```
1
```

Estructuras de Datos de Python: Parte 1

Introducción a Python para Visualización de Datos

Dr. Gerardo Ramón Fox

Julio 2022

Python nos proporciona cuatro estructuras de datos básicas:

1. Listas
2. Tuplas
3. Diccionarios
4. Sets

En este módulo veremos listas y tuplas. Más adelante veremos Diccionarios y Sets.

Listas

Son arreglos con comporatamiento dinámico y pueden contener elementos de varios tipos así como otras listas. Para generar una lista vacía, podemos usar las siguientes instrucciones:

```
mi_lista = []
mi_lista = list()
```

También podemos inicializarlas de manera literal:

```
1 numeros = [1, 2, 3, 4, 5]            # Todos enteros.
2 decimales = [1., 2., 3., 4., 5.]     # Todos decimales (note la importancia de usar el pun
3 stack = [10.3, 1, "agua", -3]        # Varios tipos
```

Obtener la longitud o tamaño:

```
1 print(len(numeros))
2 print(len(decimales))
3 print(len(stack))
```

```
5
5
4
```

Acceder a elementos individuales. Es importante siempre tener en mente que si una lista tiene N elementos, los índices van de 0 a $N-1$.

```
print(numeros[0], numeros[3])           # Primera posicion (0) y cuarta posicion (3)
print(decimales[-1], decimales[-2])     # Vamos del final hacia atras: -1 es el ultimo, -
```

```
1 4
5.0 4.0
```

Slicing: acceder a una secuencia de elementos

```
arreglo[inicio:final]
```

Nos retorna una secuencia desde el índice inicio al *índice final-1*. Importante, no va retornar lo que esté en la posición final, se queda en una posición anterior.

```
datos = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(datos[1:5])
print(datos[7:9])
print(datos[7:10])
```

```
[1, 2, 3, 4]
[7, 8]
[7, 8, 9]
```

Más ejemplos de slicing

```
datos = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(datos[:])     # Accede a toda la lista.
print(datos[5:])    # del indice 5 hasta el final
print(datos[:5])    # del inicio hasta el indice 4 !!!!
print(datos[-5:])   # desde el indice -5 hasta el final
print(datos[2:8:2]) # cada dos elementos desde el 2 hasta 8-2.
print(datos[::2])   # cada dos elementos desde el inicio
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9]
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9]
[2, 4, 6]
[0, 2, 4, 6, 8]
```

Para verificar si un elemento está en una lista, podemos usar el operador de membresía **in**:

```
datos = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(0 in datos)   # True
print(1 in datos)   # True
print(100 in datos) # False
```

```
True
True
False
```

Técnicamente podemos construir una matriz de 2x2 como una lista de listas. Lo incluimos como ejemplo, pero se recomienda usar Numpy para el uso eficiente de matrices y estructuras de mayores dimensiones.

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

print(M)
print(M[1][2])
print(M[2][:])
```

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

Las listas se concatenan con el operador +

```
lst = [1, 2, 3] + [0,] + [1.1, 2.2, -1.]
print(lst)
```

```
[1, 2, 3, 0, 1.1, 2.2, -1.0]
```

El operador * genera repeticiones y NO multiplicación. Esto puede ser útil para inicializar estructuras repetitivas.

```
10*[0,]
2*[1,2]
```

```
[1, 2, 1, 2]
```

Agregamos elementos *individuales* con la función append() y agregamos una lista de elementos con la función extend()

```
lst = [1, 2, 3] + [0,] + [1.1, 2.2, -1.]

# Uso de la función append.
```

```
print(lst)
lst.append(-5.1)
print(lst)

# Uso de la función extend.
lst.extend([10, 20, 30])
print(lst)
```

```
[1, 2, 3, 0, 1.1, 2.2, -1.0]
[1, 2, 3, 0, 1.1, 2.2, -1.0, -5.1]
[1, 2, 3, 0, 1.1, 2.2, -1.0, -5.1, 10, 20, 30]
```

NOTA: si usaramos

```
lst.append([10, 20, 30])
```

en lugar de extend, el resultado sería

```
[1, 2, 3, 0, 1.1, 2.2, -1.0, -5.1, [10, 20, 30]]
```

El último elemento almacena una lista como objeto. El uso de cada función dependerá del diseño de su programa.

También, podemos vaciar una lista con la función:

```
lst.clear()
```

El efecto es que lst queda como una lista vacía.

¿Cómo recorrer una lista con un ciclo?

```
nums = [1,2,3,4,5,6]           # Lista de numeros
tot = 0                        # Acumulador de la suma
for i in range(len(nums)):     # range debe tomar el tamaño de la lista como argumento
  tot = tot + nums[i]          # sumamos cada elemento
print("tot = ", tot)           # imprimimos el resultado
```

```
tot =  21
```

Tip de Pythonista

El ciclo anterior también se puede escribir de la forma:

```
1 nums = [1,2,3,4,5,6]          # Lista de numeros
2 tot = 0                       # Acumulador de la suma
3 for n in nums:                # La lista es un iterador
4   tot = tot + n               # sumamos cada elemento
5 print("tot = ", tot)          # imprimimos el resultado
```

```
tot =  21
```

Toda la documentación de las listas está disponible en:
https://docs.python.org/3/tutorial/datastructures.html

Tuplas

Es una estructura que puede almacenar varios elementos de distintos tipos, pero una vez declarada, no se puede modificar. Le llamamos una estructura *inmutable*. Veremos la utilidad de estas estructuras más adelante.

Podemos declarar tuplas de la siguientes maneras:

```
1 t = (1, 2, 3)
2 t2 = ('a', 1, 2.1, [1, 2, 3])
3 t3 = tuple([1, 2, 3])            # A partir de una lista
```

Su tamaño está dado por la función len():

```
1 print(len(t), len(t2), len(t3))
```

```
3 4 3
```

Tiene algunas funciones básicas como count() e index():

```
1 nums = (1, 1, 1, 2, 2, 2, 3, 3, 3)
2 print(nums.count(2))                # Cuenta el numero de elementos que son 2.
3 print(nums.index(3))                # Retorna el indice del primer 3 que encuentra.
```

```
3
6
```

La igualdad de las tuplas se puede evaluar con el operador ==

```
1 print((1, 2, 3) == (1, 2, 3))  # True  / Son iguales.
2 print((1, 2) == (1,))          # False / No son iguales.
```

```
True
False
```

Las tuplas se pueden iterar de una manera similar a las listas, pero no se pueden modificar los valores.

La documentación de las tuplas está disponible:
https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

Conversión o Casteo Entre Estructuras de Datos

Las funciones generadores de cada estuctura se usan para convertir una estructura en otra:

```
list(), tuple()
```

Veamos algunos ejemplos.

```
lst = [1,2,2,2,3,3,4]    # Generamos una lista
print(lst)

tp = tuple(lst)          # La convertimos a tupla
print(tp)

```

```
[1, 2, 2, 2, 3, 3, 4]
(1, 2, 2, 2, 3, 3, 4)
{1, 2, 3, 4}
{'d', 'a', 'c', 'b'}
['d', 'a', 'c', 'b']
```

Comprehensions o Comprensiones

Hay algunas maneras cortas de inicializar listas o diccionarios que llamamos comprensiones:

```
# Comprensiones de Listas
s = [ i for i in range(0, 11) ]              # Genera una lista del 0 al 10
print(s)

t = [ i**2 + 2 for i in range(-10, 11, 2) ]   # Evalúa i^2 + 2 de -10 a 10 con paso de 2 y
print(t)

```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[102, 66, 38, 18, 6, 2, 6, 18, 38, 66, 102]
---
```

```
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
{'0': 0, '1': 1, '2': 8, '3': 27, '4': 64, '5': 125}
```

La ventaja de usar una comprension es indicar al interpretador una operacion iterativa en una sola linea. De esta manera, se evita usar ciclos for "tradicionales", que son menos eficientes, para generar listas.

Copias de Estructuras de Datos

En muchas ocasiones es necesario crear copias de estructuras de datos. En Python, hay que usar ciertos métodos según la estructura para realizar copias. Veamos el siguiente caso para comprender como Python maneja las estructuras de datos:

```
# Creamos una lista:
lst = [1,2,3,4]
print("lst =", lst)

# Intentamos (ingenuamente) hacer una copia
lst_copy = lst

# Cambiamos datos en nuestra copia
lst_copy[1] = 4
lst_copy[3] = 8
lst_copy.extend([5,6,7,8])
print("lst_copy = ", lst_copy)

# Veamos que paso con la original.
print("lst = ", lst)

# Podemos verificar que es la misma estructura con el operador is:
print(lst_copy is lst)
```

```
lst = [1, 2, 3, 4]
lst_copy =  [1, 4, 3, 8, 5, 6, 7, 8]
lst =  [1, 4, 3, 8, 5, 6, 7, 8]
True
```

Vemos que al modificar la copia se modifica la original. Esto es porque al ejecutar:

```
lst_copy = lst
```

lst_copy guarda una referencia a lst de modo que trabajamos con la misma estructura. Esto hace muchas cosas eficientes en cuanto a uso de memoria.

Este comportamiento lo encontraremos en muchos contextos de Python, y es causa de muchos "bugs" cuando se comienza con este lenguaje. Por lo pronto, para estructuras de datos, lo

resolvemos con el método copy. Las listas, tuplas, diccionarios, y sets tienen su método copy. Veamos lo aplicado al ejemplo de lista de arriba:

```
# Creamos una lista:
lst = [1,2,3,4]
print("lst =", lst)

# Hacemos una copia.
lst_copy = lst.copy()

# Cambiamos datos en nuestra copia
lst_copy[1] = 4
lst_copy[3] = 8
lst_copy.extend([5,6,7,8])
print("lst_copy = ", lst_copy)

# La original queda intacta.
print("lst = ", lst)

# Podemos verificar que no es la misma estructura con el operador is:
print(lst_copy is lst)
```

```
lst = [1, 2, 3, 4]
lst_copy =  [1, 4, 3, 8, 5, 6, 7, 8]
lst =  [1, 2, 3, 4]
False
```

Tip de Pythonista

Los métodos copy de listas, tuplas, diccionarios y sets hace una copia "shallow" o "no profunda". ¿Qué significa? Si hay estructuras anidadas, se copian referencias a las estructuras anidadas. Veamos un ejemplo:

```
# Definimos una matriz de 2x2 como una lista que contiene dos listas.
M = [[1, 2], [3, 4]]
print("M = ", M)

# Copiamos
M_copy = M.copy()

# Hacemos cambios en la copia
M_copy[0][1] = 4
M_copy[1][1] = 8

print("M = ", M)
print("M_copy = ", M_copy)
```

```
M =  [[1, 2], [3, 4]]
M =  [[1, 4], [3, 8]]
M_copy =  [[1, 4], [3, 8]]
```

Esto se resuelve con la función deepcopy del módulo copy

```
import copy

# Definimos una matriz de 2x2 como una lista que contiene dos listas.
M = [[1, 2], [3, 4]]
print("M = ", M)

# Copiamos
M_copy = copy.deepcopy(M)

# Hacemos cambios en la copia
M_copy[0][1] = 4
M_copy[1][1] = 8

print("M = ", M)
print("M_copy = ", M_copy) # La original queda intacta.
```

```
M =  [[1, 2], [3, 4]]
M =  [[1, 2], [3, 4]]
M_copy =  [[1, 4], [3, 8]]
```

Para más información sobre el módulo copy, consultar: https://docs.python.org/3/library/copy.html