Curso Básico de Python

5. Estructuras de datos

Contenido

Este capítulo describe con mayor detalle los tipos de datos compuestos: listas, tuplas, conjuntos y diccionarios.

1 Estructuras de datos

Una *estructura de datos* es una colección de valores relacionados, organizados en tipos de datos como listas, tuplas, conjuntos o diccionarios. Son tipos de datos compuestos a partir de tipos de datos elementales: números enteros y decimales (punto flotante), cadenas de caracteres, valores booleanos `True` y `False`, el tipo nulo `None`.

Lecturas recomendadas:

1. Estructuras de datos en el Tutorial de Python. El contenido de esta capítulo del Tutorial de Python es muy accesible; las secciones siguientes de este capítulo presentan un resumen en forma de referencia rápida (*quick reference*) de ese contenido, con algunas aclaraciones.

2 Listas

Métodos de listas. La clase `list` provee un conjunto de métodos para realizar operaciones sobre una lista. Los siguientes ejemplos muestran el uso de estos métodos.

Sintaxis: `nombre_lista.nombre_metodo([parametros])`
El comando `help("list")` muestra los métodos disponibles para la clase.

Ejemplos de uso de métodos de lista:

```
>>> letras = ["h", "g", "c", "d", "c", "b", "f", "g"]  # define lista
>>> letras.append("A")         # agrega "A" al final
>>> letras
['h', 'g', 'c', 'd', 'c', 'b', 'f', 'g', 'A']
>>> ls_otras = ["W", "X", "Y"] # define otra lista
>>> letras.extend(ls_otras)    # agrega la lista ls_otras al final
>>> letras
['h', 'g', 'c', 'd', 'c', 'b', 'f', 'g', 'A', 'W', 'X', 'Y']
>>> ls_otras.insert(2, "M")    # inserta "M" con índice 2, lugar 3
>>> ls_otras
['W', 'X', 'M', 'Y']
>>> ls_otras.insert(0,"A")     # inserta "A" al principio, índice 0
>>> ls_otras.insert(len(ls_otras), "Z") # inserta "Z" al final
>>> ls_otras
```

```
['A', 'W', 'X', 'M', 'Y', 'Z']
>>> letras.remove("X")         # quita elemento de valor "X", si está
>>> letras
['h', 'g', 'c', 'd', 'c', 'b', 'f', 'g', 'A', 'W', 'Y']
>>> try:
    letras.remove("X")     # devuelve ValueError si no está
except ValueError:
    print("   ValueError: X no está en la lista:")

   ValueError: X no está en la lista:
>>> pop_ult = letras.pop()     # quita y devuelve último valor de la lista
>>> pop_ult, letras
('Y', ['h', 'g', 'c', 'd', 'c', 'b', 'f', 'g', 'A', 'W'])
>>> pop_3 = letras.pop(3)  # quita y devuelve elemento de índice 3, lugar 4
>>> pop_3, letras
('d', ['h', 'g', 'c', 'c', 'b', 'f', 'g', 'A', 'W'])
>>> pop_0 = letras.pop(0)      # quita y devuelve primer elemento, índice 0
>>> pop_0, letras
('h', ['g', 'c', 'c', 'b', 'f', 'g', 'A', 'W'])
>>> i = letras.index("b")      # índice de elemento de valor "b"
>>> i, letras[i]
(3, 'b')
>>> try:
    j = letras.index("b", 5, 7)  # índice de "b" entre 5 y 7
except ValueError:
    print("'b' no está en letras.index('b', 5, 7), entre índices 5 y 7")

'b' no está en letras.index('b', 5, 7), entre índices 5 y 7
>>> i = letras.index("A", 5, 7) # devuelve índice si está entre 5 y 7
>>> i, letras[i]
(6, 'A')
>>> cnt_c = letras.count("c")  # cantidad de veces que aparece "c"
>>> cnt_X = letras.count("X")  # 0 si "X" no está
>>> print("c está", cnt_c, "veces; X está", cnt_X, "veces")
c está 2 veces; X está 0 veces
>>> letras
['g', 'c', 'c', 'b', 'f', 'g', 'A', 'W']
>>> letras.sort()
>>> letras
['A', 'W', 'b', 'c', 'c', 'f', 'g', 'g']
>>> letras.reverse()
>>> letras
['g', 'g', 'f', 'c', 'c', 'b', 'W', 'A']
>>> ls_otras.clear()    # elimina elementos de la lista, la deja vacía
>>> ls_otras
[]
>>> del letras[:]       # lo mismo, usando rebanadas (slicing)
>>> letras
[]
```

Módulos ejemplo: `list_metodos.py`.

Comprensión de listas. La comprensión de listas permite crear una lista a partir de otra, realizando operaciones sobre los elementos de una lista y/o verificando condiciones. La nueva lista se define entre paréntesis rectos `[ ]` e incluirá una o varias sentencias `for` para recorrer los elementos de la lista original, y eventualmnete sentencias `if` para verificar condiciones. La palabra comprensión viene de comprender en el sentido de "contener o incluir en sí algo" (RAE).

Sintaxis, en formato simplificado:
`[ expresión for variable in lista ]` : comprensión sin condición.
`[ expresión for variable in lista if condicion ]` : comprensión con condición.

Ejemplos de comprensión de listas:

```
>>> ls = ['z', 'q', 'p', 'd', 'c', 'c', 'a']
>>> [x+x+x for x in ls]   # comprensión simple con for
['zzz', 'qqq', 'ppp', 'ddd', 'ccc', 'ccc', 'aaa']

>>> [x+x+x for x in ls if x == 'z' or x == 'a']  # comprensión con for e if
['zzz', 'aaa']

>>> [ [x, x+x] for x in ls if x == 'z' or x == 'a']  # lista de listas
[['z', 'zz'], ['a', 'aa']]

>>> lsn = [11, 22, 33]
>>> [ x+str(y) for y in lsn for x in ls ]  # apareo de listas
['z11', 'q11', 'p11', 'd11', 'c11', 'c11', 'a11', 'z22', 'q22', 'p22',
'd22', 'c22', 'c22', 'a22', 'z33', 'q33', 'p33', 'd33', 'c33', 'c33', 'a33']

>>> [ x+str(y) for y in lsn for x in ls[:3] ]  # 3 x 3 = 9 elementos
['z11', 'q11', 'p11', 'z22', 'q22', 'p22', 'z33', 'q33', 'p33']

>>> [ ls[i] + str(lsn[i]) for i in range(0, len(lsn)) ]  # solo 3 items
['z11', 'q22', 'p33']

>>> from math import sqrt
>>> [sqrt(x) for x in lsn]    # aplica una función a elementos de una lista
[3.3166247903554, 4.69041575982343, 5.744562646538029]
```

Matrices. La comprensión de listas facilita la creación y operación con matrices.

```
>>> # matriz 3x3 de enteros 0 a 9, creación
mat = [ [i for i in range(j, j+3)] for j in range(0, 9, 3) ]
>>> print(mat)
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]

>>> # transposición de una matriz
>>> [ [fila[i] for fila in mat] for i in range(3)]
[[0, 3, 6], [1, 4, 7], [2, 5, 8]]
```

La función `zip` itera sobre varias listas generando tuplas con un elemento de cada una de esas listas; devuelve un iterable que puede recorrerse con `for`, o convertirse en una lista; esto último permite usar `zip` para trasponer una matriz.

```
>>> # ejemplo con zip
>>> for it in zip([1, 2, 3], ["A", "B", "C"]):
   print(it, end=" ")

(1, 'A') (2, 'B') (3, 'C')

>>> # transposición de una matriz
>>> list(zip(*mat))
[(0, 3, 6), (1, 4, 7), (2, 5, 8)]
```

Los elementos de una lista peden eliminarse con `del`:

```
>>> #  serializa o aplana una matriz
>>> nums = [ i for fil in mat for i in fil]
>>> nums
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> del nums[0]   # borra primer elemento
>>> nums
[1, 2, 3, 4, 5, 6, 7, 8]
>>> del nums[-1]   # borra último elemento
>>> nums
[1, 2, 3, 4, 5, 6, 7]
```

```
>>> del nums[3:5]  # borra una rebanada (slice)
>>> nums
[1, 2, 3, 6, 7]
>>> del nums[:]    # borra toda la lista
>>> nums
[]
```

Módulo ejemplo: `list_comp.py`.

3 Tuplas

Una *tupla* es una secuencia inmutable de elementos; *inmutable* significa que no se pueden alterar. Una cadena de caracteres también es inmutable.

Sintaxis: `()` ; `elem,` ; `(elem, )` ; `(elem, elem, ... )` ; `tuple(lista)`
El comando `help("tuple")` muestra los métodos definidos para la clase.

```
>>> t0 = ()                # una tupla vacía
>>> t1 = 'uno',            # tupla de un elemento; notar la coma final
>>> t11 = ('single')       # NO ES UNA tupla, sino un string
>>> t12 = ('single',)      # es una tupla, notar la coma
>>> t3 = 'agua', 'piedra', 'palo'     # tupla de 3 elementos
>>> t31 = ('aire', 'pozo', 'pasto')   # tupla definida con paréntesis
>>> for tp in t0, t1, t11, t12, t3, t31:
    print(type(tp), tp)

<class 'tuple'> ()
<class 'tuple'> ('uno',)
<class 'str'> single
<class 'tuple'> ('single',)
<class 'tuple'> ('agua', 'piedra', 'palo')
<class 'tuple'> ('aire', 'pozo', 'pasto')

>>> x, y, z = t3    # desempaquetado de una tupla
>>> x, y, z         # muestra como una tupla ...
('agua', 'piedra', 'palo')
>>> print(x, y, z)  # ... pero las variables tienen cada una su valor
agua piedra palo
>>> t3[1]           # referencia a un elemento específico de una tupla
'piedra'
```

4 Conjuntos

En Python un conjunto se define colocando sus elementos entre llaves `{ }` o con la función `set`.

Sintaxis: `{ elem, elem, elem, ...}  ; set( [elem, elem, elem, ...] }`
El comando `help("set")` muestra los métodos definidos para la clase.

```
>>> cajon = {"manzana", "banana", "naranja", "pera", "durazno"}  # creación
>>> caja = set(["manzana", "uva", "higo"])           # crea con set y lista
>>> print(type(caja), cajon)
<class 'set'> {'naranja', 'manzana', 'durazno', 'pera', 'banana'}
>>> print(type(caja), caja)
<class 'set'> {'manzana', 'uva', 'higo'}
>>> "banana" in caja         # verifica pertenencia al conjunto
True
>>> "durazno" in caja        # verifica pertenencia al conjunto
False
```

```
>>> caja.add("banana")       # agrega elemento a un conjunto
>>> caja
{'banana', 'manzana', 'uva', 'higo'}
>>> caja.add("banana")       # no agrega elemento existente
>>> caja
{'banana', 'manzana', 'uva', 'higo'}

# operaciones con conjuntos, con operadores y con métodos de conjuntos
>>> cajon - caja         # == cajon.difference(caja)
{'naranja', 'pera', 'durazno'}
>>> cajon | caja         # == cajon.union(caja)
{'naranja', 'manzana', 'durazno', 'pera', 'banana', 'higo', 'uva'}
>>> cajon & caja         # == cajon.intersection(caja)
{'banana', 'manzana'}
>>> cajon ^ caja         # == cajon.symmetric_difference(caja)
{'naranja', 'durazno', 'pera', 'higo', 'uva'}

caja.remove('manzana')   # eliminar elemento, da KeyError si no está
>>> caja
{'banana', 'uva', 'higo'}
```

5 Diccionarios

Un *diccionario* es un conjunto de parejas *clave:valor*. Dada una clave, es posible obtener su valor. Cada clave debe ser única en un mismo diccionario. Si se intenta agregar al diccionario una pareja *clave:valor* cuando la clave ya existe en el diccionario, esa clave quedará asociada al nuevo valor.

Sintaxis: `{ clave:valor, ... } ; dict( [ (clave, valor), ... ] )`
El comando `help("dict")` muestra los métodos definidos para la clase.

```
>>> stock = {"tornillos":50, "tuercas":80, "clavos":120}
>>> stock["clavos"]
120
>>> # stock["resortes"]      # KeyError, no existe la clave
>>> stock["arandelas"]=33    # agrega pareja clave valor
>>> stock["clavos"] = 12     # cambia valor de clave existente
>>> stock
{'tornillos': 50, 'tuercas': 80, 'clavos': 12, 'arandelas': 33}

>>> stock.keys()             # iterable de claves
dict_keys(['tornillos', 'tuercas', 'clavos', 'arandelas'])
>>> for key in stock.keys():
   print(key, end=" ")

tornillos tuercas clavos arandelas

>>> stock.values()           # iterable de valores
dict_values([50, 80, 12, 33])
>>> stock.items()            # iterable de tuplas (clave, valor)
dict_items([('tornillos', 50), ('tuercas', 80), ('clavos', 12),
('arandelas', 33)])

>>> stock_mas = dict( [ ("bulones",3), ("tacos",33), ("clavos",888) ] )
>>> stock_mas
{'bulones': 3, 'tacos': 33, 'clavos': 888}
>>> stock.update(stock_mas)    # agrega o actualiza stock con stock_mas
>>> stock
{'tornillos': 50, 'tuercas': 80, 'clavos': 888, 'arandelas': 33, 'bulones':
3, 'tacos': 33}
```

```
>>> pp = stock_mas.pop("tacos")  # elimina un elemento, KeyError si no está
>>> pp
33
>>> stock_mas.clear()            # borra items, deja el diccionario vacío
>>> stock_mas
{}
```

6 Ejercicios

1. Estudiar el capítulo Estructuras de datos en el Tutorial de Python. probando en IDLE u otro intérprete de comandos los ejemplos allí propuestos. Probar también los ejemplos propuestos en el presente capítulo.

2. Mirar la documentación de las estructuras tratadas en este capítulo. Se puede ver en la documentación de la Biblioteca Estándar de Python, secciones Tipos secuencia (`list, tuple, range`), Conjuntos (`set`), y Tipos mapa (`dict`). Un resumen se puede ver en el intértprete con comandos `help("list")`, `help("tuple")`, etc.

3. En la biblioteca `collections` se encuentra la estructura `deque`, apta para implementar listas donde puede agregarse o quitarse elementos de cualquiera de los dos extremos en forma eficiente.
   a) Estudiar la ayuda para esta estructura. Se puede ver en la documentación de la Biblioteca Estándar de Python, Collections, Objetos deque, o en forma resumida en el comando `help("collections.deque")`.
   b) ¿Cómo se puede importar la estructura `deque`?
   c) ¿Qué métodos permiten implementar una lista FIFO? (*First In, First Out*, primero en entrar primero en salir).
   d) ¿Qué métodos permiten implementar una lista LIFO? (*Last In, First Out*, último en entrar, primero en salir).
   e) Crear un deque con los elementos "c","d","e". Mostrar su contenido. Agregar a la izquierda "b", a la derecha "f"; el contenido quedar en "b", "c","d","e", "f".
   f) Quitar y mostrar los elementos primero y último. El deque debe quedar en el estado original.