el cual, a la vez, es lo mismo que:

```
>>> transpuesta = []
>>> for i in range(4):
...     # las siguientes 3 lineas hacen la comprensión de listas anidada
...     fila_transpuesta = []
...     for fila in matriz:
...         fila_transpuesta.append(fila[i])
...     transpuesta.append(fila_transpuesta)
...
>>> transpuesta
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

En el mundo real, deberías preferir funciones predefinidas a declaraciones con flujo complejo. La función **`zip()`** haría un buen trabajo para este caso de uso:

```
 >>> list(zip(*matriz))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Ver *Desempaquetando una lista de argumentos* para detalles en el asterisco de esta línea.

La instrucción

Hay una manera de quitar un ítem de una lista dado su índice en lugar de su valor: la instrucción **`del`**. Esta es diferente del método **`pop()`**, el cual devuelve un valor. La instrucción **`del`** también puede usarse para quitar secciones de una lista o vaciar la lista completa (lo que hacíamos antes asignando una lista vacía a la sección). Por ejemplo:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` puede usarse también para eliminar variables:

```
>>> del a
```

Hacer referencia al nombre a de aquí en más es un error (al menos hasta que se le asigne otro valor). Veremos otros usos para **`del`** más adelante.

Tuplas y secuencias

Vimos que las listas y cadenas tienen propiedades en común, como el indizado y las operaciones de seccionado. Estas son dos ejemplos de datos de tipo *secuencia* (ver *Tipos integrados*). Como Python es un lenguaje en evolución, otros datos de tipo secuencia pueden agregarse. Existe otro dato de tipo secuencia estándar: la *tupla*.

Una tupla consiste de un número de valores separados por comas, por ejemplo:

```
>>> t = 12345, 54321, 'hola!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hola!')
>>> # Las tuplas pueden anidarse:
```

```
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hola!'), (1, 2, 3, 4, 5))
>>> # Las tuplas son inmutables:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # pero pueden contener objetos mutables:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Como puedes ver, en la salida las tuplas siempre se encierran entre paréntesis, para que las tuplas anidadas puedan interpretarse correctamente; pueden ingresarse con o sin paréntesis, aunque a menudo los paréntesis son necesarios de todas formas (si la tupla es parte de una expresión más grande). No es posible asignar a los ítems individuales de una tupla, pero sin embargo sí se puede crear tuplas que contengan objetos mutables, como las listas.

A pesar de que las tuplas puedan parecerse a las listas, frecuentemente se utilizan en distintas situaciones y para distintos propósitos. Las tuplas son *inmutables* y normalmente contienen una secuencia heterogénea de elementos que son accedidos al desempaquetar (ver más adelante en esta sección) o indizar (o incluso acceder por atributo en el caso de las **`namedtuples`**). Las listas son *mutables*, y sus elementos son normalmente homogéneos y se acceden iterando a la lista.

Un problema particular es la construcción de tuplas que contengan 0 o 1 ítem: la sintaxis presenta algunas peculiaridades para estos casos. Las tuplas vacías se construyen mediante un par de paréntesis vacío; una tupla con un ítem se construye poniendo una coma a continuación del valor (no alcanza con encerrar un único valor entre paréntesis). Feo, pero efectivo. Por ejemplo:

```
>>> vacia = ()
>>> singleton = 'hola',    # <-- notar la coma al final
>>> len(vacia)
0
>>> len(singleton)
1
>>> singleton
('hola',)
```

La declaración `t = 12345, 54321, 'hola!'` es un ejemplo de *empaquetado de tuplas*: los valores `12345`, `54321` y `'hola!'` se empaquetan juntos en una tupla.

La operación inversa también es posible:

```
>>> x, y, z = t
```

Esto se llama, apropiadamente, *desempaquetado de secuencias*, y funciona para cualquier secuencia en el lado derecho del igual. El desempaquetado de secuencias requiere que la cantidad de variables a la izquierda del signo igual sea el tamaño de la secuencia. Notá que la asignación múltiple es en realidad sólo una combinación de empaquetado de tuplas y desempaquetado de secuencias.

Conjuntos

Python también incluye un tipo de dato para *conjuntos*. Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas. Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia, y diferencia simétrica.

Las llaves o la función **`set()`** pueden usarse para crear conjuntos. Notá que para crear un conjunto vacío tenés que usar `set()`, no `{}`; esto último crea un diccionario vacío, una estructura de datos que discutiremos en la sección siguiente.

Una pequeña demostración:

```
>>> canasta = {'manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana'}
>>> print fruta                  # muestra que se removieron los duplicados
{'pera', 'manzana', 'banana', 'naranja'}
>>> 'naranja' in canasta         # verificación de pertenencia rápida
True
>>> 'yerba' in canasta
False

>>> # veamos las operaciones para las letras únicas de dos palabras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # letras únicas en a
{a', 'r', 'b', 'c', 'd'}
>>> a - b                             # letras en a pero no en b
{'r', 'b', 'd'}
>>> a | b                             # letras en a o en b
{'a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'}
>>> a & b                             # letras en a y en b
{'a', 'c'}
>>> a ^ b                             # letras en a o b pero no en ambos
{'b', 'd', 'm', 'l', 'r', 'z'}
```

De forma similar a las *comprensiones de listas*, está también soportada la comprensión de conjuntos:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

Diccionarios

Otro tipo de dato útil incluído en Python es el *diccionario* (ver *Tipos integrados*). Los diccionarios se encuentran a veces en otros lenguajes como "memorias asociativas" o "arreglos asociativos". A diferencia de las secuencias, que se indexan mediante un rango numérico, los diccionarios se indexan con *claves*, que pueden ser cualquier tipo inmutable; las cadenas y números siempre pueden ser claves. Las tuplas pueden usarse como claves si solamente contienen cadenas, números o tuplas; si una tupla contiene cualquier objeto mutable directa o indirectamente, no puede usarse como clave. No podés usar listas como claves, ya que las listas pueden modificarse usando asignación por índice, asignación por sección, o métodos como **`append()`** y **`extend()`**.

Lo mejor es pensar en un diccionario como un conjunto no ordenado de pares *clave: valor*, con el requerimiento de que las claves sean únicas (dentro de un diccionario en particular). Un par de llaves crean un diccionario vacío: `{}`. Colocar una lista de pares clave:valor separados por comas entre las llaves añade pares clave:valor iniciales al diccionario; esta también es la forma en que los diccionarios se presentan en la salida.

Las operaciones principales sobre un diccionario son guardar un valor con una clave y extraer ese valor dada la clave. También es posible borrar un par clave:valor con `del`. Si usás una clave que ya está en uso para guardar un valor, el valor que estaba asociado con esa clave se pierde. Es un error extraer un valor usando una clave no existente.

Hacer `list(d.keys())` en un diccionario devuelve una lista de todas las claves usadas en el diccionario, en un orden arbitrario (si las querés ordenadas, usá en cambio `sorted(d.keys())`. [6] Para controlar si una clave está en el diccionario, usá el **`in`**.

Un pequeño ejemplo de uso de un diccionario:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
```