
           accion = random.choice(list(acciones)) # Exploración
       else:
           accion = np.argmax(q_table[estado_actual]) # Explotación

       # Simular transición (simplificado)
       if accion == 0: # Norte
           siguiente_estado = (estado_actual[0], estado_actual[1] + 1)
       elif accion == 1: # Este
           siguiente_estado = (estado_actual[0] + 1, estado_actual[1])
```

```
            elif accion == 2: # Sur
                siguiente_estado = (estado_actual[0], estado_actual[1] - 1)
            else: # Oeste
                siguiente_estado = (estado_actual[0] - 1, estado_actual[1])

            # Determinar recompensa (simplificado)
            if siguiente_estado == (5, 5):
                recompensa = 100
                terminado = True
            elif siguiente_estado[0] < 0 or siguiente_estado[1] < 0:
                recompensa = -10
                siguiente_estado = estado_actual # Rebote en límites
            else:
                recompensa = -1 # Pequeña penalización por cada paso

            # Almacenar transición en la trayectoria
            trayectoria.append((estado_actual, accion, recompensa,
                siguiente_estado))

            # Actualizar tabla Q
            if siguiente_estado not in q_table:
                q_table[siguiente_estado] = [0, 0, 0, 0]

            # Fórmula de Q-learning
            q_actual = q_table[estado_actual][accion]
            max_q_siguiente = max(q_table[siguiente_estado])
            nuevo_q = q_actual + alpha * (recompensa + gamma * max_q_siguiente
                - q_actual)
            q_table[estado_actual][accion] = nuevo_q

            # Avanzar al siguiente estado
            estado_actual = siguiente_estado

        # Al final del episodio, podríamos analizar la trayectoria completa
        if (episodio + 1) % 100 == 0:
            print(f"Episodio {episodio + 1}: {len(trayectoria)} pasos")
```

8 Conclusiones

Las estructuras de datos avanzadas en Python proporcionan herramientas poderosas para implementar algoritmos de Aprendizaje por Refuerzo:

- La **flexibilidad y versatilidad** de las estructuras permiten modelar los diferentes componentes necesarios en RL.
- La **eficiencia** de operaciones como búsqueda en diccionarios y verificación de pertenencia en conjuntos facilita implementaciones rápidas.

- La **inmutabilidad** de tuplas garantiza consistencia en representaciones de estados y transiciones.
- La capacidad de **anidamiento y composición** permite construir estructuras complejas para representar entornos y políticas sofisticadas.

Dominar estas estructuras nos proporciona la base necesaria para abordar en próximas sesiones la implementación de algoritmos más avanzados como DQN (Deep Q-Networks), Policy Gradients, y otros métodos fundamentales en el campo del Aprendizaje por Refuerzo.