10 Tuplas

Las tuplas son inmutables

Una tupla[1] es una secuencia de valores muy parecida a una lista. Los valores almacenados en una tupla pueden ser de cualquier tipo, y están indexados por enteros. La diferencia importante es que las tuplas son **inmutables**. Las tuplas también son **comparables** y **hashables**, por lo que podemos ordenar las listas y usar tuplas como valores en los diccionarios de Python.

Sintácticamente, una tupla es una lista de valores separados por comas:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Aunque no es necesario, es común incluir tuplas entre paréntesis para ayudarnos a identificar tuplas rápidamente cuando observamos el código de Python:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Para crear una tupla con un solo elemento, debe incluir la coma final:

```
>>> var_tupla = 'a',
>>> type(var_tupla)
<class 'tuple'>
```

Sin la coma, Python trata `('a')` como una expresión con una cadena entre paréntesis que se evalúa como una cadena:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Otra forma de construir una tupla es la función incorporada `tupla`. Sin argumento, crea una tupla vacía; o usando paréntesis `()`:

```
>>> t2 = ()
>>> type(t2)
<class 'tuple'>
```

Si el argumento es una secuencia (cadena, lista o tupla), el resultado de la llamada a `tuple` es una tupla con los elementos de la secuencia:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Como `tuple` es el nombre de un constructor, debes evitar usarlo como nombre de variable.

La mayoría de los operadores de listas también funcionan con las tuplas. El operador de corchete permite acceder a un elemento:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
a
```

Y el operador de corte selecciona un rango de elementos.

```
>>> print(t[1:3])
('b', 'c')
```

Pero si intentas modificar uno de los elementos de la tupla, obtendrás un error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

No puedes modificar los elementos de una tupla, pero puedes reemplazar una tupla por otra:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t = ('A',) + t
>>> t
('A', 'a', 'b', 'c', 'd', 'e')
```

Comparando tuplas

Los operadores de comparación funcionan en tuplas y otras secuencias. Python comienza comparando el primer elemento de cada secuencia. Si son iguales, continúa con el siguiente elemento, y así sucesivamente, hasta que encuentre elementos que difieran. Los elementos subsiguientes no se consideran (incluso si son realmente grandes).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

La función `sort` funciona de la misma manera. Se ordena principalmente por el primer elemento, pero en el caso de un empate, se ordena por el segundo elemento, y así sucesivamente.

Esta característica se presta a un patrón llamado **DSU**:

Decorate

Una secuencia mediante la creación de una lista de tuplas con una o más claves de clasificación que preceden a los elementos de la secuencia,

Sort

la lista de tuplas usando el `sort` incorporado de Python, y

Undecorate

Extrayendo los elementos ordenados de la secuencia.

[DSU]

Por ejemplo, supón que tienes una lista de palabras y deseas clasificarlas de la más larga a la más corta:

https://trinket.io/embed/python3/84b1d6d4a1

El primer bucle crea una lista de tuplas, donde cada tupla es una palabra precedida por su longitud.

`sort` compara el primer elemento, la longitud, y solo considera el segundo elemento para romper empates. El argumento `reverse=True` le dice a `sort` que vaya en orden decreciente.

El segundo bucle recorre la lista de tuplas y crea una lista de palabras en orden descendente de longitud. Las palabras de cuatro caracteres están ordenadas en orden **alfabético inverso**, por lo que "what" aparece antes de "soft" en la siguiente lista.

La salida del programa es:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

Por supuesto, la línea pierde gran parte de su impacto poético cuando se convierte en una lista de Python y se clasifica en orden de longitud de palabra descendente.

Asignación de tupla

Una de las características sintácticas únicas del lenguaje Python es la capacidad de tener una tupla en el lado izquierdo de una asignación. Esto le permite asignar más de una variable a la vez cuando el lado izquierdo es una secuencia.

En este ejemplo tenemos una lista de dos elementos (que es una secuencia) y asignamos el primer y segundo elementos de la secuencia a las variables `x` y `y` en una sola declaración.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
```

No es magia, Python traduce la sintaxis de asignación de tupla para que sea la siguiente: [2]

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
```

```
>>> y
'fun'
```

Estilísticamente, cuando usamos una tupla en el lado izquierdo de la declaración de asignación, omitimos los paréntesis, pero la siguiente es una sintaxis igualmente válida:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

Una aplicación especialmente inteligente de la asignación de tuplas nos permite **intercambiar** los valores de dos variables en una sola declaración:

```
>>> a, b = b, a
```

Ambos lados de esta declaración son tuplas, pero el lado izquierdo es una tupla de variables. El lado derecho es una tupla de expresiones. Cada valor en el lado derecho se asigna a su variable respectiva en el lado izquierdo. Todas las expresiones en el lado derecho son evaluadas antes de cualquiera de las asignaciones.

El número de variables a la izquierda y el número de valores a la derecha deben ser iguales:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Generalmente, el lado derecho puede ser cualquier tipo de secuencia (cadena, lista o tupla). Por ejemplo, para dividir una dirección de correo electrónico en un nombre de usuario y un dominio, podría escribir:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

El valor de retorno de `split` es una lista con dos elementos; el primer elemento se asigna a `uname`, el segundo a `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

Diccionarios y tuplas

Los diccionarios tienen un método llamado `items` que devuelve una lista de tuplas, donde cada tupla es un par clave-valor:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('a', 10), ('b', 1), ('c', 22)]
```

Como puedes esperar de un diccionario, los elementos no están en ningún orden en particular.

Sin embargo, como la lista de tuplas es una lista y las tuplas son comparables, ahora podemos ordenar la lista de tuplas. Convertir un diccionario en una lista de tuplas es una forma de generar el contenido de un diccionario ordenado por clave:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

La nueva lista se clasifica en orden alfabético ascendente por el valor clave.

Asignación múltiple con diccionarios

Combinando `items`, asignación de tuplas y `for`, puedes ver un bonito patrón de código para recorrer las claves y los valores de un diccionario en un solo bucle:

```
for key, val in list(d.items()):
    print(val, key)
```

Este bucle tiene dos **variables de iteración** porque `items` devuelve una lista de tuplas y `key, val` es una asignación de tuplas que se repite sucesivamente a través de cada uno de los pares clave-valor en el diccionario.

Para cada iteración a través del bucle, tanto `key` como `val` avanzan al siguiente par clave-valor en el diccionario (aún en orden hash).

La salida de este bucle es:

```
10 a
22 c
1 b
```

Nuevamente, está en orden de clave hash (es decir, no hay un orden particular).

Si combinamos estas dos técnicas, podemos imprimir el contenido de un diccionario ordenado por el **valor** almacenado en cada par clave-valor.

Para hacer esto, primero hacemos una lista de tuplas donde cada tupla es `(valor, clave)`. El método `items` nos daría una lista de tuplas `(clave, valor)`, pero esta vez queremos ordenar por valor, no por clave. Una vez que hemos construido la lista con las tuplas de clave de valor, es una cuestión simple ordenar la lista en orden inverso e imprimir la nueva lista ordenada.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

Al construir cuidadosamente la lista de tuplas para que tenga el valor como primer elemento de cada tupla, podemos ordenar la lista de tuplas y obtener el contenido de nuestro diccionario ordenado por valor.

Las palabras más comunes

Volviendo a nuestro ejemplo de ejecución del texto de **Romeo y Julieta** Act 2, Scene 2, podemos aumentar nuestro programa para usar esta técnica para imprimir las diez palabras más comunes en el texto de la siguiente manera:

https://trinket.io/embed/python3/2f13e5b367

La primera parte del programa, que lee el archivo y crea el diccionario que asigna a cada palabra su conteo en el documento, no se modifica. Pero en lugar de simplemente imprimir `count` y finalizar el programa, construimos una lista de tuplas `(val, key)` y luego ordenamos la lista en orden inverso.

Como el valor es primero, se utilizará para las comparaciones. Si hay más de una tupla con el mismo valor, se verá el segundo elemento (la clave), por lo que las tuplas en las que el valor es el mismo se ordenarán según el orden alfabético de la clave.

Al final, escribimos un bonito bucle `for` que realiza una iteración de asignación múltiple e imprime las diez palabras más comunes al recorrer una parte de la lista (`lst[:10]`).

Así que ahora la salida finalmente se parece a lo que queremos para nuestro análisis de frecuencia de palabras.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
```

```
24 thee
```

El hecho de que este análisis de datos complejos se pueda realizar con un programa Python de 19 líneas fácil de entender es una de las razones por las que Python es una buena opción como lenguaje para explorar información.

Uso de tuplas como claves en los diccionarios

Debido a que las tuplas son **hashables** y las listas no, si queremos crear una clave **compuesta** para usar en un diccionario, debemos usar una tupla como clave.

Nos encontraríamos con una clave compuesta si quisiéramos crear un directorio telefónico que se asigne desde los pares de apellidos y nombre a los números de teléfono. Suponiendo que hayamos definido las variables `last`, `first` y `number`, podríamos escribir una declaración de asignación de diccionario de la siguiente manera:

```
directory[last,first] = number
```

La expresión entre corchetes es una tupla. Podríamos usar la asignación de tuplas en un bucle `for` para recorrer este diccionario.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

Este bucle recorre las claves en `directory`, que son tuplas. Asigna los elementos de cada tupla a `last` y `first`, luego imprime el nombre, apellido y el número de teléfono correspondientes.

Secuencias: cadenas, listas y tuplas - ¡Oh My!

Me he centrado en las listas de tuplas, pero casi todos los ejemplos de este capítulo también funcionan con listas de listas, tuplas de tuplas y tuplas de listas. Para evitar enumerar las posibles combinaciones, a veces es más fácil hablar de secuencias de secuencias.

En muchos contextos, los diferentes tipos de secuencias (cadenas, listas y tuplas) se pueden usar indistintamente. Entonces, ¿cómo y por qué eliges uno sobre los otros?

Para comenzar con lo obvio, las cadenas son más limitadas que otras secuencias porque los elementos deben ser caracteres. También son inmutables. Si necesita la capacidad de cambiar los caracteres en una cadena (en lugar de crear una nueva cadena), es posible que desee utilizar una lista de caracteres en su lugar.

Las listas son más comunes que las tuplas, principalmente porque son mutables. Pero hay algunos casos en los que quizás prefieras las tuplas:

1. En algunos contextos, como una declaración `return`, es sintácticamente más simple crear una tupla que una lista. En otros contextos, es posible que prefieras una lista.
2. Si deseas usar una secuencia como clave de diccionario, debes usar un tipo inmutable como una tupla o cadena.
3. Si estás pasando una secuencia como un argumento a una función, el uso de tuplas reduce el potencial de comportamiento inesperado debido al aliasing.

Debido a que las tuplas son inmutables, no proporcionan métodos como `sort` y `reverse`, que modifican las listas existentes. Sin embargo, Python proporciona las funciones integradas `sorted` y `reverseed`, que toman cualquier secuencia como parámetro y devuelven una nueva secuencia con los mismos elementos en un orden diferente.

Depurando

Las listas, los diccionarios y las tuplas se conocen genéricamente como **estructuras de datos**. En este capítulo estamos empezando a ver estructuras de datos compuestas, como listas de tuplas y diccionarios que contienen tuplas como claves y listas como valores. Las estructuras de datos compuestos son útiles, pero son propensas a lo que yo llamo **errores de forma**, es decir, los errores causados cuando una estructura de datos tiene el tipo, tamaño o composición incorrectos. Puede ocurrir que escribas un código, olvides la forma de tus datos y posteriormente introduzcas un error.

Por ejemplo, si estás esperando una lista con un entero y te doy un entero simple (no en una lista), no funcionará.

Cuando estás depurando un programa, y especialmente si estás trabajando en un error, hay cuatro cosas que puedes intentar:

léelo

Examina tu código, léelo de nuevo y verifica que diga lo que querías decir.

ejecútalo

Experimenta haciendo cambios y ejecutando diferentes versiones. A menudo, si muestras lo correcto en el lugar correcto en el programa, el problema se vuelve obvio, pero a veces debes dedicar algo de tiempo a construir andamios.

máscalo

¡Tómate un tiempo para pensar! ¿Qué tipo de error es: sintáctico, en tiempo de ejecución, semántico? ¿Qué información puedes obtener de los mensajes de error o de la salida del programa? ¿Qué tipo de error podría causar el problema que estás viendo? ¿Qué fue lo último que cambiaste, antes de que apareciera el problema?

déjalo

En algún momento, lo mejor que puede hacer es retirarte, deshacer los cambios recientes, hasta que vuelvas a un programa que funcione y que comprendas. Entonces puedes empezar a reconstruir.

Los programadores principiantes a veces se atascan en una de estas tareas y se olvidan de las demás. Cada actividad viene con su propio modo de falla.

Por ejemplo, leer tu código puede ayudar si el problema tiene un error tipográfico, pero no si el problema es un malentendido conceptual. Si no entiendes lo que hace tu programa, puedes leerlo 100 veces y nunca ver el error, porque el error está en tu cabeza.

Realizar experimentos puede ayudar, especialmente si ejecutas pruebas pequeñas y simples. Pero si ejecutas experimentos sin pensar o leer tu código, podrías caer en un patrón que yo llamo "programación aleatoria", que es el proceso de hacer cambios aleatorios hasta que el programa haga lo correcto. No hace falta decir que la programación aleatoria puede llevar mucho tiempo.

Tienes que tomarte tiempo para pensar. La depuración es como una ciencia experimental. Debes tener al menos una hipótesis sobre cuál es el problema. Si hay dos o más posibilidades, trata de pensar en una prueba que elimine una de ellas.

Tomar un descanso ayuda a pensar. También lo hace hablar. Si explicas el problema a otra persona (o incluso a ti mismo), a veces encontrarás la respuesta antes de terminar de formular la pregunta.

Pero incluso las mejores técnicas de depuración fallarán si hay demasiados errores, o si el código que intentas corregir es demasiado grande y complicado. A veces, la mejor opción es retirarte,

simplificando el programa hasta que llegues a algo que funcione y que comprendas.

Los programadores principiantes a menudo son reacios a retirarse porque no pueden soportar eliminar una línea de código (incluso si está mal). Si te hace sentir mejor, copia tu programa en otro archivo antes de comenzar a borrarlo. Luego puedes pegar las piezas de nuevo poco a poco.

Encontrar un error difícil requiere leer, correr, rumiar y, a veces, retirarse. Si te quedas estancado en una de estas actividades, prueba las otras.

Ejercicios

Ejercicio 1: Revisa un programa anterior de la siguiente manera: Lee y analiza las líneas "From" y saca las direcciones de la línea. Cuenta el número de mensajes de cada persona usando un diccionario.

Después de haber leído todos los datos, imprime la persona con el mayor número de confirmaciones creando una lista de tuplas (conteo, correo electrónico) del diccionario. Luego, ordena la lista en orden inverso e imprime a la persona que tenga más envíos.

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Ejercicio 2: Este programa cuenta la distribución de las horas del día para cada uno de los mensajes. Puedes sacar la hora de la línea "From" encontrando la cadena de tiempo y luego dividiendo esa cadena en partes usando el carácter de dos puntos. Una vez que hayas acumulado los conteos para cada hora, imprime los conteos, uno por línea, ordenados por hora como se muestra a continuación.

Ejecución de la muestra:

```
python timeofday.py
Enter a file name: mbox-short.txt
```

```
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Ejercicio 3: Escribe un programa que lea un archivo e imprima las **letras** en orden decreciente de frecuencia. tu programa debe convertir todas las entradas a minúsculas y solo contar las letras a-z. Tu programa no debe contar espacios, dígitos, puntuación o cualquier otra cosa que no sean las letras a-z. Encuentra muestras de texto de varios idiomas diferentes y observa cómo la frecuencia de las letras varía según el idioma. Compara tus resultados con las tablas en wikipedia.org/wiki/Letter_frequencies.

> [1]. Dato curioso: la palabra "tupla" proviene de los nombres que se dan a las secuencias de números de diferentes longitudes: simple, doble, triple, cuádruple, quituple, sextuple, septuple, etc. ↩

> [1]. Python no traduce la sintaxis literalmente. Por ejemplo, si intentas esto con un diccionario, no funcionará como podría esperarse. ↩

Revision #1
Created 5 April 2025 11:11:33 by Javier Quintana
Updated 5 April 2025 11:12:22 by Javier Quintana