Programación II

Estructuras de Datos

Lic. Federico Severino Guimpel
LCC - FCEIA - UNR

Estructuras de Datos

Una estructura de datos es una forma particular de representar y de organizar información. Existen diferentes estructuras de datos, cada una con características particulares por lo que no todas son adecuadas para resolver cualquier problema.

Por ejemplo, en este momento, en la materia, vimos dos estructuras de datos. ¿Se dan cuenta cuáles son?

Estructuras de Datos

Exactamente! Estoy pensando en Listas y Tuplas. Ambas son estructuras de datos.

Aparte de ellas, ¿existirán otras en Python? Y, si la respuesta es afirmativa, ¿qué características tendrán?

Esta última pregunta surge del hecho de que Listas y Tuplas son diferentes.

Ejercicio: Pensar 3 características que difieran entre Listas y Tuplas.

Diccionarios

La primera estructura de datos que vamos a mostrar se llama Diccionario. La misma tiene ciertas características que la distinguen de las otras dos conocidas.

Vamos a comenzar viendo, con ejemplos, cómo se define y usa un diccionario.

```
>>> a = {}
>>> print(a)
{}
>>> type(a)
<class 'dict'>
```

Diccionarios

Se puede ver que para definir un diccionario debemos usar {}. Ahora bien, ¿cómo hacemos para crear un diccionario con elementos?

```
>>> a = {'juan':3415555555, 'luis':3416666666}
>>> print(a)
{'juan': 3415555555, 'luis': 3416666666}
```

Si, al igual que en las listas y tuplas, los elementos se separan por comas entonces este diccionario contiene dos elementos.

Diccionarios

Analicemos cada elemento que forma parte de este diccionario. Tratemos de acceder al primer elemento del diccionario usando la misma forma con la que accedemos a listas y tuplas, es decir usando [ ] y la posición.

```
>>> a = {'juan':3415555555, 'luis':3416666666}
>>> a[0]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    a[0]
KeyError: 0
>>> a[1]
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    a[1]
KeyError: 1
```

Se puede ver que no funciona de esta manera. ¿Será que no se pueden usar [] para acceder a sus elementos o, por el contrario, será que no se pueden usar posiciones para esto?

Analicemos el siguiente ejemplo:

```
>>> a = {'juan':3415555555, 'luis':3416666666}
>>> print(a)
{'juan': 3415555555, 'luis': 3416666666}
>>> a['juan']
3415555555
```

Diccionarios

Con el ejemplo anterior se puede confirmar que no se usan posiciones para acceder a los elementos de un Diccionario.

Esto es porque un Diccionario es una estructura de datos no indexada por posición sino por clave.

```
>>> a = {'juan':3415555555, 'luis':3416666666}
>>> print(a)
{'juan': 3415555555, 'luis': 3416666666}
>>> a['juan']
3415555555
```

Diccionarios

Revisemos la declaración del Diccionario:

```
>>> a = {'juan':3415555555, 'luis':3416666666}
```

En este caso, las claves son strings y los datos asociados son números.

Un Diccionario, en Python, recibe dicho nombre porque dada una clave devuelve el valor asociado. Cada clave debe ser única dentro del diccionario.

```
>>> a = {1: 'a', 1: 'b'}
>>> print(a)
{1: 'b'}
```

Como se puede ver, en caso de querer guardar datos con la misma clave simplemente se guarda el último valor dado.

Diccionarios

A partir de esto, podemos pensar varias preguntas:

1. ¿qué se podrá usar como clave?

Comencemos con esto. Se puede usar como clave cualquier tipo de dato que no pueda cambiar su valor, es decir, que sea inmutable.

Se han visto ejemplos donde la clave fue numérica o strings. Las tuplas también son un tipo inmutable por lo que pueden ser usadas como clave.

```
>>> a = {(1,2):'hola'}
>>> a[(1,2)]
'hola'
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

1. ¿qué se podrá usar como clave?

¿Qué pasaría si tratamos de usar un tipo no inmutable, es decir mutable, como clave? Hagamos la prueba:

```
>>> a = {[1,2]:'hola'}
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    a = {[1,2]:'hola'}
TypeError: unhashable type: 'list'
```

Pasemos ahora a otra pregunta.

Diccionarios

A partir de esto, pueden surgir varias preguntas:

2. ¿qué se podrá usar como valor?

Se puede guardar como valor cualquier tipo de dato. En el siguiente ejemplo se muestra un diccionario donde se guardan tuplas asociadas a un valor numérico (el DNI por ejemplo).

```
>>> a = {6035051: ('federico', 71, 'Entre Rios 1634'), 8640181: ('maria', 65, 'Alem 1017')}
>>> a[6035051]
('federico', 71, 'Entre Rios 1634')
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

2. ¿qué se podrá usar como valor?

Acá vemos un Diccionario que tiene clave numérica y listas como valores.

```
>>> a = {1:[1, 'a'], 2:[]}
>>> a[1]
[1, 'a']
>>> a[2]
[]
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

3. ¿pueden ser de distinto tipo las claves y los valores?

La respuesta es que sí. Podemos verlo en el siguiente ejemplo:

```
>>> a = {1:['hola'], 'a': (1,2,3)}
>>> a[1]
['hola']
>>> a['a']
(1, 2, 3)
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

4. ¿qué sucede si quiero acceder a una clave no existente?

En este caso, lo que sucede es que se produce un error en la ejecución.

```
>>> a = {6035051: ('federico', 71, 'Entre Rios 1634'), 8640181: ('maria', 65, 'Alem 1017')}
>>> a[6035051]
('federico', 71, 'Entre Rios 1634')

>>> a[6035057]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    a[6035057]
KeyError: 6035057
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

5. ¿cómo se pueden modificar o agregar valores a un Diccionario?

Esto se puede hacer directamente a través de las claves.

```
>>> a = {1:['hola'], 'a': (1,2,3)}
>>> a[2]= 'abc'
>>> print(a)
{1: ['hola'], 'a': (1, 2, 3), 2: 'abc'}
>>> a[1]= a[1]+['chau']
>>> print(a)
{1: ['hola', 'chau'], 'a': (1, 2, 3), 2: 'abc'}
```

Aquí se puede ver cómo agregamos un nuevo valor y, cómo se modifica uno existente.

Diccionarios

A partir de esto, pueden surgir varias preguntas:

6. ¿cómo se puede recorrer el Diccionario?

Hay varias formas de hacerlo pero, ante todo, hagamos la prueba de hacer un for como hacíamos sobre listas o tuplas para ver qué resultado se obtiene.

```
>>> a = {1:['hola'], 'a': (1,2,3)}
>>> for x in a:
        print(x)


1
a
```

Se puede ver que lo que se muestra son las claves del Diccionario.

Diccionarios

A partir de esto, pueden surgir varias preguntas:

6. ¿cómo se puede recorrer el Diccionario?

Entonces, para recorrer un Diccionario y mostrar los valores se podría hacer:

```
>>> a = {1:['hola'], 'a': (1,2,3)}
>>> for x in a:
        print(a[x])


['hola']
(1, 2, 3)
```

Usamos el for para recorrer sobre las claves y accedemos a cada valor a través de ellas.

Diccionarios

A partir de esto, pueden surgir varias preguntas:

6. ¿cómo se puede recorrer el Diccionario?

También, el Diccionario cuenta con un método llamado items(). El mismo devuelve una secuencia de pares (clave, valor).

```
>>> a = {1:['hola'], 'a': (1,2,3)}
>>> a.items()
dict_items([(1, ['hola']), ('a', (1, 2, 3))])
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

6. ¿cómo se puede recorrer el Diccionario?

Se puede usar el resultado de dicho método para iterar sobre él en el for.

```
>>> for x, y in a.items():
        print(x, y)

1 ['hola']
a (1, 2, 3)
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

7. ¿qué otros métodos brinda un Diccionario?

Contiene varios pero, vamos a destacar tres:

a. dict() crea un diccionario tomando una lista de pares (clave, valor):

```
>>> l = [(1, 'hola'), ('a', [1, 2]), (3, ('a', 'b'))]
>>> a = dict(l)
>>> print(a)
{1: 'hola', 'a': [1, 2], 3: ('a', 'b')}
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

7. ¿qué otros métodos brinda un Diccionario?

b. keys() devuelve las claves del diccionario

```
>>> print(a)
{1: 'hola', 'a': [1, 2], 3: ('a', 'b')}
>>> a.keys()
dict_keys([1, 'a', 3])
```

Diccionarios

A partir de esto, pueden surgir varias preguntas:

7. ¿qué otros métodos brinda un Diccionario?

c. values() devuelve los valores del diccionario

```
>>> print(a)
{1: 'hola', 'a': [1, 2], 3: ('a', 'b')}
>>> a.values()
dict_values(['hola', [1, 2], ('a', 'b')])
```

Conjuntos

Algo que puede llamar la atención, por uso y costumbre, es que esta declaración:

```
>>> a = {}
>>> print(a)
{}
>>> type(a)
<class 'dict'>
```

Dé lugar a un Diccionario y no a un Conjunto. ¿Existirán Conjuntos en Python?

La respuesta es que sí. A continuación veremos cómo definir y usar Conjuntos.

Conjuntos

Ante todo, si se quiere definir un Conjunto vacío, hay que hacer lo siguiente:

```
>>> a = set()
>>> print(a)
set()
>>> type(a)
<class 'set'>
```

Para definir un Conjunto con elementos, podemos enumerarlos.

```
>>> a = {1, 'hola', (1,2)}
>>> print(a)
{(1, 2), 1, 'hola'}
>>> type(a)
<class 'set'>
```

Conjuntos

Los Conjuntos, en Python, respetan las características usuales. Es decir, no importan los elementos repetidos:

```
>>> a = {1, 1, 1, 1, 2}
>>> print(a)
{1, 2}
```

y, no importa el orden:

```
>>> a == {2, 1}
True
```

Conjuntos

En los Diccionarios nos encontramos con la limitación del tipo de dato que se puede utilizar como clave. Entonces, nos podríamos preguntar: ¿se puede guardar cualquier tipo de dato en un Conjunto?

La respuesta, lamentablemente, es que no. No se pueden almacenar elementos mutables, por ejemplo, listas o diccionarios.

```
>>> a = {1, 'a', [1, 2, 3]}
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    a = {1, 'a', [1, 2, 3]}
TypeError: unhashable type: 'list'
```

Conjuntos

Para iterar sobre los elementos de un Conjunto se puede usar el for tal como se hacía sobre listas:

```
>>> for x in a:
        print(x)


(1, 2)
1
hola
```

Ahora que ya sabemos definir un conjunto e iterar sobre él vamos a ver cómo modificarlo.

Conjuntos

Para modificar un conjunto se cuenta con algunas funciones.

a. La función add permite agregar un elemento a un conjunto.

```
>>> a = {1, 'hola', (1, 2)}
>>> a.add(17)
>>> print(a)
{(1, 2), 1, 'hola', 17}
```

Conjuntos

Para modificar un conjunto se cuenta con algunas funciones.

b. La función update permite agregar permite agregar los elementos de una lista o conjunto a un conjunto.

```
>>> a.update([2, 3, 4, 17])
>>> print(a)
{(1, 2), 1, 2, 3, 4, 17, 'hola'}
```

Conjuntos

Para modificar un conjunto se cuenta con algunas funciones.

c. La función discard permite eliminar un elemento particular del conjunto. Si el mismo no está presente no genera cambios.

```
>>> print(a)
{(1, 2), 1, 2, 3, 4, 17, 'hola'}
>>> a.discard(1)
>>> print(a)
{(1, 2), 2, 3, 4, 17, 'hola'}
>>> a.discard(5)
>>> print(a)
{(1, 2), 2, 3, 4, 17, 'hola'}
```

Conjuntos

Para modificar un conjunto se cuenta con algunas funciones.

d. La función remove es similar a discard. La única diferencia radica en que si se intenta eliminar un elemento que no está presente genera una excepción.

```
>>> print(a)
{(1, 2), 1, 2, 3, 4, 17, 'hola'}
>>> a.remove(1)
>>> print(a)
{(1, 2), 2, 3, 4, 17, 'hola'}
>>> a.remove(5)
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    a.remove(5)
KeyError: 5
```

Conjuntos

Una duda que nos puede surgir (debería) es si contamos con las operaciones típicas de conjuntos. La respuesta es que sí, la mayoría de ellas.

A continuación vamos a ver la sintaxis y un ejemplo de uso de cada una de ellas.

Conjuntos

1. Unión

La unión se realiza a través del operador |. Se retorna el conjunto resultante de la operación.

```
>>> a = {1, 3, 5, 'hola'}
>>> b = {2, 3, 1, 'chau'}
>>> print(a | b)
{1, 2, 3, 5, 'hola', 'chau'}
```

También se puede usar el método unión.

```
>>> a.union(b)
{1, 2, 3, 5, 'hola', 'chau'}
```

2. Intersección

La unión se realiza a través del operador &. Se retorna el conjunto resultante de la operación.

```
>>> a = {1, 3, 5, 'hola'}
>>> b = {2, 3, 1, 'chau'}
>>> print(a&b)
{1, 3}
```

También se puede usar el método intersection.

```
>>> a.intersection(b)
{1, 3}
```

3. Diferencia

La diferencia se realiza a través del operador -. Se retorna el conjunto resultante de la operación.

```
>>> a = {1, 3, 5, 'hola'}
>>> b = {2, 3, 1, 'chau'}
>>> print(a-b)
{'hola', 5}
```

También se puede usar el método difference.

```
>>> a.difference(b)
{'hola', 5}
```