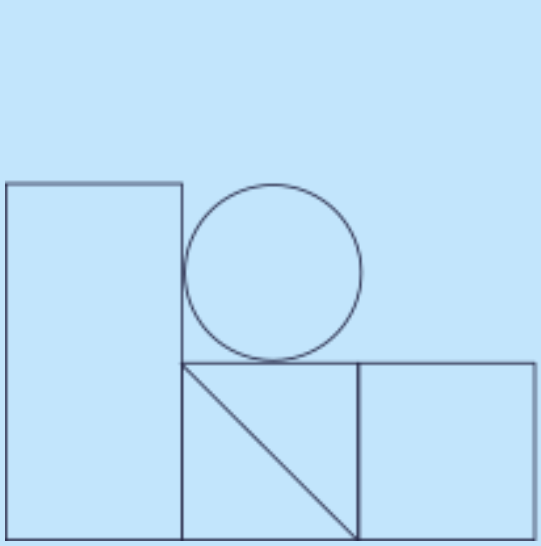
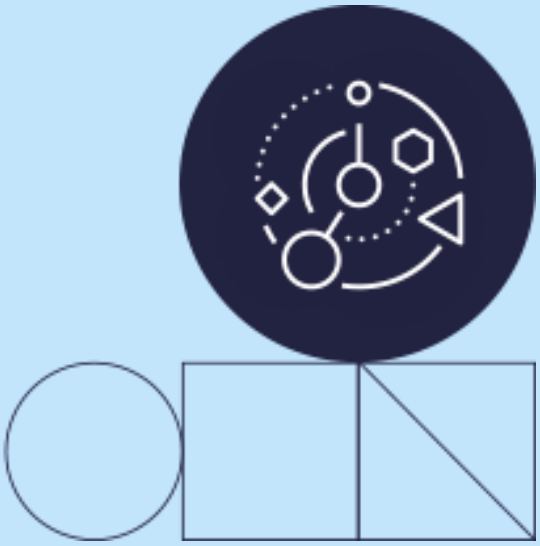
IBM SkillsBuild

IBM

Programando en Python

Estructuras de datos





Índice

Introducción

Hasta ahora hemos visto cómo guardar un dato en una variable para poder trabajar posteriormente con él. Ahora vamos a ver las estructuras que posee Python para poder trabajar con colecciones de datos.

Las estructuras de datos en Python se pueden entender como un tipo de dato compuesto, debido a que en una misma variable podemos almacenar no sólo un dato, sino infinidad de ellos. Dichas estructuras, pueden tener diferentes características y funcionalidades.

Hay cuatro tipos de estructuras de recopilación de datos en el lenguaje de programación Python:

- La lista (list); es una colección ordenada y modificable. Permite miembros duplicados.
- Tupla (tuple); es una colección ordenada e inmutable. Permite miembros duplicados.
- Conjuntos (Set); es una colección que no está ordenada ni indexada. No hay miembros duplicados.
- Diccionario (Dictionary); es una colección desordenada, modificable e indexada. No hay miembros duplicados.

Al elegir un tipo de colección, es útil comprender las propiedades de ese tipo. Elegir el tipo correcto para un conjunto de datos en particular podría significar un aumento en la eficiencia y seguridad.

Listas

La lista es un tipo de colección ordenada y modificable. Es decir, una secuencia de **valores de cualquier tipo, ordenados y de tamaño variable**.

Las listas en Python se representan con el tipo **list** y la sintaxis que se utiliza para definirlas consiste en indicar una lista de objetos separados entre comas y encerrados entre corchetes: [obj1, obj2, ..., objn]

Las listas son estructuras de datos que nos permiten almacenar gran cantidad de valores (equivalentes a los arrays en otros lenguajes de programación). Se pueden expandir dinámicamente añadiendo nuevos elementos, es decir, la cantidad de valores que contendrán podrá variar a lo largo de la ejecución del programa.

Las listas pueden contener cualquier tipo de dato: números, cadenas, booleanos... y también otras listas e incluso funciones, objetos, etc. Se pueden mezclar diferentes tipos de datos. **Las listas son mutables**.

Una de las características importantes de las listas es que se corresponden con una colección ordenada de objetos. El orden en el que se especifican los elementos cuando se define una lista es relevante y se mantiene durante toda su vida.

Sintaxis de las listas:

Crear una lista es tan sencillo como indicar entre corchetes y separados por comas, los valores que queremos incluir en la lista:

nombreLista=**[**elem1**,** elem2**,** elem3...**]**

Se pueden mezclar elementos de distinto tipo.

```
miLista=["Angel", 43, 667767250]
miLista2 = [22, True, "una lista", [1,
2]]
type(miLista)
```

Devolvería: list

También se pueden crear desde strings:

```
>>> list('Python')    # También se pueden
crear desde strings
['P', 'y', 't', 'h', 'o', 'n']
```

Podemos acceder a cada uno de los elementos de la lista escribiendo el nombre de la lista e indicando el índice del elemento entre corchetes.

```
print(miLista[0])        # Imprimiría: Angel
```

Si queremos acceder a un elemento de una lista incluida dentro de otra lista tendremos que utilizar dos veces este operador, primero para indicar a qué posición de la lista exterior queremos acceder, y el segundo para seleccionar el elemento de la lista interior:

```
miLista = [“una lista”, [1, 2]]
mi_var = miLista [1][0]          # mi_var
vale 1
```

También podemos utilizar este operador para modificar un elemento de la lista si lo colocamos en la parte izquierda de una asignación:

```
miLista = [22, True]
miLista [0] = 99                 # Con esto
miLista valdrá [99, True]
```

Para imprimir toda la lista podemos poner:

print(miLista[:])

Se puede declarar una lista vacía, sin elementos.

Como en Java, el primer elemento de la lista está en el índice o posición **cero**.

Nota: Si pongo índices negativos lo que hace **Python** es contar desde el final hasta el principio empezando por -1, es decir, en mi caso el elemento

"Angel" podría ser la posición [0] o la [-3]. Como en los **strings**.

Si pongo:

Lista2=lista[-2:] Me crea una sublista con los elementos [43, 667767250]

Podemos utilizar los operadores vistos en el tema anterior para comparar listas

```
In [16]: lista2 = ['t2', 't1', 't3']

In [17]: lista1 == lista2
Out[17]: False

In [20]: lista1 in lista2
Out[20]: False

In [18]: lista1 == ['t1', 't2', 't3']
Out[18]: True

In [19]: lista1 is ['t1', 't2', 't3']
Out[19]: False
```

Una cosa interesante es que una lista puede llegar a contener una función.

```
In [23]: def func():
             print("Hola mundo")

In [24]: func
Out[24]: <function __main__.func()>

In [31]: lista = ["texto1", "texto2", func]

In [32]: print(lista)
['texto1', 'texto2', <function func at 0x0000012D476D0EE0>]
```

Los elementos de una lista no tienen que ser únicos. Puede repetirse el mismo elemento varias veces en la misma lista

```
In [28]: lista = ["texto", "texto", "texto"]

In [29]: print(lista)
['texto', 'texto', 'texto']
```

Métodos de las listas

Method	Description
append()	Añade un elemento al final de la lista
clear()	Borra los elementos de la lista.
copy()	Devuelve una copia de la lista.
count()	Devuelve el número de veces que se encuentra un elemento en la lista
extend()	Añade los elementos de una lista a otra.
index()	Devuelve el índice del primer elemento cuyo valor es el especificado.
insert()	Añade un elemento en la posición especificada.
pop()	Borra el elemento en la posición especificada y devuelve el elemento eliminado.
remove()	Elimina el elemento con el valor especificado
reverse()	Invierte el orden de la lista.
sort()	Ordena la lista

Los métodos más usados son:

len(), append(), pop(), insert() y **remove().**

Como las listas son colecciones *mutables*, muchos de los métodos de éstas la modifican *in-place* en lugar de crear una lista nueva, como por ejemplo **sort()** o **reverse()**.

```
miLista1 = ["Angel", "Maria", "Manolo",
"Antonio", "Pepe"]
miLista2 = ["Maria", 2, 5.56, True] # Se
puede mezclar diferentes elementos

print (miLista[1])    # Para un elemento
en concreto, se empieza a contar desde la
posición cero.
print (miLista[0:2])    # Empezando desde
cero incluido hasta el dos sin incluir,
esto es, “Angel y María”.
```

Si escribimos tres números (inicio:fin:salto) en lugar de dos, el tercero se utiliza para determinar cada cuantas posiciones añadir un elemento a la lista, Ejemplo:

```
miLista1 = ["Angel", "Maria", "Manolo",
"Antonio", "Pepe"]
miLista2 = ["Maria", 2, 5.56, True] # Se
puede mezclar diferentes elementos

print (miLista[1])    # Para un elemento
en concreto, se empieza a contar desde la
posición cero.
print (miLista[0:2])    # Empezando desde
cero incluido hasta el dos sin incluir,
esto es, “Angel y María”.
```

Listas anidadas

Una lista puede contener cualquier tipo de objeto. Esto incluye otra lista. Una lista puede contener sublistas, que a su vez pueden contener sublistas, y así hasta una profundidad arbitraria.

Estas operaciones sirven para matrices pequeñas y operaciones sencillas. Sin embargo, si queremos operar con matrices grandes o en problemas complejos, **Python** dispone de librerías para este tipo de usos. El principal ejemplo es ***Numpy***, un proyecto de código abierto con gran respaldo de la comunidad científica y de numerosas organizaciones privadas. El proyecto ***Numpy*** es uno de los principales responsables del tremendo éxito de **Python** en el campo de **Data Science**.

Tuplas

Las tupas son un tipo de dato complejo y particular del lenguaje de programación Python. Una tupla es un objeto idéntico a una lista excepto por las siguientes propiedades:

- Al igual que las listas, definen una colección ordenada de objetos, sin embargo, utilizan la sintaxis (obj1, obj2, ..., objn) en lugar de [obj1, obj2, ..., objn]
- Las tuplas son inmutables, es decir, no se pueden modificar después de su creación.
- No permiten añadir, eliminar, mover elementos (no append, extend, remove)
- Sí permiten extraer porciones, pero el resultado de la extracción es una tupla nueva.
- No permiten búsquedas (no index)
- Permiten comprobar si un elemento se encentra en la tupla.

Las tuplas se representan dentro de Python con el tipo de dato **tuple**.

¿Qué utilidad o ventaja presentan las tuplas respecto a las listas?

- Más rápidas
- Manos espacio (mayor optimización)
- Formatean string.
- Pueden utilizarse como claves en un diccionario (las listas no).

La sintaxis básica de una tupla sería:

```
nombreTupla = (elem1, elem2, elem3…)
```

Los paréntesis son opcionales pero recomendables. Las posiciones son como en las listas, el **elem1** está en la posición 0, el **elem2** en la 1, etc.

Podemos utilizar el operador **[]** debido a que las tuplas, al igual que las listas, forman parte de un tipo de objetos llamados **secuencias**. Las cadenas de texto también son secuencias, por lo que no os extrañará que podamos hacer cosas como estas:

```
c = "hola mundo"
c[0]       # h
c[5:]      # mundo
c[::3]     # hauo
```

Al ser inmutables, lógicamente no podemos hacer un **append**, **pop**, etc.

¿Cuándo utilizar una tupla en lugar de una lista?

Hay determinados casos de uso en los que puede ser recomendable utilizar una tupla en lugar de una lista:

- La ejecución del programa es más rápida cuando se manipula una tupla que cuando se trata de una lista equivalente. (Esto probablemente no se note cuando la lista o tupla es pequeña).
- Si los valores de la colección van a permanecer constantes durante la vida del programa, el uso de una tupla en lugar de una lista protege contra la modificación accidental.
- Hay otro tipo de datos de Python que presentaremos próximamente llamado diccionario, que requiere como uno de sus componentes un valor inmutable. Una tupla puede ser utilizada para este propósito, mientras que una lista no puede serlo.

Acceso a elementos de tupla

Puede acceder a los elementos de tupla haciendo referencia al número de índice, entre corchetes:

Ejemplo: Imprime el segundo elemento en la tupla:

```
thistuple =
("apple", "banana", "cherry")
print(thistuple[1])
```

Indexación Negativa

La indexación negativa significa comenzar desde el final, `-1` refiere al último elemento, `-2` refiere al segundo último elemento, etc.

Ejemplo: Imprima el último elemento de la tupla

```
thistuple =
("apple", "banana", "cherry")
print(thistuple[-1])
```

Rango de índices

Puede especificar un rango de índices especificando dónde comenzar y dónde terminar el rango.

Al especificar un rango, el valor de retorno será una nueva tupla con los elementos especificados.

Ejemplo: Devuelve el tercer, cuarto y quinto elemento

```
thistuple =
("apple", "banana", "cherry", "orange"
, "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Nota: La búsqueda comenzará en el índice 2 (incluido) y finalizará en el índice 5 (no incluido).

Recuerde que el primer elemento tiene el índice 0.

Rango de índices negativos

Especifique índices negativos si desea comenzar la búsqueda desde el final de la tupla:

Ejemplo:

Este ejemplo devuelve los elementos del índice -4 (incluido) al índice -1 (excluido)

```
thistuple =
("apple", "banana", "cherry", "orange"
, "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

Cambiar valores de tupla

Una vez que se crea una tupla, no puede cambiar sus valores. Las tuplas son inmutables.

Pero hay una solución alternativa. Podemos convertir la tupla en una lista, cambiar la lista y volver a convertir la lista en una tupla.

Ejemplo: Convierte la tupla en una lista para poder cambiarla:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Recorrer una tupla

Puedes recorrer los elementos de la tupla utilizando un bucle for.

Ejemplo: Iterar a través de los elementos e imprimir los valores.

```
thistuple =
("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Comprobar si el artículo existe

Para determinar si un elemento específico está presente en una tupla, usa la palabra clave in:

Ejemplo: Comprueba si "manzana" está presente en la tupla.

```
thistuple =
("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits
tuple")
```

Longitud de la tupla

Para determinar cuántos elementos tiene una tupla, usa el método len():

Ejemplo: Imprime el número de elementos en la tupla.

```
thistuple =
("apple", "banana", "cherry")
print(len(thistuple))
```

Agregar artículos

Una vez que se crea una tupla, no puedes agregarle elementos. Las tuplas son **inmutables**.

Ejemplo: No puedes agregar elementos a una tupla.

```
thistuple =
("apple", "banana", "cherry")
thistuple[3] = "orange" # This will
raise an error
print(thistuple)
```

Crear tupla con un artículo

Para crear una tupla con un solo elemento, debes agregar una coma después del elemento, a menos que Python no reconozca la variable como una tupla.

Ejemplo: Tupla de un artículo, recuerda la coma.

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Eliminar elementos

Nota: No puedes eliminar elementos en una tupla.

Las tuplas no se pueden **cambiar**, por lo que no puedes eliminar elementos de él, pero puedes eliminarlas por completo:

Ejemplo: La palabra clave `del` puede eliminar la tupla por completo.

```
thistuple =
("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an
error because the tuple no longer
exists
```

Une dos tuplas

Para unir dos o más tuplas, puedes usar el operador `+` :

Ejemplo: Une dos tuple.

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

El constructor tuple ()

También es posible usar el constructor `tuple ()` para hacer una tupla.

Ejemplo: Usando el método tuple () para hacer una tupla.

```
thistuple =
tuple(("apple", "banana", "cherry")) #
note the double round-brackets
print(thistuple)
```

Métodos de las tuplas

Python tiene dos métodos integrados que puede usar en tuplas.

Método	Descripción
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

```
miTupla=("Angel", 4, 5.345, True, 4)
tuplaUnitaria=("Sara",)          #Tupla unitaria. Hay que poner al final ","

print(miTupla[2])                #Igual que en las listas
miLista=list(miTupla)            #Con list convierto una tupla en una lista
miTupla2=tuple(miLista)          #Con tuple convierto una lista en una tupla.
print("Angel" in miTupla)        #Está "Angel" en miTupla?, devuelve True o False
print(miTupla.count(4))          #Cuantas veces se encuentra el elemento 4 en miTupla?
print(len(miTupla))        #Longitud de miTupla
```

Podemos definir la tupla sin poner los paréntesis, es lo que se conoce como “empaquetado de tupla”. En principio no se recomienda:

```
miTupla="Angel", 4, 5.345, True, 4
#Desempaquetado de tupla. Permite asignar todos los elementos de una tupla a diferentes variables:
miTupla=("Angel", 4, 5.345, True, 4)
nombre, num1, num2, valor1, num3=miTupla
```

Solo con esto ya nos ha asignado los valores de miTupla a las variables que hemos definido, podemos hacer la prueba imprimiéndolas:

```
print(nombre)
print(num1)
print(num2)
print(valor1)
print(num3)
```

Diccionarios

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor.

Un diccionario es una colección desordenada, modificable e indexada. En Python, los diccionarios se escriben entre llaves y tienen claves y valores.

Son estructuras de datos que nos permiten almacenar valores de diferente tipo (números, strings, etc) e incluso listas y otros diccionarios.

La principal característica de los diccionarios es que los datos se almacenan asociados a una clave de tal forma que se crea una asociación de tipo clave-valor para cada elemento.

Los elementos almacenados no están ordenados.

Ejemplo: Crear e imprimir un diccionario.

```
dic = {
    "Nombre":"Santiago",
    "Apellido":"Hernandez",
    "Pais":"España",
    "Ciudad":"Madrid"
}
print(dic)

# Otra forma de definir diccionarios con
la funcion dict()
dic2 = dict(
    Nombre="Santiago",
    Apellido="Hernandez",
    Pais="España",
    Ciudad="Madrid"
)
```

En **Python** un diccionario sería similar a lo que en **Java** un **hashtable** o en **PHP** un **array asociativo**.

El primer valor se trata de la clave y el segundo del valor asociado a la clave. Como clave podemos utilizar cualquier valor inmutable: podríamos usar números, cadenas, booleanos, tuplas... pero no listas o diccionarios, dado que son mutables. Esto es así porque los diccionarios se implementan como tablas hash, y a la hora de introducir un nuevo par clave-valor en el diccionario se calcula el hash de la clave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara el objeto clave después de haber sido introducido en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado.

Métodos de los diccionarios

Python tiene un conjunto de métodos integrados que podemos usar en los diccionarios.

Método	Descripción
clear()	Borra todos los elementos de un diccionario
copy()	Devuelve una copia de un diccionario
fromkeys()	Devuelve un diccionario con las claves y valores especificados
get()	Devuelve el valor de una clave
items()	Devuelve una lista que contiene una tupla por cada par clave-valor
keys()	Devuelve una lista que contiene las claves del diccionario
pop()	Borra el elemento con la clave especificada
popitem()	Borra el último par clave-valor insertado
setdefault()	Devuelve el valor de la clave especificada. Si no existe, inserta la clave con el valor especificado.
update()	Actualiza el diccionario con el par clave-valor que se especifique
values()	Devuelve una lista con los valores del diccionario

Bytes y Bytearray

¿Qué son los bytes en Python?

Un **byte** es una ubicación de memoria con un tamaño de 8 bits. Un objeto **bytes** es una secuencia inmutable de **bytes**, conceptualmente similar a un **string**.

El objeto **bytes** es importante porque cualquier tipo de dato que se escribe en disco se escribe como una secuencia de **bytes**, los enteros o las cadenas de texto son secuencias de **bytes**. Lo que convierte la cadena de **bytes** en una cadena de texto o un número entero, es la forma en la que se interpreta.

Sets, Conjuntos

Los sets son un tipo de datos en Python que permite almacenar múltiples elementos en una misma variable. A diferencia de las listas, la colección de elementos que forman un set:

- No se puede indexar
- No respeta un orden
- No puede contener valores duplicados

Los sets se representan dentro de python con el tipo de dato **set**.

La sintaxis que se utiliza para definir un set en Python es la siguiente:

{val1, val2, ..., valn}

También llamados **sets**. Los **sets** (conjuntos) son un tipo de datos básico de Python diferente a las secuencias y los diccionarios, pero aun así de gran utilidad. Los **sets** son conjuntos de objetos mutables no ordenados, (únicos y no ordenados), que se basan en la noción matemática de conjuntos.

De hecho, implementan una serie de métodos que permiten trabajar con conjuntos de objetos de la misma manera que lo hacemos en conjuntos matemáticos. Podemos hacer intersecciones, uniones, diferencias, etc.

No obstante, un conjunto no puede incluir objetos mutables como listas, diccionarios, e incluso otros conjuntos.

Son útiles cuando queremos trabajar con datos masivos y queremos extraer información relevante de ellos.

Establecer métodos

Python tiene un conjunto de métodos integrados que puede usar en conjuntos.

Método	Descripción
add()	Añade un elemento al set
clear()	Borra todos los elementos del set
copy()	Devuelve una copia del set
difference()	Devuelve un set que contiene las diferencias entre dos o más sets
difference_update()	Borra los elementos del set que están incluidos en otro
discard()	Borra el elemento especificado
intersection()	Devuelve un set que es la intersección resultante de otros dos
intersection_update()	Borra los elementos del set que no están presentes en otro
isdisjoint()	Informa si dos sets tienen una intersección o no
issubset()	Informa si otro set contiene a este set o no
issuperset()	Informa si este set contiene a otro set o no
pop()	Borra un elemento del set, no podremos elegir cuál.
remove()	Borra un elemento específico
symmetric_difference()	Devuelve un set con las diferencias simétricas de dos sets
symmetric_difference_update()	Inserta las diferencias simétricas de este set y otro
union()	Devuelve un set con la unión de dos sets
update()	Actualiza el set con la unión de este set y otros