Facultad de Ciencias de la Administración

TECNICATURA UNIVERSITARIA EN DESARROLLO WEB

PROGRAMACIÓN I

Unidad VI – Colecciones y tipos de datos compuestos

Listas

Tecnicatura Universitaria en Desarrollo Web

Facultad de Ciencias de la Administración

Universidad Nacional de Entre Ríos

- **Objetivos**
  - Identificar características de tipos compuestos y diferencias con los elementales.
  - Comprender las principales características de listas, tuplas y diccionarios y cómo hacer uso de ellas.
- **Temas a desarrollar:**
  - Listas. Definición.
  - Operaciones de modificación y consulta.
  - Clasificación: unidimensionales, bidimensionales.
  - Tuplas. Definición. Operaciones de modificación y consulta.
  - Diccionarios. Definición. Operaciones de modificación y consulta.

Mutable vs Inmutable

- *La **mutabilidad** define si un dato puede ser mutado (cambiado) tras ser inicializado o siempre mantiene el mismo valor, tanto para la variable original como para las demás referencias que tenga el valor.*
  - Este aspecto es muy importante a la hora de trabajar con **listas** e **intentar cambiar sus valores** en cualquier parte del programa ***para evitar al máximo la aparición de efectos colaterales*** al actualizar una variable que se espera que no se actualice.
- Los diferentes tipos de **Python**, pueden ser clasificados atendiendo a su mutabilidad. Pueden ser:
  - **Inmutables:** Si no permiten ser modificados una vez creados.
  - **Mutables**: Si permiten ser modificados una vez creados.

Inmutable	Mutable
Booleanos, Complejos, Enteros, Float, Frozenset, Cadenas, Tuplas, Range y Bytes	Listas, Sets, Bytearray, Memoryview y Diccionarios

Listas tipos de datos mutables

- A diferencia de los strings, las **listas** son **mutables**, lo que significa que podemos **cambiar** sus elementos. Podemos **modificar** uno de sus elementos usando el operador **corchetes** en el lado izquierdo de una asignación:

  » `frutas = ["ananá", "banana", "manzana"]`

  » `frutas[0] = "pera"`

  » `frutas[-1] = "naranja"`

  » `print(frutas)` → `['pera', 'banana', 'naranja']`

- Con el operador de **slice** podemos **reemplazar varios** elementos a la vez:

  » `lista = ['a', 'b', 'c', 'd', 'e', 'f']`

  » `lista[1:3] = ['x', 'y']`

  » `print(lista)` → `['a', 'x', 'y', 'd', 'e', 'f']`

- Además, puede **eliminar** elementos de una lista asignándoles la lista vacía:

  » `lista = ['a', 'b', 'c', 'd', 'e', 'f']`

  » `lista[1:3] = []`

  » `lista` → `['a', 'd', 'e', 'f']`

Borrado en una lista

- El uso de slices para borrar elementos de una lista puede ser confuso y propicio a errores. **Python** nos da una alternativa que resulta más legible: **del** que elimina un elemento de una lista:

```
a = ['uno', 'dos', 'tres']
del a[1]
print(a) → ['uno', 'tres']
```

- **del** maneja índices negativos y provoca un error en tiempo de ejecución sin el índice está fuera de límites.
- Puede usarse un slice como índice para **del**:

```
lista = ['a', 'b', 'c', 'd', 'e', 'f']
del lista[1:5]
print(lista) → ['a', 'f']
```

Métodos de listas

- Python nos provee de métodos que operan con las listas. Por ejemplo, **append** agrega un nuevo elemento al final de la lista:

```
» t = ['a', 'b', 'c']
» t.append('d')
» print(t) → ['a', 'b', 'c', 'd']
```

- El método **extend** toma una lista como argumento y agrega todos sus elementos:

```
» t1 = ['a', 'b', 'c']
» t2 = ['d', 'e']
» t1.extend(t2)
» print(t1) → ['a', 'b', 'c', 'd', 'e']
```

- El método **sort** ordena todos los elementos de la lista de mayor a menor:

```
» t3 = ['d', 'c', 'e', 'b', 'a']
» t3.sort()
» print(t3) → ['a', 'b', 'c', 'd', 'e']
```

Métodos de listas (2)

- Otros métodos:
  - **`insert()`:** agrega un elemento en la posición especificada por el índice.
  - **`remove()`:** quita un elemento a partir del valor especificado.
  - **`pop()`**: quita un elemento a partir del índice especificado.
  - **`index()`**: devuelve el índice del elemento pasado por parámetro.
  - **`clear()`**: vacía la lista.
  - **`reverse()`**: invertir el orden de los elementos en la lista.

Listas como parámetros

- Cuando se pasa una lista como argumento, en realidad se pasa una referencia a ella, no una copia de la lista. Por ejemplo, la función **cabeza()** toma una lista como parámetro y devuelve el primer elemento.

```
def cabeza(lista):
    return lista[0]
```

- Así es como se usa.

```
numeros = [1,2,3]
cabeza(numeros) → 1
```

- El parámetro **lista** y la variable **numeros** son **alias** de un mismo objeto. Si la función modifica una lista pasada como parámetro, el que hizo la llamada verá a el cambio. **borra_cabeza()** elimina el primer elemento de una lista.

```
def borra_cabeza(lista):
    del lista[0]
```

- Aquí vemos el uso de **borra_cabeza()**:

```
numeros = [1,2,3]
borra_cabeza(numeros)
print(numeros) → [2, 3]
```

Listas anidadas

- Una lista anidada es una lista que aparece como elemento dentro de otra lista.
- En esta lista, el tercer elemento es una lista anidada:

  » `lista = ["hola", 2.0, 5, [10, 20]]`

- Si imprimimos `lista[3]`, obtendremos `[10, 20]`. Para extraer los elementos de la lista anidada, podemos proceder en dos pasos:

  » `elt = lista[3]`

  » `elt[0]` → `10`

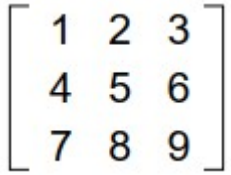
- O podemos combinarlos:

  » `lista[3][1]` → `20`

- Los operadores corchete se evalúan de izquierda a derecha, así que esta expresión saca el tercer elemento de lista y luego extrae el primer elemento de ella

Matrices

- Es común usar listas anidadas para representar matrices. Por ejemplo, la matriz:



```
» matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- **matriz** es una lista con tres elementos, siendo cada elemento una fila de la matriz. Podemos elegir una fila entera de la matriz de la forma normal:

```
» matriz[1] → [4, 5, 6]
```

- O tomar sólo un elemento de la matriz usando la forma de doble índice:

```
» matriz[1][1] → 5
```

- El primer índice escoge la fila y el segundo la columna. Aunque esta manera de representar matrices es común no es la única posibilidad. Una pequeña variación consiste en usar una lista de columnas en lugar de filas.

Cadenas y listas

- Dos de las funciones más útiles de los strings tienen que ver con listas de cadenas. La función **`split()`** divide una cadena en una lista de palabras.
- Por defecto, cualquier número de caracteres de espacio en blanco se considera un límite de palabra:

  » `cancion = "La lluvia en Sevilla..."`

  » `cancion.split()` → `['La', 'lluvia', 'en', 'Sevilla…']`

- Se puede usar un argumento opcional llamado delimitador para especificar qué caracteres se usarán como límites de palabra. El siguiente ejemplo usa la cadena `"l"` como delimitador:

  » `cancion = "La lluvia en Sevilla..."`

  » `cancion.split("l")` → `['La ', 'uvia en Sevi', 'a...']`

- Observe que el delimitador no aparece en la lista.

Cadenas y listas (2)

- La función **`join()`** es la inversa de **`split()`**. A partir de un string, toma una lista de cadenas y concatena los elementos con el primer string como separador:

  » `lista = ['La', 'lluvia', 'en', 'Sevilla...']`

  » `''.join(lista)` → `'LalluviaenSevilla...'`

- En el ejemplo anterior usamos el string vacío como delimitador, por eso el resultado. Si usamos "_" como separador, el efecto es:

  » `'_'.join(lista)` → `'La_lluvia_en_Sevilla...'`

Bibliografía

- Óscar Ramírez Jiménez: ***"Python a fondo"*** 1era Edición. Ed. Marcombo S.L.. 2021.
- Allen Downey. ***"Think Python"***. 2Da Edición. Green Tea Press. 2015.
- Bill Lubanovic. ***"Introducing Python"***. 2Da Edición. O' Reilly. 2020.
- Eirc Matthes: ***"Python Crash Course"***. 1era Edición. Ed. No Starch Press. 2016.
- Zed A. Shaw: ***"Learn Python 3 the Hard Way"***. 1era Edición. Ed. Addison-Wesley. 2017.