OFICIOS DIGITALES

PROGRAMACIÓN PYTHON

Estructura de datos

LIC. GUILLERMO ALFARO

IPFL
Instituto Provincial de Formación Laboral

GOBIERNO DE LA PROVINCIA DE BUENOS AIRES

Unidad 2
Estructura de datos

1 – Tuplas

Las Tuplas son secuencias de elementos ordenadas, inmutables y que admiten duplicados. Decimos que una variable es inmutable cuando no puede ser modificada, es un dato estático.

Las Tuplas se definen, en su forma más sencilla, usando paréntesis dentro de los cuales, encerramos los elementos que la forman, separados por comas.

```
tupla = (1,2,3)
print("La variable es de tipo {} y sus elementos son: {}".format(type(tupla),tupla))
```

```
La variable es de tipo <class 'tuple'> y sus elementos son: (1, 2, 3)
```

En el caso particular de necesitar una Tupla de un solo elemento, debemos definirla usando una coma a la derecha de dicho elemento para que el intérprete no la tome como un número entero u otro tipo de dato simple.

```
tupla1=(1)
tupla2=(1,)
print("La variable es de tipo {} y sus elementos son: {}".format(type(tupla1),tupla1))
print("La variable es de tipo {} y sus elementos son: {}".format(type(tupla2),tupla2))
```

```
La variable es de tipo <class 'int'> y sus elementos son: 1
La variable es de tipo <class 'tuple'> y sus elementos son: (1,)
```

Para acceder a los datos de una Tupla, lo hacemos a través de sus índices, recordando siempre que Python comienza a asignar dichos índices a partir del número cero.

```
tupla = (1,2,3)
print(tupla[0])
```

```
1
```

También podemos usar índices negativos para comenzar a contar desde el final de la Tupla, por ejemplo, para obtener el último elemento, utilizaremos el índice -1:

```
tupla = (1,2,3)
print(tupla[-1])
```

```
3
```

Una forma muy utilizada de acceder a los elementos de una secuencia, en este caso de la Tupla, es usando lo que llamamos SLICING, que consiste en determinar la posición inicial y final que dará forma a la sub Tupla.
La sintaxis general es: "Tupla [desde: hasta+1: paso]"

```
tupla = (1,2,3,4,5,6)
print("Tupla desde el índice 2 hasta el 4: {}".format(tupla[2:5]))
```

```
Tupla desde el índice 2 hasta el 4: (3, 4, 5)
```

El paso indica el salto que queremos que haga nuestra SLICING, por ejemplo, de 2 en 2, de 3 en 3, etc.

En el caso de omitir alguno de los argumentos principales, ya sea él DESDE o el HASTA, Python lo reemplaza automáticamente por el primer o el último elemento dependiendo de cuál sea el caso.

→ Intentar reemplazar el valor de algún elemento de la Tupla.

2 – Listas

Una lista es una estructura similar a la Tupla, pero con las siguientes características:

→ Es ordenada.

→ Es mutable.

→ Permite elementos duplicados.

Posee el mismo manejo de datos de las Tuplas, pero se le agrega la posibilidad de modificar el valor de sus elementos.
Pueden definirse de varias maneras, la más habitual es la siguiente:

```
lista = [1,2,3,4]
print(lista)
```

```
[1, 2, 3, 4]
```

También podemos usar el método constructor LIST, por ejemplo, si poseemos una Tupla y necesitamos una lista podemos hacer lo siguiente:

```
tupla = (1,2,3)
lista = list(tupla)
print(type(lista))
```

```
<class 'list'>
```

El acceso a los datos lo hacemos de igual manera que como lo hacíamos con la Tupla y de la misma manera funciona el SLICING.
También nos permite utilizar los operadores de pertenencia sobre las listas:

```
text = "Curso de introducción a Python"
palabra = 'de'
if palabra in texto:
    print("Palabra encontrada")
else:
    print("Palabra no encontrada")
```

```
Palabra encontrada
```

Manejo de datos mediante el usode métodos.

Las operaciones básicas de manejos de datos en listas son:

- → •Agregar elementos
- → •Eliminar elementos.
- → •Ordenar elementos.

a – POP

Extrae un determinado elemento de una lista y lo elimina. En caso de necesitarlo, deberemos almacenarlo en una variable.

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# POP: Elimino el último elemento
print(lista.pop())
print(lista)
```

```
10
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# POP: Elimino el tercer elemento
print(lista.pop(3))
print(lista)
```

```
4
[1, 2, 3, 5, 6, 7, 8, 9, 10]
```

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# POP: Elimino el penúltimo elemento
print(lista.pop(-2))
print(lista)
```

```
9
[1, 2, 3, 4, 5, 6, 7, 8, 10]
```

b – Agregar y modificar elementos

Para modificar un elemento, accedemos a él como siempre, a través de su índice, por ejemplo, siguiendo con nuestra lista de números del 1 al 10, cambiaremos el valor 4 (índice 3) por el valor 'a'.

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# Modifico el 4 por la letra a
lista[3]='a'
print(lista)
```

```
[1, 2, 3, 'a', 5, 6, 7, 8, 9, 10]
```

→ IMPORTANTE: si el elemento no existe, no podemos agregarlo mediante una simple asignación.

Para agregar un nuevo elemento, utilizamos el método INSERT, al que le pasamos como parámetros, el índice y el valor.

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# Modifico el 4 por la palabra Guillermo
lista.insert(15,'Guillermo')
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'Guillermo']
```

Analicemos un poco el ejemplo, si vemos bien, notaremos que INSERT nos permite insertar un nuevo elemento en una posición que no tiene que ser necesariamente la siguiente a la última.

Pero el elemento queda, de todas formas, ubicado en la última posición, es por esto que la lista es una estructura ordenada.

Si por error o de manera intencional, aplicamos INSERT sobre una posición existente, el elemento se inserta en dicha posición y todos los que estaban a partir de esa posición se desplazan a la derecha (su índice aumenta en una unidad).

El método APPEND me permite agregar un elemento al final de la lista de la misma manera que hicimos con la Tupla:

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# append
lista.append('A')
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'A']
```

Por último, en lo que a inserción de elementos se refiere, podemos hacer uso del método EXTEND que me permite hacer una inserción múltiple al final de la lista:

c – Eliminar elementos

Podemos eliminar un determinado elemento de una lista utilizando REMOVE, si dicho elemento no existe, el programa nos dará un error.

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# remove(elemento)
lista.remove(4)
print(lista)
```

```
[1, 2, 3, 5, 6, 7, 8, 9, 10]
```

También podemos eliminar por posición o índice, de manera similar a como lo hicimos por elemento, pero mediante el método DEL, que además permite el uso del SLICING.

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# del(índice o Slicing)
del lista[0]
print(lista)
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
#Creamos una lista con la función RANGE(desde,hasta)
lista = list(range(1,11))

# del(índice o Slicing)
del lista[1:3]
print(lista)
```

```
[1, 4, 5, 6, 7, 8, 9, 10]
```

3 – Diccionarios

Los diccionarios son estructuras que almacenan los datos en base a un sistema de clave/valor (KEY/VALUE).
Sus principales características son:

- → Estructura del tipo clave/valor.
- → Desordenado.
- → Mutable.

Los valores de las claves deben ser inmutables, pueden ser valores numéricos, alfanuméricos o Tuplas.

```
# Definición de un diccionario.
d1 = {'clave1':'valor1',2:[1,2,3]}
print(d1)
```

```
{'clave1': 'valor1', 2: [1, 2, 3]}
```

Definición

```
# Definición de un diccionario.
d1 = {'clave1':'valor1',2:[1,2,3]}
print(d1)
```

```
{'clave1': 'valor1', 2: [1, 2, 3]}
```

Como vemos, cada elemento del diccionario está formado por un par clave/valor y todos los elementos están separados por una coma y encerrados entre corchetes.
Podemos crear y asignar un nuevo elemento en un solo paso:

```
# Asignación
d1['clave2'] = 27
print(d1)
```

```
{'clave1': 'valor1', 2: [1, 2, 3], 'clave2': 27}
```

→ Función ZIP

La función ZIP une los elementos de 2 listas en forma de Tupla.
Es necesario que ambas listas tengan el mismo largo.

Como vemos, la función ZIP nos deja formados pares en formato de Tupla, que podemos convertir en una lista de pares de valores:

```
lista1 = [1,2,3]
lista2 = [4,5,6]

for elemento in zip(lista1,lista2):
    print(elemento)
```

```
(1, 4)
(2, 5)
(3, 6)
```

```
print(list(zip(lista1,lista2)))
```

```
[(1, 4), (2, 5), (3, 6)]
```

Unión de diccionarios

Conociendo el funcionamiento de ZIP, lo usaremos para crear un nuevo diccionario, al que llamaremos d2 y lo uniremos con diccionario d1 que teníamos creado.

```
# Creo el diccionario d2.
d2 = dict(zip(range(5),reversed(range(5))))

print(d2)
```

```
{0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Una vez creado el diccionario, procedemos a unirlo con el que ya teníamos mediante el método UPDATE:

```
# Creo el diccionario d2.
d2 = dict(zip(range(5),reversed(range(5))))

print(d2)
```

```
{0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Manejo de datos

El acceso a los datos del diccionario es similar al que aplicamos a las listas, salvo en que, en este caso, los índices serán las claves.

Una de las principales ventajas de los diccionarios es que sus claves se encuentran hasheadas, esto posibilita el rápido acceso a la información cuando realizamos una búsqueda.

```
# Búsqueda por clave
d3 = {1:'uno',2:'dos',3:'tres'}
clave = 2
print("El valor almacenado en la clave {} es {}".format(clave,d3[clave]))
```

```
El valor almacenado en la clave 2 es dos
```

Mediante el uso del método POP podemos eliminar un determinado elemento del diccionario, por ejemplo, siguiendo con nuestro diccionario d3, vamos a eliminar el valor, cuya clave es 1:

```
d3.pop(1)
print(d3)
```

```
{2: 'dos', 3: 'tres'}
```

Por último, el método GET me devuelve un determinado valor en base a su clave, pero podemos pasarle también un argumento para que nos devuelva un valor por defecto en caso de no existir la clave solicitada.

```
print(d3.get(2))
print(d3.get(12,'Clave no encontrada'))
```

```
dos
Clave no encontrada
```

Control de flujo

Python, como la mayoría de los lenguajes de programación, nos permite agregarle lógica y funciones mediante estructuras de repetición y de decisión.

1 – Estructuras condicionales

Una estructura condicional permite ejecutar o no una determinada porción del código, dependiendo del resultado booleano del análisis de una proposición.
Por ejemplo, el condicional IF – ELSE evalúa el valor lógico de una proposición y en base a dicha evaluación realiza una acción u otra.

```
día = 'lunes'
if día == 'domingo':
    print('Descanso')
else:
    print('Trabajo')
```

```
Trabajo
```

Existen diferentes tipos de evaluación, no necesariamente debemos evaluar si el valor de una variable coincide con un valor determinado, podemos evaluar directamente la existencia o no de una variable o cualquier proposición booleana.

```
día = ''
if día:
    print('Verdadero')
else:
    print('Falso')
```

```
Falso
```

También es posible considerar que obtengamos más de una evaluación booleana verdadera y la sentencia ELIF nos permite contemplarla y tenerla en cuenta para dirigir el flujo de la ejecución.

```
valor = 4
if 0 <= valor < 11:
    print('El valor está entre 0 y 10')
elif valor < 0:
    print('El valor es negativo')
else:
    print('El valor es mayor a 10')
```

```
El valor está entre 0 y 10
```

Ciclos o iteraciones

Los ciclos permiten ejecutar bloques de códigos repetidas veces, haciendo que el código sea legible y compacto.

Ciclo For

Este ciclo realiza una cantidad fija de repeticiones. La cantidad de iteraciones es determinada por un iterador.

Un iterador es un tipo de dato que posee múltiples elementos por los cuales se puede iterar en un orden.

Los ejemplos más comunes son repetir una cantidad N de veces un bloque de código o, para cada componente en una secuencia, ejecutar ciertos comandos.

Veamos un par de ejemplos:

```
# Loop de 5 iteraciones
for x in range(5):
    print("Iteración número {}".format(x+1))
```

```
Iteración número 1
Iteración número 2
Iteración número 3
Iteración número 4
Iteración número 5
```

Ciclo While

Este bucle se ejecutará siempre y cuando se cumpla una determinada condición.

```
x = 1
while x < 6:
    print(x)
    x += 1
else:
    print("Ciclo finalizado")
```

```
1
2
3
4
5
Ciclo finalizado
```

Tanto en los ciclos FOR como WHILE, existe la posibilidad de ejecutar un bloque de código luego de finalizar el bucle, utilizando la palabra clave ELSE

Esto es muy útil para realizar tareas posteriores al ciclo, por ejemplo, si en el bucle se leen las tablas de una base de datos, y al finalizar se cierran las conexiones a dicha base.

Iteradores

Los Iteradores de código son los objetos por los cuales se utilizan los bucles.
Python da la posibilidad de crear Iteradores personalizados, veamos una breve introducción.
Para utilizar un iterador es necesario conocer las funciones ITER y NEXT.

- → ITER: permite crear un iterador básico en función de un tipo de dato secuencial, (lista, Tupla, diccionario).
- → NEXT: devuelve el siguiente elemento de un iterador.

Ejemplo:

```
tupla = ('Enero','Febrero','Marzo')
iterador = iter(tupla)

print(next(iterador))
print(next(iterador))
print(next(iterador))
```

```
Enero
Febrero
Marzo
```

De esta manera, podemos trabajar con una secuencia de Iteradores en lugar de usar solo uno.