

Tipos de datos estructurados

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Conceptos básicos	1
1.1. Introducción	2
1.2. <i>Hashables</i>	2
1.3. Iterables	2
1.4. Iteradores	3
1.4.1. Expresiones generadoras	4
1.4.2. El bucle <code>for</code>	4
1.4.3. El módulo <code>itertools</code>	5
2. Secuencias	5
2.1. Concepto de secuencia	5
2.2. Operaciones comunes	6
2.3. Inmutables	6
2.3.1. Cadenas (<code>str</code>)	6
2.3.2. Tuplas	8
2.3.3. Rangos	9
2.4. Mutables	10
2.4.1. Listas	10
3. Estructuras no secuenciales	12
3.1. Conjuntos (<code>set</code> y <code>frozenset</code>)	12
3.1.1. Operaciones	13
3.1.2. Operaciones sobre conjuntos mutables	13
3.2. Diccionarios (<code>dict</code>)	14
3.2.1. Operaciones	15
Bibliografía	15

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 `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*.

1.3. Iterables

Un **iterable** es un dato compuesto que se puede recorrer o visitar elemento a elemento, es decir, que se puede *iterar* por sus elementos uno a uno.

Como iterables tenemos:

- Todas las secuencias: listas, cadenas, tuplas y rangos

- Estructuras no secuenciales: diccionarios y conjuntos

No representa un tipo concreto, sino más bien una *familia* de tipos que comparten la misma propiedad.

Muchas funciones, como `map()` y `filter()`, actúan sobre iterables en general, en lugar de hacerlo sobre un tipo concreto.

La forma básica de recorrer un dato iterable es usando un **iterador**.

1.4. Iteradores

Un **iterador** representa un flujo de datos *perezoso* (no se entregan todos de una vez, sino de uno en uno).

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 esa excepción.

Se puede obtener un iterador a partir de cualquier dato iterable aplicando la función `iter()` al iterable.

Si se le pasa un dato 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
```

También se suele decir que **los iteradores son iterables perezosos de un solo uso**:

- Son **perezosos** porque calculan sus elementos a medida que los vas recorriendo.
- Son **de un solo uso** porque, una vez que se ha consumido un elemento, ya no vuelve a aparecer.

Se dice que un iterador está **agotado** si se ha consumido completamente, es decir, si se han consumido todos sus elementos.

Funciones como `map()` y `filter()` devuelven iteradores porque, al ser perezosos, son más eficiente en memoria que devolver toda una lista o tupla.

Por ejemplo: ¿qué ocurre si sólo necesitamos los primeros elementos del resultado de un `map()`?

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.

1.4.1. Expresiones generadoras

Una **expresión generadora** es una expresión que **devuelve un iterador** y que tiene la misma sintaxis que las **listas por comprensión**, salvo que va encerrada entre paréntesis en lugar de entre corchetes:

```
<expr_gen> ::= (<expresión> (for <identificador> in <secuencia> [if <condición>]))+
```

Ejemplo:

```
>>> cuadrados = (x ** 2 for x in range(1, 10))
>>> cuadrados
<generator object <genexpr> at 0x7f6a0fc7db48>
>>> next(cuadrados)
1
>>> next(cuadrados)
4
>>> next(cuadrados)
9
```

1.4.2. 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>
```

1.4.3. 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()` \Rightarrow 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
- `itertools.count(10)` \Rightarrow 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
- `itertools.count(10, 5)` \Rightarrow 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])` \Rightarrow 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')` \Rightarrow abc, abc, abc, abc, abc, abc, abc, ...
- `itertools.repeat('abc', 5)` \Rightarrow 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
<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)

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. Formateado de cadenas

Una **cadena formateada** (también llamada **f-string**) es una cadena literal que lleva un prefijo `f` o `F`.

Estas cadenas contienen **campos de sustitución**, que son expresiones encerradas entre llaves.

En realidad, las cadenas formateadas son expresiones evaluadas en tiempo de ejecución.

Sintaxis:

```
<f_string> ::= (<carácter_literal> | {{ | }} | <sustitución>)*
<sustitución> ::= {<expresión> [!<conversión>] [:<especif>]}
<conversión> ::= s | r | a
<especif> ::= (<carácter_literal> | NULL | <sustitución>)*
<carácter_literal> ::= <cualquier code point excepto ****, **** o **'NULL'***>
```

Las partes de la cadena que van fuera de las llaves se tratan literalmente, excepto las dobles llaves {{ y }}, que son sustituidas por una sola llave.

Una { marca el comienzo de un **campo de sustitución** (<sustitución>), que empieza con una expresión.

Tras la expresión puede venir un **conversión** (<conversión>), introducida por una exclamación !.

También puede añadirse un **especificador de formato** (<especif>) después de dos puntos :.

El campo de sustitución termina con una }.

Las expresiones en un literal de cadena formateada son tratadas como cualquier otra expresión Python encerrada entre paréntesis, con algunas excepciones:

- No se permiten expresiones vacías.
- Las expresiones lambda deben ir entre paréntesis.

Los campos de sustitución pueden contener saltos de línea pero no comentarios.

Si se indica una conversión, el resultado de evaluar la expresión se convierte antes de aplicar el formateado.

La conversión !s llama a `str()` sobre el resultado, !r llama a `repr()` y !a llama a `ascii()`.

A continuación, el resultado es formateado usando `format()`.

Finalmente, el resultado del formateado es incluido en el valor final de la cadena completa.

La sintaxis general de un especificador de formato es:

```
<especif> ::= [[<relleno>]<alig>][<signo>][#][0][<ancho>][<grupos>][.<precision>][<tipo>]
<relleno> ::= <cualquier carácter>
<alig> ::= < | > | = | ^
<signo> ::= + | - | <espacio>
<ancho> ::= <dígito>+
<grupos> ::= _ | ,
<precision> ::= <dígito>+
<tipo> ::= b | c | d | e | E | f | F | g | G | n | o | s | x | X | %
```

Los especificadores de formato de nivel superior pueden incluir campos de sustitución anidados.

Estos campos anidados pueden incluir, a su vez, sus propios campos de conversión y sus propios especificadores de formato, pero no pueden incluir más campos de sustitución anidados.

Para más información, consultar <https://docs.python.org/3.7/library/string.html#format-specification-mini-language>

Algunos ejemplos de cadenas formateadas:

```
>>> nombre = "Fred"
>>> f"Dice que su nombre es {nombre!r}."
"Dice que su nombre es 'Fred'."
>>> f"Dice que su nombre es {repr(nombre)}." # repr() es equivalente a !r
"Dice que su nombre es 'Fred'."
>>> ancho = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{ancho}.{precision}}" # campos anidados
'result:      12.35'
>>> import datetime
>>> hoy = datetime.datetime(year=2017, month=1, day=27)
>>> f"{hoy:%B %d, %Y}" # usando especificador de formato de fecha
'January 27, 2017'
>>> numero = 1024
>>> f"{numero:#0x}" # usando especificador de formato de enteros
'0x400'
```

2.3.1.2. Expresiones regulares

Las **expresiones regulares** (también llamados *regex*) son, esencialmente, un pequeño lenguaje de programación muy especializado incrustado dentro de Python y disponible a través del módulo `re`.

Usando este pequeño lenguaje es posible especificar reglas para comprobar si una cadena se ajusta a un patrón.

Este patrón puede ser frases en español, o direcciones de correo electrónico o cualquier otra cosa.

A continuación, se pueden hacer preguntas del tipo: «¿Esta cadena se ajusta al patrón?» o «¿Hay algo que se ajuste al patrón en alguna parte de esta cadena?».

También se pueden usar las *regexes* para modificar una cadena o dividirla en partes.

El lenguaje de las expresiones regulares es relativamente pequeño y restringido, por lo que no es posible usarlo para realizar cualquier tipo de procesamiento de cadenas.

Además, hay procesamientos que se pueden realizar con *regexes* pero las expresiones que resultan se vuelven muy complicadas.

En estos casos, es mejor escribir directamente código Python.

Aunque el código resultante pueda resultar más lento, probablemente resulte más fácil de leer.

Para más información sobre cómo crear y usar expresiones regulares, consultar:

- Tutorial de introducción en <https://docs.python.org/3/howto/regex.html>
- Documentación del módulo `re` en <https://docs.python.org/3/library/re.html>

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 $r[j] > fin$.

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 son perezosos y, por tanto, 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>)
<code>s.remove(x)</code>	Quita el primer elemento de <code>s</code> que sea igual a <code>x</code>

Operación	Resultado
<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>

Bibliografía

Python Software Foundation. n.d. "Sitio Web de Documentación de Python." <https://docs.python.org/3>.