IV. TIPOS DE COLECCIÓN DE DATOS

Objetivo:
Identificar las colecciones de datos utilizadas en Python.

a. Listas
b. Tuplas
c. Diccionarios

Colección de datos

Definición:

Una colección representa un grupo de objetos, conocidos como sus elementos. Las colecciones permiten elementos duplicados y los mismos no están ordenados. Los elementos de una colección no son accedidos a través de un índice. Por el contrario su única forma de recuperación es uno a uno, secuencialmente.

Collection en inglés significa colección, conjunto, acopio, agregación.

La característica fundamental de las colecciones es que aceptan elementos duplicados.

Listas

La lista es un tipo de colección ordenada. Sería equivalente a lo que en otro lenguajes se conoce por arrays, o vectores.

Las listas pueden contener cualquier tipo de dato: números, cadenas, booleanos y también listas.

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

l = [22, True, “una lista”, [1, 2]]

Listas

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. Ten en cuenta sin embargo que el índice del primer elemento de la lista es 0, y no 1:

```
l = [11, False]

mi_var = l[0] # mi_var vale 11
```

Listas

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:

l = [22, True, “una lista”, [1, 2]]

l[3][0] #Muestra el valor 1

l[3][1] #Muestra el valor 2

Listas

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

```
l = [22, True]

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

El uso de los corchetes para acceder y modificar los elementos de una lista es común en muchos lenguajes, pero Python nos depara varias sorpresas muy agradables.

Listas

Una curiosidad sobre el operador [] de Python es que podemos utilizar también números negativos. Si se utiliza un número negativo como índice, esto se traduce en que el índice empieza a contar desde el final, hacia la izquierda; es decir, con [-1] accederemos al último elemento de la lista, con [-2] al penúltimo, con [-3], al antepenúltimo, y así sucesivamente.

listas

```
>>> x = [1,2,3,4,5,6,7,8,9,10]
>>> x[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> x[0:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> x[:0]
[]
>>> x[9:]
[10]
>>> x[:9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[0:9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[0:10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> x[-1]
10
>>> x[-10]
1
>>> x[:-10]
[]
>>> x[-10:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Operaciones en Listas

1. Operaciones de definición

Estas operaciones nos permiten definir o crear una lista.

1.1. [ ]

Crea una lista vacía.

```
y = [ ]
```

2. Operaciones mutables

Estas operaciones nos permiten trabajar con listas alterando o modificando su definición previa.

Operaciones en Listas

append

Añade un único elemento al final de la lista.

```
>>> x = [1,2]
>>> x.append("Uno")
>>> print(x)
 [1, 2, 'Uno']
```

Operaciones en Listas

extend
Añade otra lista al final de una lista.

```
>>> x = [1, 2]
>>> x.extend([3, 4])
>>> print(x)
[1, 2, 3, 4]
```

Operaciones en Listas

insert
Inserta un nuevo elemento en una posición determinada de la lista, este método recibe un primer argumento que equivale a la posición, y un segundo argumento que corresponde al elemento por agregar.

```
>>> x = [1, 2]
>>> x.insert(0, "y")
>>> print(x)
['y', 1, 2]
```

Operaciones en Listas

del

Elimina el elemento ubicado en el índice descrito, a su vez este método tiene la capacidad de eliminar una sección de elementos de la lista, mediante el operador ":" que permite definir un punto de [inicio:fin] sin considerar el punto final, esto puntos pueden ser obviados, con lo cual como punto de inicio se tomará la posición 0 y punto de fin la última posición de la lista.

```
>>> x = [1, 2, 3]
>>> del x[1]
>>> print(x)
[1, 3]
>>> y = [1,2,3,4,5]
>>> del y[:2]
>>> print(y)
[3, 4, 5]
```

Operaciones en Listas

remove

Remueve la primer coincidencia del elemento especificado.

```
>>> x = [1,2,"h",3,"h"]
>>> x.remove("h")
>>> print(x)
[1, 2, 3, 'h']
```

Operaciones en Listas

reverse
Invierte el orden de los elementos de la lista, dando como resultado una lista con los elementos del final al inicio y del inicio al final.

```
>>> x = [1,2,3,4,5,6,7,8,9]
>>> x.reverse()
>>> print(x)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Operaciones en Listas

sort
Por defecto este método ordena los elementos de la lista de menor a mayor, este comportamiento puede modificarse mediante el parámetro reverse=True.

```
>>> x = [4,2,3,1]
>>> x.sort()
>>> print(x)
[1, 2, 3, 4]
>>> y = [14,12,13,11]
>>> y.sort(reverse=True)
>>> print(y)
[14, 13, 12, 11]
```

Operaciones en Listas

3. Operaciones inmutables

Estas operaciones nos permiten trabajar con listas sin alterar o modificar su definición previa.

sorted

Este método nos permite crear una nueva lista ordenada a partir de otra lista no ordenada.
Algo interesante del método sorted, es que este no se encuentra limitado a las listas, sino que a su vez es aplicable a las tuplas y los diccionarios, algo que en otro artículo estaremos abordando.

```
>>> lista1 = [2,5,8,1,4,5,8,3,2,5]
>>> lista2 = sorted(lista1)
>>> print(lista1)
[2, 5, 8, 1, 4, 5, 8, 3, 2, 5]
>>> print(lista2)
[1, 2, 2, 3, 4, 5, 5, 5, 8, 8]
```

Operaciones en Listas

+

Esta operación nos permite concatenar o unir dos listas diferentes en nueva lista.

```
>>> x = [1,2,3,4,5]
>>> y = [6,7,8,9,10]
>>> print(x+y)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Operaciones en Listas

*

Esta operación nos permite replicar una lista hasta la cantidad de veces indicada.

```
>>> x = [1,2,3]
>>> print(x * 3)
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Operaciones en Listas

min
Este método retorna el elemento más pequeño dentro de una lista.
max
Al contrario del metodo min, este método retorna el elemento más grande dentro de una lista.

```
>>> x = [40,100,3,9,4]
>>> print(min(x))
3
>>> print(max(x))
100
>>> print(min(x),max(x))
3 100
```

Operaciones en Listas

index
Retorna la posición en la lista del elemento especificado.

```
>>> x = [10,30,20]
>>> print(x.index(30))
1
```

Operaciones en Listas

count

Retorna la cantidad de veces que el elemento especificado se encuentra en la lista.

```
>>> x = [10,30,20,30,30,20,10]
>>> print(x.count(30))
3
```

Operaciones en Listas

sum
Este método realiza la suma de los elementos de la lista, esto siempre que los mismos puedan ser sumados. Sum es un método muy utilizado con listas de tipo numéricas.

```
>>> x = [2.5,3,3.5]
>>> print(sum(x))
9.0
```

Operaciones en Listas

in

El método nos permite determinar si un elemento específico se encuentra en una lista, este retorna únicamente dos posibles valores True cuando el elemento se encuentre en la lista, y False cuando este no lo esté. Este método es ampliamente utilizado para evitar excepciones en métodos tales como: index y remove, ya que en caso que el elemento buscado no se encuentre en la lista, dará como resultado una excepción.

```
>>> x = ["h",2,"a",6,9]
>>> print("a" in x)
True
```

Operaciones en Listas

len

La función len devuelve la longitud (el número de elementos) de un objeto.

```
>>> x = ["h",2,"a",6,9]
>>> print(len(x))
5
```

Duplas

Todo lo que hemos explicado sobre las listas se aplica también a las tuplas, a excepción de la forma de definirla, para lo que se utilizan paréntesis en lugar de corchetes.

t = (1, 2, True, “python”)

En realidad el constructor de la tupla es la coma, no el paréntesis, pero el intérprete muestra los paréntesis, y nosotros deberíamos utilizarlos, con claridad.

>>> t = 1, 2, 3

>>> type(t)

type “tuple”

Duplas

Para referirnos a elementos de una tupla, como en una lista, se usa el operador [ ]:

```
mi_var = t[0] # mi_var es 1
mi_var = t[0:2] # mi_var es (1, 2)
```

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

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

Diferencias Listas/Duplas

La diferencia es que las listas presentan una serie de funciones adicionales que permiten un amplio manejo de los valores que contienen. Basándonos en esta definición, puede decirse que las listas son dinámicas, mientras que las tuplas son estáticas.

Diccionarios

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor. Por ejemplo, veamos un diccionario de películas y directores:

d = {“Love Actually “: “Richard Curtis”,

“Kill Bill”: “Tarantino”,

“Amélie”: “Jean-Pierre Jeunet”}

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.

Diccionarios

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.
La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les accede no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador [].

d[“Love Actually “] # devuelve “Richard Curtis”

Al igual que en listas y tuplas también se puede utilizar este operador para reasignar valores.
d[“Kill Bill”] = “Quentin Tarantino”

Sin embargo en este caso no se puede utilizar slicing, entre otras cosas porque los diccionarios no son secuencias, si no mappings (mapeados, asociaciones).

V. OPERADORES RELACIONALES

Objetivo:
Utilizar operadores relacionales en expresiones lógicas en Python

a. Operadores

Tabla

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	`>>> 5 == 3` `False`
!=	¿son distintos a y b?	`>>> 5 != 3` `True`
<	¿es a menor que b?	`>>> 5 < 3` `False`
>	¿es a mayor que b?	`>>> 5 > 3` `True`
<=	¿es a menor o igual que b?	`>>> 5 <= 5` `True`
>=	¿es a mayor o igual que b?	`>>> 5 >= 3` `True`

VI. Estructuras de control en Python

Objetivo:
Conocer y utilizar las estructuras condicionales e iterativas con Python.

a. if/else
b. while
c. do while
d. for

Estructuras de Control

Las estructuras de control se utilizan para controlar el flujo de un programa (o bloque de instrucciones), son métodos que permiten especificar el orden en el cual se ejecutarán las instrucciones en un algoritmo. Si no existieran las estructuras de control, los programas se ejecutarían linealmente desde el principio hasta el fin, sin la posibilidad de tomar decisiones.

Por lo general, en la mayoría de lenguajes de programación encontraremos dos tipos de estructuras de control. Encontraremos un tipo que permite la ejecución condicional de bloques del programa y que son conocidas como estructuras condicionales. Por otro lado, encontraremos las estructuras iterativas que permiten la repetición de un bloque de instrucciones, un número determinado de veces o mientras se cumpla una condición.

Aquí es donde cobran su importancia el tipo booleano y los operadores lógicos y relacionales.

if/else

Una sentencia if en Python esencialmente indica:
"Si la expresión evaluada, resulta ser verdadera(True), **entonces ejecuta una vez el código** en la expresión. Si sucede el caso contrario y la expresión es falsa, **entonces no ejecutes el código que sigue.**"

Una sentencia if consiste en:

- La palabra reservada if , da inicio al condicional if .
- La siguiente parte es la condición. Esta puede evaluar si la declaración es verdadera o falsa. En Python estas son definidas por las palabras reservadas (True or False).
- Paréntesis (()) Los paréntesis son opcionales, no obstante, ayudan a mejorar la legibilidad del código cuando más de una condición está presente.
- Dos puntos : cuya función es separar la condición de la declaración de ejecución siguiente.
- Una nueva línea.
- Un nivel de indentación de cuatro espacios, que es una convención en Python. El nivel de indentación es asociado con la estructura de la declaración que sigue.
- Finalmente, la estructura de la sentencia. Este es el código que será ejecutado, únicamente si la sentencia a ser evaluada es verdadera. Es posible tener múltiples líneas en la estructura de código que pueden ser ejecutadas; en este caso es necesario tener cautela en cuanto a que todas las líneas tengan el mismo nivel de indentación.

if/else

Estructura básica

```
if (True):
    print(“La expresión es Verdadera”)
```

Una sentencia if ejecuta el código, solo en caso de cumplirse la condición especulada.
¿Qué sucede si queremos ejecutar el código alternativo en caso de no cumplirse las condiciones? En este caso es necesario usar else.

```
if condicion:
    ejecutar codigo si la condicion es True
else:
    ejecutar codigo si la condicion es False
```

elif

¿Qué si necesitamos más de dos opciones?

En vez de decir: "Si la primera condición es verdadera, realiza esto, si no, realiza esto otro", ahora le indicamos al programa, "Si esto no es verdadero, intenta esto otro, y si todas las condiciones fallan en ser verdaderas, entonces haz esto.

```
if condicion1:
    ejecutar codigo
elif condicion2:
    ejecutar codigo
else:
    ejecutar codigo
```

Ciclos

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los ciclos nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

while

El ciclo **while** permite ejecutar un bloque de instrucciones mientras se cumpla la condición dada. Primero comprueba que en efecto se cumple la condición dada y entonces, ejecuta el segmento de código correspondiente hasta que la condición no se cumpla.

Estrucutura básica:

```
numero = 0
while numero <= 10:
    print (numero)
    numero += 1

numero = 0
while numero <= 10:
    print (random.randint(1,10))
    numero += 1
```

while

Estrucutura básica:

```
import random

numero = 0
while numero <= 10:
    print (random.randint(1,10))
    numero += 1
```

do while

La sentencia do while es una estructura de repetición o iterativa que se utiliza en muchos lenguajes de programación.

Cuando utilizamos este bucle nos aseguramos que las instrucciones se ejecuten por lo menos una vez.

Estructura básica:

for

Los ciclos for permiten ejecutar una o varias instrucciones de forma iterativa, una vez por cada elemento en la colección.

Las colecciones pueden ser de varios tipos, el for puede recibir una colección predefinida o directamente de la salida de una función.

Estructura básica:

```
for contador in range(1,10):
    print (contador)
```

Iteraciones sobre una lista (for)

Los ciclos for se utilizan en Python para recorrer secuencias, por lo que vamos a utilizar un tipo de secuencia lista.

Leamos la cabecera del bucle como si de lenguaje natural se tratara: “para cada elemento en secuencia”. Y esto es exactamente lo que hace el bucle: para cada elemento que tengamos en la secuencia, ejecuta estas líneas de código. Lo que hace la cabecera del bucle es obtener el siguiente elemento de la secuencia y almacenarlo en una variable de nombre elemento. Por esta razón en la primera iteración del bucle elemento valdrá “uno”, en la segunda “dos”, y en la tercera “tres”.

Estructura básica:

```
numeros = [“uno”, ”dos”, ”tres”]

for indice in numeros:
    print (indice)
```

Ejercicios

1.- Poblar una lista con 100 números (reales) aleatorios

2.- Poblar una lista con 10 números aleatorios y obtener la suma total

3.- Poblar una lista con datos introducidos en el teclado

4.- Poblar una lista de tuplas pares, que contengan Nombre y edad de personas.

Funciones

Una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos.

En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor None (nada), equivalente al null de Java. Además de ayudarnos a programar y depurar dividiendo el programa en partes las funciones también permiten reutilizar código.En Python las funciones se declaran de la siguiente forma:

Funciones

Estructura básica:

1. La palabra clave def
2. Un nombre de función
3. Paréntesis ’()’, y dentro de los paréntesis los parámetros de entrada, aunque los parámetros de entrada sean opcionales. (Argumentos)
4. Dos puntos ’:’
5. Algún bloque de código para ejecutar
6. Una sentencia de retorno (opcional)

```
def mi_funcion(param1, param2):
        print param1
        print param2
        return
```

Funciones

También podemos encontrarnos con una cadena de texto como primera línea del cuerpo de la función. Estas cadenas se conocen con el nombre de docstring (cadena de documentación) y sirven, como su nombre indica, a modo de documentación de la función.

```
def mi_funcion(param1, param2):
    """Esta función imprime los dos valores pasados como parámetros"""
    print param1
    print param2
    return
```