

Tipos de datos estructurados

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Introducción	1
1.1. Conceptos	1
1.2. <i>Hashables</i>	2
2. Secuencias	2
2.1. Concepto de secuencia	2
2.2. Operaciones comunes	3
2.3. Inmutables	3
2.3.1. Cadenas (<i>str</i>)	3
2.3.2. Tuplas	4
2.3.3. Rangos	4
2.4. Mutables	5
2.4.1. Listas	5
3. Estructuras no secuenciales	7
3.1. Conjuntos (<i>set</i> y <i>frozenset</i>)	7
3.1.1. Operaciones	8
3.1.2. Operaciones sobre conjuntos mutables	8
3.2. Diccionarios (<i>dict</i>)	9
3.2.1. Operaciones	10
4. Iterables	10
4.1. Iteradores	10

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.

1.2. Hashables

- Un valor es *hashable* si cumple las siguientes dos condiciones:
 - Tiene asociado un valor *hash* que nunca cambia durante su vida.
Para ello debe responder al método `__hash__()`, que es el que se invoca cuando se usa la función `hash()` sobre el valor.
 - Puede compararse con otros valores para ver si es igual a alguno de ellos.
Para ello debe responder al método `__eq__()`, que es el que se invoca cuando se usa el operador `==` con ese valor como su primer argumento.
- Si dos valores *hashables* son iguales, entonces deben tener el mismo valor de *hash*.
- La mayoría de los valores inmutables predefinidos en Python son *hashables*.
- Los contenedores mutables (como las listas o los diccionarios) **no** son *hashables*.
- Los contenedores inmutables (como las tuplas o los `frozensets`) sólo son *hashables* si sus elementos también lo son.
- El concepto de *hashable* es importante en Python ya que existen tipos de datos estructurados que sólo admiten valores *hashables*.
- Por ejemplo, los elementos de un conjunto y las claves de un diccionario deben ser *hashables*.

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`).
- **Mutable:** 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
<code>x in s</code>	<code>True</code> si algún elemento de <code>s</code> es igual a <code>x</code>
<code>x not in s</code>	<code>False</code> si algún elemento de <code>s</code> es igual a <code>x</code>
<code>s + t</code>	La concatenación de <code>s</code> y <code>t</code>
<code>s * n</code>	Equivale a añadir <code>s</code> a sí mismo <code>n</code> veces
<code>n * s</code>	
<code>s[i]</code>	El <i>i</i> -ésimo elemento de <code>s</code> , empezando por 0
<code>s[i:j]</code>	Rodaja de <code>s</code> desde <code>i</code> hasta <code>j</code>
<code>s[i:j:k]</code>	Rodaja de <code>s</code> desde <code>i</code> hasta <code>j</code> con paso <code>k</code>
<code>len(s)</code>	Longitud de <code>s</code>
<code>min(s)</code>	El elemento más pequeño de <code>s</code>
<code>max(s)</code>	El elemento más grande de <code>s</code>
<code>s.index(x[, i[, j]])</code>	El índice de la primera aparición de <code>x</code> en <code>s</code> (desde el índice <code>i</code> inclusive y antes del <code>j</code>)
<code>s.count(x)</code>	Número total de apariciones de <code>x</code> en <code>s</code>

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 la función `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

- Los **rangos** representan secuencias inmutables de números y se usan frecuentemente para hacer bucles que se repitan un determinado número de veces.
- Los rangos se crean con la función `range()`:

```
<rangos> ::= range(<fin>)  
          | range(<inicio>, <fin>[, <paso>])
```

donde `<inicio>`, `<fin>` y `<paso>` son números enteros.

- Si se omite `<inicio>`, su valor por defecto es 0.
- Si se omite `<paso>`, su valor por defecto es 1.
- El contenido de un rango `r` vendrá determinado por la fórmula $r[i] = inicio + fin \cdot i$, donde $i \geq 0$. Además:

- Si el paso es positivo, se impone también la restricción $r[i] < fin$.
- Si el paso es negativo, se impone también la restricción $r[i] > fin$.
- Un rango es vacío cuando $r[0]$ no satisface las restricciones anteriores.
- Los rangos admiten índices negativos, pero se interpretan como si se indexara desde el final de la secuencia usando índices positivos.
- Los rangos implementan todas las operaciones de las secuencias, *excepto* la concatenación y la repetición.

Esto es debido a que los rangos sólo pueden representar secuencias que siguen un patrón muy estricto, y las repeticiones y las concatenaciones a menudo violan ese patrón.

- Los rangos ocupan mucha menos memoria que las listas o las tuplas.
- Ejemplos:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

- Dos rangos son considerados iguales si representan la misma secuencia de valores, sin importar si tienen distintos valores de *<inicio>*, *<fin>* o *<paso>*.
- Por ejemplo:

```
>>> range(0) == range(2, 1, 3)
True
>>> range(0, 3, 2) == range(0, 4, 2)
True
```

2.4. Mutables

2.4.1. Listas

- Las **listas** son secuencias *mutables*, usadas frecuentemente para almacenar colecciones de elementos heterogéneos.
- Se pueden construir de varias maneras:
 - Usando corchetes vacíos para representar la lista vacía: `[]`
 - Usando corchetes y separando los elementos con comas:

[a]

[a, b, c]

- Usando una lista por comprensión: `[x for x in <iterable>]`
- Usando la función `list`: `list()` o `list(<iterable>)`
- La función `list` construye una lista cuyos elementos son los mismos (y están en el mismo orden) que los elementos de `<iterable>`.
- `<iterable>` puede ser:
 - una secuencia,
 - un contenedor sobre el que se pueda iterar, o
 - un iterador.
- Si se llama sin argumentos, devuelve una lista vacía.
- Por ejemplo:

```
>>> list('abc')
['a', 'b', 'c']
>>> list( (1, 2, 3) )
[1, 2, 3]
```

- 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
<code>s[i] = x</code>	El elemento <code>i</code> de <code>s</code> se sustituye por <code>x</code>
<code>s[i:j] = t</code>	La rodaja de <code>s</code> desde <code>i</code> hasta <code>j</code> se sustituye por el contenido del iterable <code>t</code>
<code>del s[i:j]</code>	Igual que <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	Los elementos de <code>s[i:j:k]</code> se sustituyen por los de <code>t</code>
<code>del s[i:j:k]</code>	Elimina de la secuencia los elementos de <code>s[i:j:k]</code>

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

Operación	Resultado
<code>s.remove(x)</code>	Quita el primer elemento de <code>s</code> que sea igual a <code>x</code>
<code>s.reverse()</code>	Invierte los elementos de <code>s</code>

- Las listas, además, admiten el método `sort()`:

```
>>> x = [3, 6, 2, 9, 1, 4]
>>> x.sort()
>>> x
[1, 2, 3, 4, 6, 9]
>>> x.sort(reverse=True)
>>> x
[9, 6, 4, 3, 2, 1]
```

3. Estructuras no secuenciales

3.1. Conjuntos (set y frozenset)

- Un conjunto es una colección no ordenada de elementos *hashables*. Se usan frecuentemente para comprobar si un elemento pertenece a un grupo, para eliminar duplicados en una secuencia y para realizar operaciones matemáticas como la unión, la intersección y la diferencia simétrica.
- Como cualquier otra colección, los conjuntos permiten el uso de:
 - `x in c`
 - `len(c)`
 - `for x in c`
- Como son colecciones no ordenadas, los conjuntos no almacenan la posición de los elementos o el orden en el que se insertaron.
- Además, tampoco admite la indexación, las rodajas ni cualquier otro comportamiento propio de las secuencias.
- Existen dos tipos predefinidos de conjuntos: `set` y `frozenset`.
- El tipo `set` es mutable, es decir, que su contenido puede cambiar usando métodos como `add()` y `remove()`.
 - Como es mutable, no tiene valor *hash* y, por tanto, no puede usarse como clave de un diccionario o como elemento de otro conjunto.
- El tipo `frozenset` es inmutable y *hashable*. Por tanto, su contenido no se puede cambiar una vez creado y puede usarse como clave de un diccionario o como elemento de otro conjunto.
- Los conjuntos se crean con las funciones `set()` y `frozenset()`.
- Además, los conjuntos `set` no vacíos se pueden crear encerrando entre llaves una lista de elementos separados por comas: `{'pepe', 'juan'}`.

3.1.1. Operaciones

- `s` y `o` son conjuntos, y `x` es un valor cualquiera:

Operación	Resultado
<code>len(s)</code>	Número de elementos de <code>s</code> (su cardinalidad)
<code>x in s</code>	<code>True</code> si <code>x</code> pertenece a <code>s</code>
<code>x not in s</code>	<code>True</code> si <code>x</code> no pertenece a <code>s</code>
<code>s.isdisjoint(o)</code>	<code>True</code> si <code>s</code> no tiene ningún elemento en común con <code>o</code>
<code>s.issubset(o)</code>	<code>True</code> si <code>s</code> es un subconjunto de <code>o</code>
<code>s <= o</code>	
<code>s < o</code>	<code>True</code> si <code>s</code> es un subconjunto propio de <code>o</code>
<code>s.is superset(o)</code>	<code>True</code> si <code>s</code> es un superconjunto de <code>o</code>
<code>s >= o</code>	
<code>s > o</code>	<code>True</code> si <code>s</code> es un superconjunto propio de <code>o</code>

Operación	Resultado
<code>s.union(o)</code>	Unión de <code>s</code> y <code>o</code> ($s \cup o$)
<code>s o</code>	
<code>s.intersection(o)</code>	Intersección de <code>s</code> y <code>o</code> ($s \cap o$)
<code>s & o</code>	
<code>s.difference(o)</code>	Diferencia entre <code>s</code> y <code>o</code> ($s \setminus o$)
<code>s - o</code>	
<code>s.symmetric_difference(o)</code>	Diferencia simétrica entre <code>s</code> y <code>o</code> ($s \Delta o$)
<code>s ^ o</code>	
<code>s.copy()</code>	Devuelve una copia superficial de <code>s</code>

- Tanto `set` como `frozenset` admiten comparaciones entre conjuntos.
- Dos conjuntos son iguales si y sólo si cada elemento de un conjunto pertenece también al otro conjunto, y viceversa; es decir, si cada uno es un subconjunto del otro.
- Un conjunto es menor que otro si y sólo si el primer conjunto está contenido en el otro; es decir, si el primero es un subconjunto propio del segundo (es un subconjunto, pero no es igual).
- Un conjunto es mayor que otro si y sólo si el primero es un superconjunto propio del segundo (es un superconjunto, pero no es igual).

3.1.2. Operaciones sobre conjuntos mutables

- Estas tablas sólo se aplica a conjuntos mutables (o sea, al tipo `set` y no al `frozenset`):

Operación	Resultado
<code>s.update(o)</code>	Actualiza <code>s</code> añadiendo los elementos de <code>o</code>
<code>s = o</code>	

Operación	Resultado
<code>s.intersection_update(o)</code> <code>s &= o</code>	Actualiza <code>s</code> manteniendo sólo los elementos que están en <code>s</code> y <code>o</code>
<code>s.difference_update(o)</code> <code>s -= o</code>	Actualiza <code>s</code> eliminando los elementos que están en <code>o</code>
<code>s.symmetric_difference_update(o)</code> <code>s ^= o</code>	Actualiza <code>s</code> manteniendo sólo los elementos que están en <code>s</code> y <code>o</code> pero no en ambos

Operación	Resultado
<code>s.add(x)</code>	Añade <code>x</code> a <code>s</code>
<code>s.remove(x)</code>	Elimina <code>x</code> de <code>s</code> (produce <code>KeyError</code> si <code>x</code> no está en <code>s</code>)
<code>s.discard(x)</code>	Elimina <code>x</code> de <code>s</code> si está en <code>s</code>
<code>s.pop()</code>	Elimina y devuelve un valor cualquiera de <code>s</code> (produce <code>KeyError</code> si <code>s</code> está vacío)
<code>s.clear()</code>	Elimina todos los elementos de <code>s</code>

3.2. Diccionarios (dict)

- Un **diccionario** es un dato que almacena *correspondencias* (o asociaciones) entre valores.
- Tales correspondencias son datos mutables.
- Los diccionarios se pueden crear:
 - Encerrando entre llaves una lista de pares `<clave>:<valor>` separados por comas: `{'juan': 4098, 'pepe': 4127}`
 - Usando la función `dict()`.
- Las claves de un diccionario pueden ser *casi* cualquier valor.
- No se pueden usar como claves los valores que no sean *hashables*, es decir, los que contengan listas, diccionarios o cualquier otro tipo mutable.
- Los tipos numéricos que se usen como claves obedecen las reglas normales de comparación numérica.
 - Por tanto, si dos números son considerados iguales (como `1` y `1.0`) entonces se consideran la misma clave en el diccionario.
- Los diccionarios se pueden crear usando la función `dict()`. Por ejemplo:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

3.2.1. Operaciones

- `d` y `o` son diccionarios, `c` es una clave válida y `v` es un valor cualquiera:

Operación	Resultado
<code>d[c] = v</code>	Asigna a <code>d[c]</code> el valor <code>v</code>
<code>del d[c]</code>	Elimina <code>d[c]</code> de <code>d</code> (produce <code>KeyError</code> si <code>c</code> no está en <code>d</code>)
<code>c in d</code>	<code>True</code> si <code>d</code> contiene una clave <code>c</code>
<code>c not in d</code>	<code>True</code> si <code>d</code> no contiene una clave <code>c</code>
<code>d.clear()</code>	Elimina todos los elementos del diccionario
<code>d.copy()</code>	Devuelve una copia superficial del diccionario
<code>d.get(c, def)</code>	Devuelve el valor de <code>c</code> si <code>c</code> está en <code>d</code> ; en caso contrario, devuelve <code>def</code> (por defecto, <code>def</code> vale <code>None</code>)
<code>d.pop(c, def)</code>	Elimina y devuelve el valor de <code>c</code> si <code>c</code> está en <code>d</code> ; en caso contrario, devuelve <code>def</code> (si no se pasa <code>def</code> y <code>c</code> no está en <code>d</code> , produce un <code>KeyError</code>)

Operación	Resultado
<code>d.popitem()</code>	Elimina y devuelve una pareja (<i>clave, valor</i>) del diccionario siguiendo un orden LIFO (produce un <code>KeyError</code> si <code>d</code> está vacío)
<code>d.setdefault(c, def)</code>	Si <code>c</code> está en <code>d</code> , devuelve su valor; si no, inserta <code>c</code> en <code>d</code> con el valor <code>def</code> y devuelve <code>def</code> (por defecto, <code>def</code> vale <code>None</code>)
<code>d.update(o)</code>	Actualiza <code>d</code> con las parejas (<i>clave, valor</i>) de <code>o</code> , sobrescribiendo las claves ya existentes, y devuelve <code>None</code>

4. Iterables

4.1. Iteradores