Tipos de datos estructurados

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Conceptos básicos

1.1. Introducción

- 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 `frozenset`s) 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*.

1.3. Iterables

- Un **iterable** es un dato compuesto que tiene la capacidad de devolver sus elementos de uno en uno.
- Como iterables tenemos:
  - Todas las secuencias: listas, cadenas, tuplas, rangos…

  - Estructuras no secuenciales: diccionarios, conjuntos…
- Los iterables pueden usarse en un bucle `for` y en muchos otros lugares donde se necesite una secuencia (funciones `zip()`, `map()`, etc.).
- La forma manual de recorrer un dato iterable es usando un **iterador**.

1.4. Iteradores

- Un **iterador** representa un flujo de datos.
- Cuando se llama repetidamente a la función `next()` aplicada a un iterador, se van obteniendo los sucesivos elementos del flujo.
- Cuando ya no hay más elementos disponibles, se levanta una excepción de tipo `StopIteration`.

  Eso indica que el iterador se ha agotado, por lo que si se sigue llamando a la función `next()` se seguirá levantando la excepción `StopIteration`.
- Se puede obtener un iterador a partir de cualquier dato iterable usando la función `iter()` con el iterable.
- Si se le pasa un valor no iterable, levanta una excepción `TypeError`.
- Ejemplo:

```
>>> lista = [1, 2, 3, 4]
>>> it = iter(lista)
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
4
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- Los iteradores se pueden convertir en listas o tuplas usando las funciones `list()` y `tuple()`:

```
>>> l = [1, 2, 3]
>>> iterador = iter(l)
>>> t = tuple(iterador)
>>> t
(1, 2, 3)
```

- Los iteradores también son iterables que actúan como sus propios iteradores.
- Por tanto, cuando llamamos a `iter()` pasándole un iterador, se devuelve el mismo iterador:

```
>>> lista = [1, 2, 3, 4]
>>> it = iter(lista)
>>> it
<list_iterator object at 0x7f3c49aa9080>
```

```
>>> it2 = iter(it)
>>> it2
<list_iterator object at 0x7f3c49aa9080>
```

- Por tanto, también podemos usar un iterador en cualquier sitio donde se espere un iterable.

1.5. El bucle `for`

- Probablemente, la mejor forma de recorrer los elementos que devuelve un iterador es mediante un bucle `for`.
- Su sintaxis es:

```
for <variable>(, <variable>)* in <iterable>:
    <sentencia>
```

que equivale a:

```
iterador = iter(<iterable>)
fin = False
while not fin:
    try:
        <variable>(, <variable>)* = next(iterador)
    except StopIteration:
        fin = True
    else:
        <sentencia>
```

1.6. El módulo `itertools`

- El módulo `itertools` contiene una variedad de iteradores de uso frecuente así como funciones que combinan varios iteradores.
- Veremos algunos ejemplos.
- `itertools.count(`[*<inicio>*`[, `*<paso>*`]])` devuelve un flujo infinito de valores separados uniformemente. Se puede indicar opcionalmente un valor de comienzo (que por defecto es 0) y el intervalo entre números (que por defecto es 1):
  - `itertools.count()` ⇒ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
  - `itertools.count(10)` ⇒ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
  - `itertools.count(10, 5)` ⇒ 10, 15, 20, 25, 30, 35, 40, 45, ...
- `itertools.cycle(`*<iterador>*`)` devuelve un nuevo iterador que va generando sus elementos del primero al último, repitiéndolos indefinidamente:
  - `itertools.cycle([1, 2, 3, 4])` ⇒ 1, 2, 3, 4, 1, 2, 3, 4, ...
- `itertools.repeat(`*<elem>*`,` [*<n>*]`)` devuelve *<n>* veces el elemento *<elem>*, o lo devuelve indefinidamente si no se indica *<n>*:
  - `itertools.repeat('abc')` ⇒ abc, abc, abc, abc, abc, abc, abc, ...

  - `itertools.repeat('abc', 5)` ⇒ abc, abc, abc, abc, abc

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*
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`)

- Las **cadenas** son secuencias inmutables de caracteres.
- No olvidemos que en Python no existe el tipo *carácter*. En Python, un carácter es una cadena de longitud 1.
- Las cadenas implementan todas las operaciones de las secuencias, además de los métodos que se pueden consultar en https://docs.python.org/3/library/stdtypes.html#string-methods

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

  <rango> ::= **`range(`**[*<inicio>*`,`] *<fin>*[`,` *<paso>*]**`)`**

- *<inicio>*, *<fin>* y *<paso>* deben ser números enteros.
- Cuando se omite *<inicio>*, se entiende que es `0`.
- El valor de *<fin>* no se alcanza nunca.
- Cuando *<inicio>* y *<fin>* son iguales, representa el *rango vacío*.
- Cuando *<inicio>* es mayor que *<fin>*, el *<paso>* debería ser negativo.
  En caso contrario, también representaría el rango vacío.
- 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
`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*

- 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
`len(`*s*`)`	Número de elementos de *s* (su cardinalidad)
x `in` *s*	`True` si *x* pertenece a *s*
x `not in` *s*	`True` si *x* no pertenece a *s*
s`.isdisjoint(`*o*`)`	`True` si *s* no tiene ningún elemento en común con *o*
s`.issubset(`*o*`)`	`True` si *s* es un subconjunto de *o*
s `<=` *o*	
s `<` *o*	`True` si *s* es un subconjunto propio de *o*
s`.isuperset(`*o*`)`	`True` si *s* es un superconjunto de *o*
s `>=` *o*	

Operación	Resultado
s < *o*	`True` si *s* es un superconjunto propio de *o*

Operación	Resultado
s.union(*o*) *s* | *o*	Unión de *s* y *o* ($s \cup o$)
s.intersection(*o*) *s* & *o*	Intersección de *s* y *o* ($s \cap o$)
s.difference(*o*) *s* - *o*	Diferencia entre *s* y *o* ($s \setminus o$)
s.symmetric_difference(*o*) *s* ^ *o*	Diferencia simétrica entre *s* y *o* ($s \triangle o$)
s.copy()	Devuelve una copia superficial de *s*

- 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
s.update(*o*) *s* |= *o*	Actualiza *s* añadiendo los elementos de *o*
s.intersection_update(*o*) *s* &= *o*	Actualiza *s* manteniendo sólo los elementos que están en *s* y *o*
s.difference_update(*o*) *s* -= *o*	Actualiza *s* eliminando los elementos que están en *o*
s.symmetric_difference_update(*o*) *s* ^= *o*	Actualiza *s* manteniendo sólo los elementos que están en *s* y *o* pero no en ambos

Operación	Resultado
s.add(*x*)	Añade *x* a *s*
s.remove(*x*)	Elimina *x* de *s* (produce `KeyError` si *x* no está en *s*)
s.discard(*x*)	Elimina *x* de *s* si está en *s*

Operación	Resultado
`s.pop()`	Elimina y devuelve un valor cualquiera de *s* (produce `KeyError` si *s* está vacío)
`s.clear()`	Elimina todos los elementos de *s*

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
`d[c] = v`	Asigna a `d[c]` el valor *v*
`del d[c]`	Elimina `d[c]` de *d* (produce `KeyError` si *c* no está en *d*)
`c in d`	`True` si *d* contiene una clave *c*
`c not in d`	`True` si *d* no contiene una clave *c*
`d.clear()`	Elimina todos los elementos del diccionario
`d.copy()`	Devuelve una copia superficial del diccionario

Operación	Resultado
d`.get(`*c*[`,`*def*]`)`	Devuelve el valor de *c* si *c* está en *d*; en caso contrario, devuelve *def* (por defecto, *def* vale `None`)
d`.pop(`*c*[`,`*def*]`)`	Elimina y devuelve el valor de *c* si *c* está en *d*; en caso contrario, devuelve *def* (si no se pasa *def* y *c* no está en *d*, produce un `KeyError`)

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

Bibliografía

Python Software Foundation. n.d. “Sitio Web de Documentación de Python.” https://docs.python.org/3.