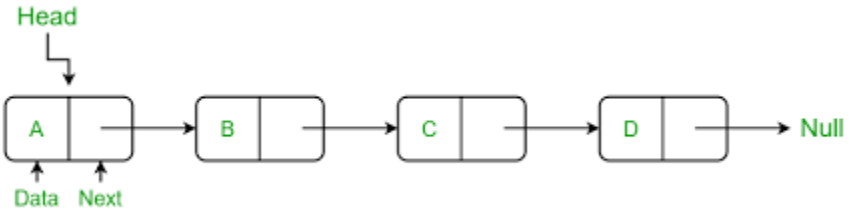
Tema 5: Listas

Héctor Xavier Limón Riaño

April 3, 2024

Contents

1 Fundamentos



- Son estructuras de datos lineales, dinámicas y mutables
- Su forma de uso es muy similar a los arreglos en lenguajes como `C`
- En Python su forma literal es mediante corchetes `[]`
- En general el acceso a elementos es secuencial y no aleatorio (como el de los arreglos), lo cual es ineficiente
- Esto es así porque los elementos no están contiguos en memoria (en los arreglos si)
- Para mitigar el problema anterior, Python tiene varias optimizaciones como utilizar memoria contigua en la medida de lo posible (aun así un arreglo siempre será más eficiente)
- La eficiencia de las listas no es un tema del que preocuparse mucho en Python, salvo que se tengan que manipular mucho datos (millones)

1.1 Tipos principales

- Listas ligadas:
  - Se componen de nodos, donde cada nodo tiene una referencia al nodo siguiente (como se vio en el ejercicio de clase)
- Listas doblemente ligadas
  - También se componen de nodos, pero cada nodo tiene una referencia al nodo siguiente y al anterior, también hay una referencia al primer (cabeza) y último elemento (cola)

2 Operaciones sobre listas

- No se abordaran algunas cosas básicas ya vistas (recorrido de listas, pertenencia a una lista mediante operador `in`)

2.1 Tipos de operaciones sobre estructuras de datos datos

- Entiéndase por operación una función o método
- Lo que se analiza en este punto es un concepto general de programación, no ligado específicamente a listas
- Existen dos tipos principales de operaciones sobre una estructura de datos
  - In-place:
    - La operación modifica la estructura interna, esto es modifica memoria
    - En un objeto, cambia el estado del objeto
    - Son eficientes en memoria pero propensas a tener efectos colaterales (un tema que se verá en segundo parcial)
    - No regresan nada (son procedimientos)
  - Inmutable (no mutables)
    - No modifican la estructura interna
    - En vez de modificar memoria existente, pueden generan nueva memoria (lo cual puede ser ineficiente)
    - Regresan algo (se comportan como funciones)

2.2 Longitud

- Simplemente se usa `len` la cual funciona con varias estructuras de datos de Python
- Es una operación no mutable

```
l = [1, 2, 3]
print(len(l))
```

```
3
```

2.3 Concatenación

- Se refiere a unir listas entre si
- Es una operación no mutable
- Se logra mediante el operador `+`

```
l1 = [1, 2, 3]
l2 = [4, 5]
lres = l1 + l2
print(lres)
```

```
[1, 2, 3, 4, 5]
```

2.4 Agregar elementos

- Son operaciones in-place

2.4.1 Al inicio

- Mediante el método `insert` de una lista
- Este método permite insertar en cualquier posición
- El primer parámetro es una posición y el segundo el elemento a insertar
- Cuidado: `insert` permite posiciones inválidas (mayores a la longitud o menores a 0)

```
l = [2, 3, 4]
l.insert(0, 1)
print(l)
```

```
[1, 2, 3, 4]
```

2.4.2 Al final

- Usar método `append` de lista

```
l = [1, 2, 3]
l.append(4)
print(l)
```

```
[1, 2, 3, 4]
```

2.4.3 En cualquier posición

- Con el método `insert`

```
l = [1, 2, 4]
l.insert(2, 3)
print(l)
```

```
[1, 2, 3, 4]
```

2.5 Remplazar elementos

- Se puede con asignación directa
- Es una operación in-place

```
l = [0, 2, 3]
l[0] = 1
print(l)
```

```
[1, 2, 3]
```

2.6 Borrar elementos

- Operación in-place
- Se utiliza la función general `del` que funciona para varias estructuras de datos de Python

```
l = [0, 1, 2, 3]
del(l[0])
print(l)
```

```
[1, 2, 3]
```

2.7 Obtener sublistas

- Es una operación no mutable
- Usando rebanadas (como se vio en el tema anterior)
- Las rebanadas regresan nueva memoria (por eso son no mutables)

```
l = [1, 2, 3, 4]
print(l[:-1]) # todos menos último
print(l[1:]) # todos menos primero
print(l[1:-1]) # sin primero y último
```

```
[1, 2, 3]
[2, 3, 4]
[2, 3]
```

2.8 Hacer una copia

- Operación no mutable
- Se puede mediante el método `copy` de listas o mediante rebanadas
- CUIDADO: estos métodos hacen copias superficiales (shallow)
- Esto es importante si los elementos de la lista son otras estructuras de datos (como otras listas)
- Si se quiere hacer copias profundas (deep) Python cuenta con un módulo llamada `copy`

```
l = [1, 2, 3]
copia1 = l[:]
copia2 = l.copy()
copia1[0] = 11
copia2[0] = 22
print(l)
print(copia1) # Diferente a l
print(copia2)

l = [[1], [2]] # lista de listas
copia = l[:]
copia[0][0] = 66
print(l) # se alteró por copia superficial
print(copia)

import copy
copia_buena = copy.deepcopy(l)
copia_buena[0][0] = 99
print(l)
print(copia_buena)
```

```
[1, 2, 3]
[11, 2, 3]
[22, 2, 3]
[[66], [2]]
[[66], [2]]
[[66], [2]]
[[99], [2]]
```

2.9 Ordenar listas

- El ordenamiento es un tema extenso de las estructuras de datos, considerándose como un problema computacional muy importante
- Se refiere a ordenar, bajo algún criterio, los elementos de alguna estructura de datos (tradicionalmente arreglos o listas)
- Muchas operaciones críticas (por ejemplo selección de elementos en bases de datos) dependen de que los algoritmos de ordenamiento sean eficientes (tanto en memoria como en procesador)
- Existen muchos algoritmos de ordenamiento, algunos ejemplos clásicos son los siguientes (en este curso se verán algunos como parte de temas o ejercicios) :
  - Burbuja
  - Inserción directa
  - Quicksort
  - Mergesort
  - Heapsort
  - Timsort (es el que usa Python)
- Hay dos variantes principales de operaciones en Python:
  - Método `sort` de lista: in-place
  - Función `sorted`: no mutable, para diferentes estructuras de datos

2.9.1 Orden ascendente

- Por defecto estos métodos ordenan de forma ascendente (de menor a mayor), de acuerdo a criterios de comparación entre objetos del mismo tipo (Python define métodos especiales para comparación de objetos, similar a `__eq__`)

```
l = [44, 11, 7, 22]
l2 = sorted(l)
l.sort()

print(l)
print(l2)
```

```
[7, 11, 22, 44]
[7, 11, 22, 44]
```

2.9.2 Orden descendente

- Se logra con el parámetro nombrado (keyword) `reverse`
- En un tema posterior del curso se hablará más a fondo de este tipo de parámetros

```
l = [44, 11, 7, 22]
l2 = sorted(l, reverse=True)
l.sort(reverse=True)

print(l)
print(l2)
```

```
[44, 22, 11, 7]
[44, 22, 11, 7]
```

- Es posible ordenar de forma más arbitraria o para tipos de datos creados por el programador, sin embargo, se requieren algunos conocimientos que rebasan los alcances de este curso (en lenguajes y paradigmas de programación se debería abordar)

3 Comprensión de listas

- En inglés list comprehensions
- Son una forma de crear listas a partir de otras (es una operación inmutable)
- La idea es generar esa nueva lista aplicando una transformación o filtrado de los elementos de la lista original

- Evita que se tengan que usar ciclos `for` en muchas ocasiones

```
  # forma básica con sólo trasformaciones
nueva = [trasformacion(elemento) for elemento in lista_original]
```

```
original = ['hola', 'mundo']
mayusculas = [s.upper() for s in original]
print(mayusculas)
```

```
['HOLA', 'MUNDO']
```

```
# obtener sólo números pares
nums = [1, 2, 3, 4]
# lo que va después del if es una expresión booleana
pares = [num for num in nums if num % 2 == 0]
print(pares)
```

```
[2, 4]
```

```
# Combinación con transformación y filtrado
# Quedarse sólo con directorios de una ruta
# Agregar la ruta como parte de la salida
import os
ruta = '/tmp'
dirs = ['%s/%s' % (ruta, elemento)
        for elemento in os.listdir(ruta)
        if os.path.isdir('%s/%s' % (ruta, elemento))]

print(dirs)
```

```
['/tmp/nuevoDir', '/tmp/otroDir']
```

4 Listas de varias dimensiones

- También llamadas listas de listas, son listas donde los elementos son a su vez listas
- Permiten definir estructuras muy útiles en cálculos matemáticos (álgebra lineal, graficación, etc.) como las matrices
- En computación aparecen de forma recurrente en la manipulación de imágenes

```
l1 = [1, 2]
l2 = [3, 4]
ll = [l1, l2]
print(ll)
print(ll[0])
print(ll[1][0])
```

```
[[1, 2], [3, 4]]
[1, 2]
3
```

4.1 Matrices, cubos e hipercubos

- Son listas de listas donde el número de elementos en las sublistas coincide

```
matriz = [[1, 2, 3], [3, 4, 5]] # sublistas de 3 elementos
no_matriz = [[1], [3, 4], [5, 6]]
```

- Cada nivel que se agrega se le llama "dimensión", las matrices son de 2 dimensiones, los cubos de 3 y los hipercubos de más de 3
- Para simplificar, en este curso sólo se verá cómo procesar matrices
- Las matrices tienen filas y columnas

```
matrix = [
    [0, 64, 128, 192, 255], # fila 0
    [0, 64, 128, 192, 255],
    [0, 64, 128, 192, 255],
    [0, 64, 128, 192, 255],
    [0, 64, 128, 192, 255]
]

# recuperar elemento en fila 2 columna 3
print(matrix[2][3])
```

```
192
```

4.1.1 Recorrer una matriz

- Se requieren dos ciclos (for o while) uno anidado dentro del otro

```
matriz = [
    [1, 2, 3],
    [4, 5, 6]
]

for fila in matriz:
    for celda in fila:
        print(celda)
```

```
1
2
3
4
5
6
```

- Al estilo `C`

```
matriz = [
    [1, 2, 3],
    [4, 5, 6]
]

for fila in range(len(matriz)):
    for columna in range(len(matriz[fila])):
        print(matriz[fila][columna])
```

```
1
2
3
4
5
6
```

5 Tuplas

- Son una estructura de datos similar a las listas (lineales) excepto que son estáticas y no mutables

- Su forma literal es mediante `()`
- Si no se requiere cambiar elementos, es mejor utilizar tuplas en lugar de listas
- Tiene operaciones similares a las listas pero sólo inmutables
- Con la función `tuple` se puede convertir a tupla (con ciertos tipos de datos)

```
tupla = (1, 2, 3)
print(tupla[0])

print(tuple('hola'))
```

```
1
('h', 'o', 'l', 'a')
```

- Para que no haya ambigüedad, cuando la tupla tenga un elemento se agrega una coma extra al final

```
tupla = (1)
print(type(tupla))

tupla = (1, )
print(type(tupla))
```

```
<class 'int'>
<class 'tuple'>
```