Programando con Python

Estructuras de datos

Estructuras de datos

- Son *estructuras* que mantienen algunos *datos* a la vez. En otras palabras, son usadas para almacenar una colección de datos relacionados.
- Hay cuatro estructuras de datos incorporadas en Python:
  - list
  - Tuple
  - set
  - diccionary

Estructuras de datos - List

- Es una estructura que mantiene una colección ordenada de elementos, por ejemplo, se puede almacenar una *secuencia* de elementos en una list
- Su formato viene dado por una secuencia de elementos separados por coma, entre corchetes:

  variable = [item1, item2, item3, …, itemN]

- Una vez creada una list, se pueden agregar, remover o buscar entre sus elementos. De allí que se diga que es un tipo de datos *mutable*

Estructuras de datos - List

- Métodos de la clase list:

Nombre(args)	Qué hace
list.**append**(x)	Agrega un elemento al final de la list
list.**extend**(*iterable*)	Extiende la list agregando todos los elementos del iterable
list.**insert**(*i*, x)	Inserta un elemento *i* en la posición x dada
list.**remove**(x)	Remueve el primer elemento de la list cuyo valor es x
list.**pop**([*i*])	Remueve y retorna el elemento *i* especificado. Si este no es declarado, remueve y retorna el último elemento de la list
list.**clear**()	Remueve todos los elementos de la list
list.**index**(x [, start[, end]])	Retorna un índice sobre base cero del primer elemento en la list cuyo valor es igual a x. Los argumentos opcionales *start* y *end* son interpretados como en la notación de un slide de la list y son usados para buscar una subsecuencia particular de la list
list.**count**(x)	Retorna el número de veces que x aparece en la list

Estructuras de datos - List

- Métodos de la clase list:

Nombre(args)	Qué hace
list.**sort**(*key = None, reverse = False*)	Ordena los elementos de la lista
list.**reverse**()	Invierte el orden de los elementos de la list
list.**copy**()	Retorna una copia de la list

- Funciones para trabajar con list:

Nombre(args)	Qué hace
len*(list)*	Retorna el número de elementos de la list
sum*(list)*	Retorna la suma de los elementos de la list
max*(list)*	Retorna el elemento de mayor valor en la list
min*(list)*	Retorna el elemento de menor valor en la list

Estructuras de datos - List

- Ejemplos:

```
>>> frutas = ['naranja', 'manzana', 'pera', 'cambur', 'kiwi', 'manzana', 'cambur']
>>> frutas.count('manzana')
2
>>> frutas.count('mandarina')
0
>>> frutas.index('cambur')
3
>>> frutas.index('cambur', 4)
6
```

Estructuras de datos - List

- Ejemplos:

```
>>> frutas.reverse()
>>> frutas
['cambur', 'manzana', 'kiwi', 'cambur', 'pera', 'manzana', 'naranja']
>>> frutas.append('fresa')
>>> frutas
['cambur', 'manzana', 'kiwi', 'cambur', 'pera', 'manzana', 'naranja', 'fresa']
>>> frutas.sort()
>>> frutas
['cambur', 'cambur', 'fresa', 'kiwi', 'manzana', 'manzana', 'naranja', 'pera']
>>> frutas.pop()
'pera'
```

Estructuras de datos - List

- Lists usadas como **stacks**:
  - Los métodos de list permiten usar una lista como un stack (pila), donde el último elemento es agregado es el primero retribuido (LIFO):

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Estructuras de datos - List

- Lists usadas como **queues (colas)**:
  - También podemos implementar una queue donde el primer elemento ingresado es el primer elemento que sale (FIFO)
  - Si bien los métodos appends y pops se ejecutan rápido en una list, son ineficientes para crear la queue, ya que insertar o extraer del principio ocasiona que haya que correr todos los elementos de su posición.
  - Para implementar una queue, se usa **collections.deque**, la cual se diseñó para hacer appends y pops de cualquier extremo, de una forma rápida

Estructuras de datos - List

- Lists usadas como **queues (colas)**, ejemplo:

```
>>> from collections import deque
>>> queue = deque(["Juan", "María", "Alberto"])
>>> queue.append("Rosa")
>>> queue.append("José")
>>> queue.popleft()
'Juan'
>>> queue.popleft()
'María'
>>> queue
deque(['Alberto', 'Rosa', 'José']) ← mantiene el orden de llegada
>>>
```

Estructuras de datos - List

- Creando una list usando la función **range()**
  - La función range() retorna un objeto *iterable*
  - Simplemente se pasa el objeto a la list
  - Ejemplo1:

```
>>> list1 = list(range(8))
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7]
```

Estructuras de datos - List

- Creando una list usando la función **range()**
  - Ejemplo2:

```
>>> list2 = list(range(20, 35))
>>> list2
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]
```

  - Ejemplo 3:

```
>>> list3 = list(range(30, 60, 3))
>>> list3
[30, 33, 36, 39, 42, 45, 48, 51, 54, 57]
```

Estructuras de datos - List

- Las lists son **mutables**. Podemos modificarla sin crear un nuevo objeto
- Ejemplo:

```
>>> lista = ["carro", "pelota", "niño", "casa"]
>>> id(lista)
50947536
>>> lista[1] = "animal"
>>> lista
['carro', 'animal', 'niño', 'casa'] ← cambió el índice 1
>>> id(lista)
50947536 ← pero el objeto sigue siendo el mismo
```

Estructuras de datos - List

- Iterando los elemento de una list
- Se usan los bucles: for

```
>>> marcas = [10, 15, 33, 95, 44]
>>> for m in marcas:
...     print(m)
...
10
15
33
95
44
>>>
```

Estructuras de datos - List

- Iterando los elemento de una list
- Se usan los bucles: while

```
>>> marcas = [10, 15, 33, 95, 44]
>>> i = 0
>>> while i < len(marcas):
...     print(marcas[i])
...     i += 1
...
10
15
33
95
44
```

Estructuras de datos - List

- Al igual que con strings, con la list podemos:
  - Hacer **slicing**: lista[start: end]
  - Aplicar el operador **in** y **not in**
  - Concatenar: listC + listD
  - Aplicar el operador de repetición: listA = listB * 3

Estructuras de datos - List

- **Comprensión**: Cuando se desea crear una list donde cada elemento es el resultado de una operación, o cuando cumple con alguna condición
- Sintaxis: [expresión **for** elemento **in** iterable]
- Ejemplo: se desea obtener los cubos de una secuencia de 2 a 7:

```
>>> cubos = [ i ** 3 for i in range(1,8)]
>>> cubos
[1, 8, 27, 64, 125, 216, 343]
```

Estructuras de datos - List

- También se puede colocar una condición **if**
- Sintaxis: [expresión **for** elemento **in** iterable **if** condición]
- Ejemplo: se desea obtener los cubos de una secuencia de 2 a 7:

```
>>> cubos = [ i ** 3 for i in range(1,8) if i % 2 == 0]
>>> cubos
[8, 64, 216]
```

Estructuras de datos - Tuples

- Tuples
  - Forma parte de la familia de tipos de datos llamados secuencias, al igual que las **list** y el **range**
  - Al igual que las list, son una agrupación de datos separados por coma, pero encerrados entre paréntesis

```
>>>  t = 1, 3, 5, 7, 9
>>> t
>>>(1, 3, 5, 7, 9)
```

  - Son inmutables
  - Pero pueden contener múltiples objetos
  - Se usan en situaciones y propósitos diferentes de list

Estructuras de datos - Tuples

- Operaciones con Tuples
  - Se puede tener acceso a un elemento o hacer slicing con el operador **[]**
  - Las operaciones incluidas **min()**, **max()**, **sum()** son válidas
  - Los operadores de membresía **in** o **not in**
  - Los operadores de comparación
  - Los operadores **+** y *****
  - Hacer iteraciones con **for** o **while**

Estructuras de datos - Sets

- Sets
  - Es una colección de elementos sin orden que no permite elementos repetidos
  - Los datos se encuentran encerrados entre llaves "{ }"

    ```
    >>>  s = {1, 3, 5, 7, 9}
    ```

  - Son mutables, para agregar elementos se usa el método **add()**
  - Permiten eliminar elementos usando los métodos **remove()** y **discard()**. La diferencia radica en que discard() no genera error si el elemento no existe
  - No se puede tener acceso a sus elemento usando un índice

Estructuras de datos - Sets

- Sets – métodos (algunos):

Método	Descripción
add()	Agrega un elemento al set
clear()	Remueve todos los elementos del set
copy()	Retorna una copia del set
difference()	Retorna un set con la diferencia entre uno o más sets
discard()	Remueve el elemento especificado
intersection()	Retorna un set que es la intersección de otros dos sets
issubset()	Retorna un boolean indicando si otro set contiene a este
issuperset()	Retorna un boolean indicando si este set contiene a otro
pop()	Remueve el elemento indicado
update()	Actualiza el set con la unión de este y otros sets

Estructuras de datos - Dictionarios

- Dictionarios
  - Son conjuntos de datos agrupados en pares clave-valor
  - A diferencia de las secuencias los diccionarios son indexados por claves
  - La claves pueden ser de cualquier tipo inmutable: strings, números o tuples. SI un tuple contiene un objeto mutable, directo o indirecto, no puede ser usado como clave
  - Las claves deben ser únicas
  - Se usan {} para representarlos, de la manera *{clave: valor}*
  - Las principales operaciones de un dictionary es almacenar un valor con una clave y luego extraer el valor usando la clave.

Estructuras de datos - Dictionarios

- Dictionarios
  - Pueden ser cambiados, no son inmutables
  - Para eliminar un par *clave:valor* se usa **del**

```
>>> dic = {'juan': 1234, 'maria': 5678, 'rosa': 2468}
>>> del dic['maria']
>>> dic
{'juan': 1234, 'rosa': 2468}
```

  - **list(dic)**, retorna una lista de todas las claves del diccionario
  - El constructor **dict()**, crea diccionarios desde secuencias clave:valor

Estructuras de datos - Dictionaries

- Operaciones con dictionaries (algunas)

Método	Descripción
clear()	Remueve todos los elementos del diccionario
copy()	Retorna una copia del diccionario
fromkeys()	Retorna un diccionario con las claves y valores especificados
get()	Retorna el valor especificado por la clave
items()	Retorna una list conteniendo un tuple por cada clave:valor
keys()	Retorna una list con las claves del diccionario
pop()	Remueve el elemento con la clave especificada
popitem()	Remueve el ultimo par clave valor insertado
setdefault()	Retorna el valor de la clave especificada. SI esta no existe, inserta la clave con el valor indicado
update()	Actualiza el diccionario con la clave valor especificada
values()	Retorna una lista con todos los valores del diccionario

Técnicas de looping

- Al hacer un lazo a través de diccionarios, la clave y el correspondiente valor pueden ser obtenidos de una sola vez usando el método **items()**
- Al hacer un lazo en una secuencia, el índice de posición y su valor se pueden obtener usando la función **enumerate()**
- Para hacer loop sobre dos o mas secuencias al mismo tiempo, las entradas pueden se apareadas usando la función **zip()**
- Para hacer loop en forma inversa sobre una secuencia use la función **reversed()**

Técnicas de looping

- Para hacer loop sobre una secuencia de forma ordenada, use la función **sorted()**, la cual retorna una nueva lista ordenada, dejando la original intacta
- Si alguna vez se trata de cambiar una lista mientras está haciendo un loop sobre ella; es más simple y seguro hacer una copia de la lista.