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.

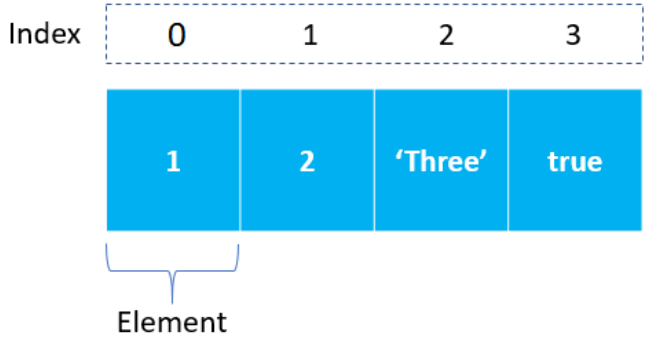
Tipos de datos compuestos

- Algunos de los tipos de datos que hemos visto hasta el momento son: **`bool`**, **`int`**, **`float`**, y **`string`.**
- Los strings son muy diferentes del resto porque están **hechos de piezas menores**: ***caracteres***.
- Los tipos que comprenden **piezas menores** se llaman ***tipos de datos compuestos***.
- Dependiendo de lo que necesitemos hacer, podemos tratar un tipo compuesto como **una única cosa** o acceder sus partes componentes.
- Los tipos de datos compuestos de **Python** que trataremos en esta asignatura son:
  - Listas
  - Tuplas
  - Diccionarios

Listas - Definición

- *Una **lista** es un conjunto ordenado de valores, en el cual cada valor se puede identificar por un índice.*

- Las listas en **Python** son uno de los tipos o ***estructuras de datos*** más versátiles del lenguaje, ya que permiten almacenar un conjunto arbitrario de datos. Es decir, podemos guardar en ellas prácticamente lo que sea.
- Los **valores** que constituyen una **lista** se llaman ***elementos***.
- Las **listas** son similares a los strings, que son **conjuntos ordenados de caracteres**, excepto en que los elementos de una lista pueden ser de ***cualquier tipo***.
- Las **listas** y los strings, (entre otros tipos) que se comportan como **conjuntos ordenados**, se denominan ***secuencias***.



Valores de una lista

- Hay varias maneras de crear una nueva lista; la más sencilla es encerrar sus elementos entre ***corchetes***:

```
[10, 20, 30, 40]
["spam", "elástico", "golondrina"]
```

- Los elementos de una lista pueden ser de distinto tipo:

```
["hola", 2.0, 5, [10, 20]]
```

- Se dice que una **lista** dentro de otra **lista** está ***anidada***.
- Hay unas lista especial que no contiene elementos. Se llama ***lista vacía*** y se representa así: `[]`
- Anteriormente ya creamos listas! ¿Se acuerdan de `print(list(range(4)))`?
  - Allí temporalmente convertimos un rango en una lista para poder imprimirlo.
- También podemos crear una lista a partir de un string: `list("Python")` ¿Revisamos qué efecto tiene ?

Acceso a elementos

- Podemos asignar listas a variables o pasar listas como parámetros a funciones:

  » `beatles = ["John", "Paul", "George", "Ringo"]`

  » `numeros = [17, 123]`

  » `vacio = []`

  » `print(beatles, numeros, vacio)`

  → `['John', 'Paul', 'George', 'Ringo'] [17, 123] []`

- La sintaxis para acceder a los elementos de una lista es la misma que para acceder a los caracteres de un string: el operador corchetes **[]**. La expresión dentro de los corchetes especifica el índice.

- Recordar que los índices siempre comienzan en cero:

  » `print(beatles[0])` → `'John'`

  » `numeros[1] = 5`

- El operador **[]** puede aparecer en cualquier parte de una expresión. Cuando aparece a la izquierda de una asignación, cambia uno de los elementos de la lista, de manera que el "unésimo" elemento de **numeros**, que era **123**, ahora es **5**.

- Se puede usar como índice cualquier expresión entera.

Acceso a elementos (2)

- Si intenta acceder (leer o modificar) un elemento que no existe, obtendrá un error en tiempo de ejecución:

  » `numeros[2] = 5`

  `IndexError: list assignment index out of range`

- Si especificamos un índice negativo, se cuenta hacia atrás desde el final de la lista.

  » `numeros[-1]` → `5`

  » `numeros[-2]` → `17`

  » `numeros[-3]`

Recorridos

- La función **`len()`** toma una lista y devuelve su tamaño. Es buena idea usar este valor como límite superior a la hora de recorrer un bucle en lugar de usar constantes literales.
- Es muy habitual usar una variable de bucle como índice para una lista:

```
tortugas = ["Leonardo", "Raphael", "Donatello", "Miguel Ángel"]
i = 0
while i < len(tortugas):
    print(tortugas[i])
    i = i + 1
```

- Este bucle **`while`** cuenta desde **`0`** hasta la longitud de la lista (en este caso **`4`**). Cuando la variable de bucle vale **`4`**, la condición falla y acaba el bucle.
- Por tanto, el cuerpo del bucle sólo se ejecuta cuando **`i`** es **`0, 1, 2`** y **`3`**.
- Cada vez que recorremos el bucle, la variable i se usa como índice de la lista, imprimiendo el elemento í-esimo. Esta plantilla de computación se llama ***recorrido de lista***

Pertenencia

- `in` es un operador booleano que comprueba la pertenencia a una secuencia.
- Lo usamos cuando vimos el bucle `for` para rangos y strings, pero también funciona con las listas y otras secuencias:

```
» tortugas = ["Leonardo", "Raphael", "Donatello", "Miguel Ángel"]
» "Leonardo" in tortugas
True
» "Rocoso" in tortugas
False
```

- Como “Leonardo” es un miembro de la lista **tortugas**, el operador in devuelve `True` (verdadero). Como “Rocoso” no está en la lista **tortugas**, `in` devuelve `False` (falso).
- Podemos usar `not` en combinación con `in` para comprobar si un elemento no es miembro de una lista:

```
» "Bebop" not in tortugas
True
```

Listas y bucles for

- Los bucles `for` también funcionan con listas. Comparamos los bucles para el ejemplo anterior:

```
for tortuga in tortugas:
    print(tortuga)
```

```
i = 0
while i < len(tortugas):
    print(tortugas[i])
    i = i + 1
```

- El bucle `for` es más conciso porque podemos eliminar `i` (la variable de bucle) y también su actualización.
- Más aún, casi se lee igual que en español, ***"Para (cada) tortuga en (la lista de) tortugas, imprime el nombre de la tortuga".***
- Otros ejemplos:

```
for fruta in ["ananá", "banana", "manzana"]:
    print("Me gusta comer" + fruta + "s!")
```

Operaciones con listas

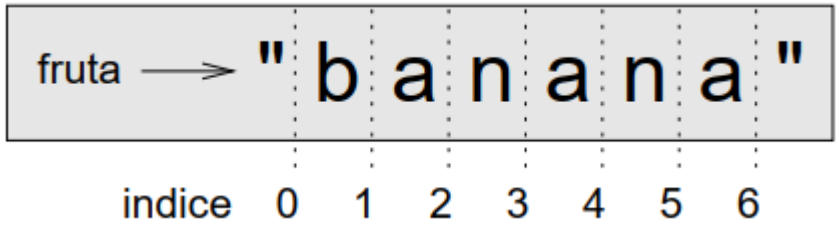
- El operador **+** concatena listas:
  - **a = [1, 2, 3]**
  - **b = [4, 5, 6]**
  - **c = a + b**
  - **print(c)** → **[1, 2, 3, 4, 5, 6]**

- De forma similar, el operador ***** repite una lista un número dado de veces:
  - **[0] * 4** → **[0, 0, 0, 0]**
  - **[1, 2, 3] * 3** → **[1, 2, 3, 1, 2, 3, 1, 2, 3]**

- En el primer ejemplo la lista **[0]** contiene un solo elemento que es repetido cuatro veces. En el segundo ejemplo, la lista **[1, 2, 3]** se repite tres veces.

Slices

- Las operaciones de slices que vimos para strings también funcionan sobre listas:




```
» lista = ['a', 'b', 'c', 'd', 'e', 'f']
» lista[1:3] → ['b', 'c']
» lista[:4] → ['a', 'b', 'c', 'd']
» lista[3:] → ['d', 'e', 'f']
» lista[:] → ['a', 'b', 'c', 'd', 'e', 'f']
```

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.