Tipos de datos estructurados

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Introducción

1.1. Conceptos

- Un **dato estructurado** o **dato compuesto** es un dato formado, a su vez, por otros datos llamados **componentes** o **elementos**.
- Un **tipo de dato estructurado**, también llamado **tipo compuesto**, es aquel cuyos valores son datos estructurados.

- Frecuentemente se puede acceder de manera individual a los elementos que componen un dato estructurado y a veces, también, se pueden modificar de manera individual.
- El término **estructura de datos** se suele usar como sinónimo de **tipo de dato estructurado**, aunque nosotros haremos una distinción:
  - Usaremos **tipo de dato estructurado** cuando usemos un dato sin conocer sus detalles internos de implementación.
  - Usaremos **estructura de datos** cuando nos interesen esos detalles internos.

2. Secuencias

2.1. Concepto de secuencia

- Una secuencia es una estructura de datos que:
  - permite el acceso eficiente a sus elementos usando índices enteros, y
  - se le puede calcular su longitud mediante la función `len`.
- Las secuencias se dividen en:
  - **Inmutables**: cadenas (`str`), tuplas (`tuple`), rangos (`range`).
  - **Mutables**: listas (`list`), principalmente.

2.2. Operaciones comunes

- Todas las secuencias (ya sean cadenas, listas, tuplas o rangos) comparten un conjunto de operaciones comunes.
- Además de estas operaciones, las secuencias del mismo tipo admiten comparaciones. Las tuplas y las listas se comparan lexicográficamente elemento a elemento.
  - Eso significa que dos secuencias son iguales si cada elemento es igual y las dos secuencias son del mismo tipo y tienen la misma longitud.
- La siguiente tabla enumera las operaciones sobre secuencias, ordenadas por prioridad ascendente. `s` y `t` son secuencias del mismo tipo, `n`, `i`, `j` y `k` son enteros y `x` es un dato cualquiera que cumple con las restricciones que impone `s`.

Operación	Resultado
x `in` *s*	`True` si algún elemento de *s* es igual a *x*
x `not in` *s*	`False` si algún elemento de *s* es igual a *x*
s `+` *t*	La concatenación de *s* y *t*
s `*` *n*	Equivale a añadir *s* a sí mismo *n* veces
n `*` *s*	
s`[`*i*`]`	El *i*-ésimo elemento de *s*, empezando por 0
s`[`*i*`:`*j*`]`	Rodaja de *s* desde *i* hasta *j*

Operación	Resultado
`s[i:j:k]`	Rodaja de *s* desde *i* hasta *j* con paso *k*
`len(s)`	Longitud de *s*
`min(s)`	El elemento más pequeño de *s*
`max(s)`	El elemento más grande de *s*
`s.index(x [, i [, j ]])`	El índice de la primera aparición de *x* en *s* (desde el índice *i* inclusive y antes del *j*)
`s.count(x)`	Número total de apariciones de *x* en *s*

2.3. Inmutables

2.3.1. Cadenas (`str`)

2.3.1.1. Funciones

2.3.1.2. Métodos

2.3.1.3. Expresiones regulares

2.3.2. Tuplas

- Las tuplas son secuencias inmutables, usadas frecuentemente para almacenar colecciones de datos heterogéneos (de tipos distintos).
- También se usan en aquellos casos en los que se necesita una secuencia inmutable de datos homogéneos (por ejemplo, para almacenar datos en un conjunto o un diccionario).
- Las tuplas se pueden crear:
  - Con paréntesis vacíos, para representar la tupla vacía: `()`
  - Usando una coma detrás de un único elemento:

    ```
    a,
    (a,)
    ```

  - Separando los elementos con comas:

    ```
    a, b, c
    (a, b, c)
    ```

  - Usando el constructor `tuple()`
- Observar que lo que construye la tupla es realmente la coma, no los paréntesis.
- Los paréntesis son opcionales, excepto en dos casos:
  - La tupla vacía: `()`

  - Cuando son necesarios para evitar ambigüedad.

    Por ejemplo, `f(a, b, c)` es una llamada a función con tres argumentos, mientras que `f((a, b, c))` es una llamada a función con un único argumento que es una tupla de tres elementos.

- Las tuplas implementan todas las operaciones comunes de las secuencias.

2.3.3. Rangos

2.4. Mutables

2.4.1. Listas

- En la siguiente tabla, `s` es una instancia de un tipo de secuencia mutable (en nuestro caso, una lista), `t` es cualquier dato iterable y `x` es un dato cualquiera que cumple con las restricciones que impone `s`:

Operación	Resultado
`s[i] = x`	El elemento *i* de *s* se sustituye por *x*
`s[i:j] = t`	La rodaja de *s* desde *i* hasta *j* se sustituye por el contenido del iterable *t*
`del s[i:j]`	Igual que `s[i:j] = []`
`s[i:j:k] = t`	Los elementos de `s[i:j:k]` se sustituyen por los de *t*
`del s[i:j:k]`	Elimina de la secuencia los elementos de `s[i:j:k]`

Operación	Resultado
`s.append(x)`	Añade *x* al final de la secuencia; es igual que `s[len(s):len(s)] = [x]`
`s.clear()`	Elimina todos los elementos de *s*; es igual que `del s[:]`
`s.copy()`	Crea una copia *superficial* de *s*; es igual que `s[:]`
`s.extend(t)`	Extiende *s* con el contenido de *t*; es como hacer
`s += t`	`s[len(s):len(s)] = t`
`s *= n`	Modifica *s* repitiendo su contenido *n* veces
`max(s)`	El elemento más grande de *s*
`s.insert(i,x)`	Inserta *x* en *s* en el índice *i*; es igual que `s[i:i] = [x]`
`s.pop([i])`	Extrae el elemento *i* de *s* y lo devuelve (por defecto, *i* vale -1)
`s.remove(x)`	Quita el primer elemento de *s* que sea igual a *x*
`s.reverse()`	Invierte los elementos de *s*

3. Estructuras no secuenciales

3.1. Conjuntos (`set` y `frozenset`)

3.2. Diccionarios (`dict`)

3.2.1. *Hashables*

4. Iterables

4.1. Iteradores