Tipos de datos estructurados

Tuplas, conjuntos y diccionarios

Mario González
Facultad de Ingeniería y Ciencias Aplicadas
Universidad de las Américas, Quito, Ecuador

aug-sep, 2019

Tuplas I

- Hasta el momento hemos visto dos tipos compuestos de datos: **strings**, que están constituidas de caracteres, y **listas**, que están constituidas de elementos de cualquier tipo.
- Una de las diferencias entre strings y listas, era que los elementos de estas últimas podían ser modificados, pero los caracteres de un string no.
- En otras palabras, los strings son inmutables y las listas son mutables.
- El tipo de dato **tuple** en python, es similar a una lista con la excepción de que es inmutable.

Tuplas II

- Sintácticamente una tupla es una lista de valores separados por coma.

```
tupla = 'a', 'b', 'c', 'd', 'e'
tupla = ('a', 'b', 'c', 'd', 'e')
t1 = ('a',)
type(t1)
<type 'tuple'>
```

- Dejando de un lado la sintaxis, las operaciones en las tuplas son las mismas operaciones que para las listas.

```
tupla = ('a', 'b', 'c', 'd', 'e')
tupla[0]
'a'
tuple[1:3]
('b', 'c')
```

Tuplas III

- Si tratamos de modificar un elemento de la tupla, ocurre un error.
- Siempre podemos remplazar la tupla por una diferente:

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

tupla = ('A',) + tuple[1:]
tupla
('A', 'b', 'c', 'd', 'e')
```

Tuplas IV

Asignación de tuplas

- A veces es útil intercambiar los valores de dos variables. Python provee una forma de **asignación de tupla** que nos ayuda en esta tarea:

```
a, b = b, a

a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

- El lado izquierdo es una tupla de variables y el derecho una tupla de valores. Todas las expresiones en el lado derecho son evaluadas antes de ser asignadas.
- El número de variables en la izquierda y el número de valores en la derecha deben coincidir.

Tuplas V

Tuplas como valores de retorno

- Las funciones pueden devolver tuplas como valores de retorno.

```
def minmax(lista):
    valMax=lista[0]
    valMin=lista[0]
    for i in lista:
        if i>valMax:
            valMax=i
        if i<valMin:
            valMin=i

    return valMin,valMax,valMax-valMin
```

Tuplas VI

Cuándo utilizar tuplas

- Cuando se necesita almacenar datos que no tienen que cambiar.
- Cuando el rendimiento de la aplicación es muy importante. En esta situación se puede utilizar tuplas cuando los valores son colecciones de datos fijos.
- Cuando se desea almacenar datos en pares inmutables, triples, etc.

Diccionarios I

- Los datos compuestos que hemos visto hasta el momento (strings, listas, tuplas) usan enteros como índices. Si intentamos usar otro tipo como índice obtenemos un error.
- **Los diccionarios** están representados por un par `clave:valor`.
- En otras palabras, son mapas o colecciones asociativas.
- Las claves (índices), a diferencia de las listas donde deben ser numéricos, pueden ser de cualquier tipo inmutable y deben ser únicos.
- Los valores pueden ser de cualquier tipo, mutable o inmutable.

Diccionarios II

- Una forma de crear un diccionario es empezar con un diccionario vacío ({}) y añadir elementos.

```
eng2sp = {}
eng2sp['one'] = 'uno'
eng2sp['two'] = 'dos'
print(eng2sp)
{'one': 'uno', 'two': 'dos'}
```

- Otra forma de crear un diccionario es:

```
eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
print(eng2sp)
{'three': 'tres', 'two': 'dos', 'one': 'uno'}
```

- Los pares **`clave:valor`** no están en orden. No debemos preocuparnos por el orden, dado que los elementos no están indexados por enteros sino por la clave:

```
eng2sp['one']
'uno'
```

Diccionarios III

Operaciones con diccionarios

- La declaración **`del`** elimina un par clave-valor de un diccionario. Por ejemplo, el siguiente diccionario contiene los nombres de varias frutas y el número de cada fruta en el almacén:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
'pears': 217}
print(inventory)
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}
```

- Si alguien compra todas las peras podemos remover la entrada del diccionario:

```
del inventory['pears']
print(inventory)
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

Diccionarios IV

- O si estamos esperando que lleguen más peras, podemos solamente cambiar el valor asociado con peras:

```
inventory['pears'] = 0
print(inventory)
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}

len(inventory)
4
```

- La función `len` también funciona con los diccionarios, devuelve el numero de pares clave-valor.

Diccionarios V

Métodos

- Un método es similar a una función, toma argumentos de entrada y devuelve un valor, pero la sintaxis es diferente.
- Por ejemplo el método **`keys`** toma un diccionario y devuelve una lista de claves que aparecen en este.
- En lugar de una sintaxis de función **`keys(eng2sp)`**, usamos la sintaxis del método **`eng2sp.keys()`**.

```
eng2sp.keys()
['one', 'three', 'two']
```

- La llamada de un método se conoce como **invocación**, en este caso hemos invocado el método `keys` sobre el objeto `eng2sp`. Los paréntesis vacíos indican que el método no tiene parámetros.

Diccionarios VI

- Otros métodos del diccionario son:

```
eng2sp.values()
['uno', 'tres', 'dos']

eng2sp.items()
[('one','uno'), ('three', 'tres'), ('two', 'dos')]

'pears' in inventory.keys()
True

'deux' in eng2sp.keys()
False
```

Diccionarios VII

Aliasing y Copias

- Dado que los diccionarios son mutables, necesitamos tener cuidado con el aliasing. Cuando dos variables se refieren al mismo objeto, los cambios en una afectan al otro.
- Si queremos modificar un diccionario y mantener una copia del original, debemos usar el método **`copy`**.

```
opuestos={'up': 'down', 'right': 'wrong', 'true': 'false'}
alias=opuestos
copia=opuestos.copy()
```

- `alias` y `opuestos` se refieren al mismo objeto, si modificamos `alias`, `opuestos` también cambia.
- `copia` se refiere a una copia *fresca* del diccionario. Si modificamos `copia`, `opuestos` no cambia.

Diccionarios VIII

```
alias['right'] = 'left'
opuestos['right']
'left'

copia['right'] = 'privilege'
opuestos['right']
'left'
```

Diccionarios IX

Matrices dispersas

- Podemos usar una lista anidada para representar una matriz. Es una buena opción si la matriz no es dispersa.
- Una matriz dispersa es aquella en la que la mayoría de sus elementos son cero.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

```
matrix = [ [0,0,0,1,0],
           [0,0,0,0,0],
           [0,2,0,0,0],
           [0,0,0,0,0],
           [0,0,0,3,0] ]

matrix[0][3]
1
```

Diccionarios X

- Una alternativa es usar un diccionario. Como claves utilizamos una tupla que contenga la fila y columna, por ejemplo:

```
matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
matrix[0,3]
1
matrix[1,3]
KeyError: (1, 3)
```

- Note que la sintaxis para acceder a un elemento del diccionario no es la misma que para una lista anidada. En lugar de enteros entre corchetes [][], utilizamos un índice que es una tupla.
- Si especificamos un elemento que es cero, obtenemos un error, porque no hay una entrada en el diccionario que corresponda a dicha clave.

Diccionarios XI

- El método **`get`** resuelve este problema:

```
matrix.get((0,3), 0)
1
matrix.get((1,3), 0)
0
```

- El primer argumento es la clave, el segundo argumento es el valor que `get` devuelve si la clave no está en el diccionario.

Conjuntos (sets) I

- Un conjunto es una colección desordenada de objetos, a diferencia de secuencias de objetos tales como listas y tuplas, en los que se indexa cada elemento.
- Los conjuntos no pueden tener miembros duplicados - un objeto dado aparece en un conjunto 0 o 1 veces.
- Todos los miembros de un conjunto tienen que ser *hashable*, como las claves de un diccionario.
- Números enteros, números de coma flotante, tuplas, y las cadenas (strings) son hashable.
- Diccionarios, listas y otros conjuntos (excepto frozensets) no lo son.

Conjuntos (sets) II

- Un objeto es **hashable** si tiene un valor hash que nunca cambia durante su vida útil.
- Hashability hace que un objeto utilizable como clave de diccionario y un miembro de conjunto, porque estas estructuras de datos utilizan el valor hash internamente. Todos inmutables objetos de Python son hashable, pero no los contenedores mutables (como listas o diccionarios).

Conjuntos (sets) III

Cómo construir un conjunto

- El constructor `set` acepta cualquier objeto secuencial como argumento:

```
set([0, 1, 2, 3])
set([0, 1, 1, 2, 3, 3])
set("obtuse")
set(['b', 'e', 'o', 's', 'u', 't'])

s = {1, 2, 3}
s
```

- El consctructor $\{\}$ también puede ser utilizado.

Conjuntos (sets) IV

- También podemos agregar elementos a un conjunto uno a uno utilizando la función `add`.
- Si agregamos un elemento que existe en el conjunto, la función `add` no surte efecto.
- Ocurre lo mismo con la función `update`, que agrega un grupo de elementos a un conjunto.

```
s = set([12, 26, 54])
s.add(32)
s
set([32, 26, 12, 54])

s.update([26, 12, 9, 14])
s
set([32, 9, 12, 14, 54, 26])
```

Conjuntos (sets) V

- Podemos realizar una copia del conjunto con `copy`

```
s2 = s.copy()
s2
set([32, 9, 12, 14, 54, 26])
```

Pertenencia a un conjunto

- Podemos comprobar si un objeto está en el conjunto usando el operador `in`.

```
32 in s
True
6 in s
False
6 not in s
True
```

Conjuntos (sets) VI

- También podemos comprobar la pertenencia de conjuntos enteros, comprobando si son sub o super conjuntos:

```
s.issubset(set([32, 8, 9, 12, 14, -4, 54, 26, 19]))
True
s.issuperset(set([9, 12]))
True
s.issuperset([32, 9])
True
```

- Note que los operadores $<=$ y $>=$ son equivalentes a las funciones `issubset` y `issuperset` respectivamente.

```
set([4, 5, 7]) <= set([4, 5, 7, 9])
True
set([9, 12, 15]) >= set([9, 12])
True
```

Conjuntos (sets) VII

Eliminando elementos de un conjunto

- Hay tres funciones que eliminan elementos individuales de un conjunto `pop`, `remove` y `discard`.
- `pop`, simplemente elimina un elemento del conjunto. Tenga en cuenta que no hay ningún comportamiento que defina como se elige el elemento a eliminar.

```
s = set([1,2,3,4,5,6])
s.pop()
1
s
set([2,3,4,5,6])
```

Conjuntos (sets) VIII

- La función `remove` elimina un elemento específico del conjunto:

```
s.remove(3)
s
set([2,4,5,6])
s.remove(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 9
```

- Eliminar un elemento que no es parte del conjunto devuelve un error. Para evitar esto se puede utilizar la función `discard`.
- La función `clear` elimina todos los elementos de un conjunto:

```
s.clear()
s
set([])
```

Conjuntos (sets) IX

Iteración sobre un conjunto

- Podemos usar un bucle para iterar por los elementos de un conjunto. Dado que los conjuntos son desordenados, no está definido el orden que seguirá la iteración:

```
s = set("blerg")
for n in s:
...     print(n, end=" ")
...
r b e l g
```

Conjuntos (sets) X

Operaciones de conjuntos

- **Intersección:** Todo elemento que pertenezca a S_1 y S_2 aparecerá en su intersección.

```
s1 = set([4, 6, 9])
s2 = set([1, 6, 8])
s1.intersection(s2)
set([6])
s1 & s2
set([6])
s1.intersection_update(s2)
s1
set([6])
```

Conjuntos (sets) XI

- **Unión:** es la *fusión* de dos conjuntos. Todo elemento que pertenezca a S_1 o S_2 aparecerá en su unión.

```
s1 = set([4, 6, 9])
s2 = set([1, 6, 8])
s1.union(s2)
set([1, 4, 6, 8, 9])
s1 | s2
set([1, 4, 6, 8, 9])
```

Conjuntos (sets) XII

- La **diferencia simétrica** de dos conjuntos es el conjunto de elementos que están en uno u otro conjunto pero no en ambos.

```
s1 = set([4, 6, 9])
s2 = set([1, 6, 8])
s1.symmetric_difference(s2)
set([8, 1, 4, 9])
s1 ^ s2
set([8, 1, 4, 9])
s1.symmetric_difference_update(s2)
s1
set([8, 1, 4, 9])
```

Conjuntos (sets) XIII

- Python también puede encontrar **la diferencia** de S_1 y S_2, que son todos los elementos que están en S_1 pero no en S_2.

```
s1 = set([4, 6, 9])
s2 = set([1, 6, 8])
s1.difference(s2)
set([9, 4])
s1 - s2
set([9, 4])
s1.difference_update(s2)
s1
set([9, 4])
```

Conjuntos (sets) XIV

Múltiples conjuntos

- La unión, intersección y la diferencia funcionan con múltiples conjuntos usando el constructor `set`. Por ejemplo, usando `set.intersection()`:

```
s1 = set([3, 6, 7, 9])
s2 = set([6, 7, 9, 10])
s3 = set([7, 9, 10, 11])
set.intersection(s1, s2, s3)
set([9, 7])
```

Conjuntos (sets) XV

`frozenset`

- Un `frozenset` es básicamente lo mismo que un conjunto, excepto que es inmutable, una vez que se crea, sus elementos no se pueden cambiar.
- Puesto que son inmutables, también son hashable, lo que significa que los frozensets pueden utilizarse como miembros de otros conjuntos y claves como diccionarios.
- Los frozensets tienen las mismas funciones que los conjuntos normales, excepto las funciones que cambian el contenido (update, remove, pop, etc.) están disponibles.

Conjuntos (sets) XVI

```
fs = frozenset([2, 3, 4])
s1 = set([fs, 4, 5, 6])
s1
set([4, frozenset([2, 3, 4]), 6, 5])
fs.intersection(s1)
frozenset([4])
fs.add(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```