Curso de Especialización de «Desarrollo de Aplicaciones en Lenguaje Python»

Avanza 2024/25

Módulo «Estructuras de control en Python»

Unidad de trabajo 3

Sentencias iterativas

Profesor: Ismael Reyes Rodríguez

Tabla de contenido

1 Objetivos 4
2 Concepto de sentencia iterativa 5
3 Diferencias entre estructuras condicionales e iterativas 6
4 Funcionamiento de las sentencias iterativas. 7
5 Aplicación de sentencias iterativas 8
5.1 Recorrer colecciones de datos 8
5.2 Repetir acciones hasta que se cumpla una condición 8
5.3 Procesamiento por índices 9
5.4 Realizar operaciones en un rango de valores 9
5.5 Iteración en diccionarios 10
5.6 Procesamiento de datos hasta una entrada del usuario 10
6 Sentencia iterativa «while» 11
6.1 Romper un bucle while 12
6.2 Comprobar la rotura 13
6.3 Continuar un bucle 14
6.4 Bucle infinito 15
6.5 Bucles «while» anidados 17
7 Sentencia iterativa «for» 19
7.1 Bucles «for» anidados 22
8 Estructuras de datos iterables en Python 24
8.1 Listas 24
8.1.1 Obtener un elemento 25
8.1.2 Trocear una lista 26
8.1.3 Invertir una lista 26
8.1.4 Añadir al final de la lista 27
8.1.5 Creando desde vacío 27
8.1.6 Añadir en cualquier posición de una lista 28
8.1.7 Repetir elementos 28
8.1.8 Combinar listas 28
8.1.9 Modificar una lista 29
8.1.10 Modificar con troceado 29
8.1.11 Borrar elementos 30

8.1.12 Borrado completo de la lista 31
8.1.13 Encontrar un elemento 31
8.1.14 Pertenencia de un elemento 32
8.1.15 Obtener la longitud de una lista 32
8.1.16 Obtener el número de ocurrencias 32
8.1.17 Dividir una cadena de texto en lista 33
8.1.18 Unir una lista en cadena de texto 33
8.1.19 Ordenar una lista 34
8.1.20 Iterar sobre una lista 34
8.1.21 Iterar usando enumeración 35
8.1.22 Iterar sobre múltiples listas 35
8.2 Tuplas 36
8.2.1 Crear tuplas 36
8.2.2 Convertir un iterable en tupla 37
8.2.3 Otras operaciones con tuplas 37
8.3 Diccionarios 37
8.3.1 Crear diccionarios 38
8.3.2 Convertir iterables en diccionarios 38
8.3.3 Obtener un elemento 39
8.3.4 Añadir o modificar un elemento 40
8.3.5 Utilizar un patrón de creación de diccionarios 40
8.3.6 Comprobar si una clave existe 41
8.3.7 Obtener la longitud 41
8.3.8 Obtener todos los elementos 41
8.3.9 Borrar elementos 42
8.3.10 Iterar sobre un diccionario 43
8.4 Conjuntos (Sets) 44
8.5 Funciones para estructuras iterables 46
8.6 Creación de iterables personalizados 47
9 Web para explorar 48

1 Objetivos

El objetivo principal que se conseguirá con la realización de esta unidad de trabajo será el de utilizar sentencias iterativas analizando las necesidades del código para resolver un problema.

Para la consecución de dicho objetivo se trabajarán varios aspectos entre los que destacan:

- Concepto de sentencia iterativa.
- Diferencias entre estructuras condicionales e iterativas.
- Funcionamiento de las sentencias iterativas.
- Aplicación de sentencias iterativas.
- Programando con bucles «for» y «while».
- Anidamientos de bucles.

Algunos de estos aspectos ya se han tratado en la Unidad de trabajo 1, pero se analizarán con más detenimiento en este tema.

2 Concepto de sentencia iterativa

Las **sentencias iterativas en Python** permiten ejecutar un bloque de código repetidamente mientras se cumpla una condición o durante un número determinado de iteraciones. Son fundamentales en la programación para automatizar tareas repetitivas y manipular colecciones de datos.

Python ofrece dos tipos principales de sentencias iterativas:

- **Bucle «for»:** Se utiliza para iterar sobre una secuencia (como listas, tuplas, cadenas, conjuntos, diccionarios o cualquier objeto iterable). En cada iteración, el bucle asigna un elemento de la secuencia a una variable y ejecuta el bloque de código asociado. Es útil para recorrer colecciones o realizar operaciones un número específico de veces.
  Un ejemplo de esta sentencia sería:

```
for elemento in [1, 2, 3]:
    print(elemento)
```

  Salida: 1
  2
  3

- **Bucle «while»:** Ejecuta un bloque de código mientras una condición lógica sea verdadera. Este tipo de bucle es ideal cuando no se conoce de antemano cuántas veces se ejecutará la iteración.
  Un ejemplo de esta sentencia sería:

```
contador = 0
while contador < 3:
    print(contador)
    contador += 1
```

  Salida: 0
  1
  2

Las sentencias iterativas son una herramienta esencial para procesar datos, implementar algoritmos y resolver problemas de manera eficiente en Python.

Para su correcto uso no se deben olvidar conceptos trabajados en temas anteriores, como los bloques de código o los operadores condicionales y lógicos.

3 Diferencias entre estructuras condicionales e iterativas

No se deben confundir las sentencias condicionales con las iterativas. Aunque los dos tipos de sentencias son dos pilares fundamentales de la programación, cumplen propósitos diferentes:

- Las **estructuras condicionales** permiten tomar decisiones en el flujo del programa basándose en una condición (verdadera o falsa). Ejecutan un bloque de código solo si la condición especificada se cumple. Ejemplo*: if, elif, else*.
- Las **estructuras iterativas** permiten repetir un bloque de código varias veces mientras una condición se cumpla o se iteren elementos en una colección. Ejemplo: *for, while*.

Es habitual, en cualquier lenguaje de programación, utilizar sentencias iterativas y condicionales mezcladas en el código para, por ejemplo, repetir varias veces un bloque si se cumple una determinada condición o, evaluar en un bucle una condición sobre los datos iterados, pero siempre deben quedar claras las diferencias entre ambos tipos de sentencia.

Las **sentencias condicionales** alteran el flujo de ejecución dependiendo del resultado de la evaluación lógica. No tienen un mecanismo de repetición.

Las **sentencias iterativas** mantienen un flujo cíclico, ejecutando el mismo bloque de código repetidamente hasta que una condición se invalida o se recorren todos los elementos.

4 Funcionamiento de las sentencias iterativas.

Teóricamente, las **sentencias iterativas** en programación son estructuras de control de flujo diseñadas para ejecutar un bloque de código de manera repetitiva, ya sea mientras se cumpla una condición lógica o al recorrer una secuencia de elementos.

Su funcionamiento básico puede explicarse en tres pasos clave:

1. **Inicialización**

   Antes de que comience la iteración, se establece el contexto inicial necesario.

   Para un **bucle for**, esto puede ser el inicio de una colección iterable o la definición de un rango de valores.

   Para un **bucle while**, implica evaluar una condición inicial que determine si el bucle debe ejecutarse.

2. **Ejecución del Bloque de Código**

   En cada iteración, se ejecuta el bloque de código asociado a la sentencia iterativa.

   En un **bucle for**, se asocia una variable temporal que toma el valor de cada elemento de la secuencia, uno a la vez.

   En un **bucle while**, la ejecución continúa mientras la condición se mantenga verdadera.

3. **Evaluación y Continuación**

   Después de cada ejecución del bloque, se verifica si se cumplen las condiciones para continuar con la siguiente iteración.

   En un **bucle for**, el proceso avanza al siguiente elemento de la secuencia, deteniéndose al alcanzar el final.

   En un **bucle while**, la condición lógica se vuelve a evaluar. Si esta se invalida (*False*), el bucle finaliza.

¡Ojo! Si no se implementa correctamente un criterio para finalizar la iteración, puede surgir un **bucle infinito**, donde la sentencia iterativa nunca termina. Por eso, es importante garantizar que la condición de salida sea alcanzable.

5 Aplicación de sentencias iterativas

Las **sentencias iterativas** se aplican en programación cuando se necesita repetir una tarea de manera sistemática. Esto incluye recorrer colecciones de datos, realizar cálculos repetitivos, procesar información basada en condiciones dinámicas, y más.

Las sentencias iterativas, por lo tanto, son extremadamente útiles en tareas repetitivas y dinámicas, desde manipulación de datos hasta interacción con usuarios o cálculos matemáticos.

Veamos algunas circunstancias comunes con ejemplos en Python.

5.1 Recorrer colecciones de datos

Las sentencias iterativas se utilizan cuando se necesita procesar cada elemento de una colección como una lista, tupla o diccionario.

En el siguiente ejemplo, se recorre una lista de nombres y se imprime un saludo para cada elemento.

```
nombres = ["Ana", "Luis", "María"]
for nombre in nombres:
    print(f"Hola, {nombre}!")
```

Salida: Hola, Ana!

Hola, Luis!

Hola, María!

5.2 Repetir acciones hasta que se cumpla una condición

Las sentencias iterativas se utilizan cuando es necesario repetir un proceso hasta que una condición cambie.

En el siguiente ejemplo, se incrementa un contador hasta que llegue al valor 5.

```
contador = 0
while contador < 5:
    print(f"Contador: {contador}")
    contador += 1
```

Salida: Contador: 0

Contador: 1

Contador: 2

Contador: 3

Contador: 4

5.3 Procesamiento por índices

Las sentencias iterativas se utilizan cuando se necesita acceso a los índices de una colección.

En el siguiente ejemplo, se usa el índice para acceder y procesar los valores de una lista.

```
numeros = [10, 20, 30, 40]
for i in range(len(numeros)):
    print(f"Índice {i}, Valor {numeros[i]}")
```

Salida: Índice 0, Valor 10

Índice 1, Valor 20

Índice 2, Valor 30

Índice 3, Valor 40

5.4 Realizar operaciones en un rango de valores

Las sentencias iterativas se utilizan cuando se requiere repetir un bloque de código para un conjunto de números dentro de un rango.

En el siguiente ejemplo, se calcula el cuadrado de los números del 1 al 5.

```
for i in range(1, 6):
    print(f"El cuadrado de {i} es {i**2}")
```

Salida: El cuadrado de 1 es 1

El cuadrado de 2 es 4

El cuadrado de 3 es 9

El cuadrado de 4 es 16

El cuadrado de 5 es 25

5.5 Iteración en diccionarios

Las sentencias iterativas se utilizan cuando se necesita trabajar con claves y valores de un diccionario.

En el siguiente ejemplo, se recorren las claves y valores del diccionario y se imprime información formateada.

```
edades = {"Ana": 25, "Luis": 30, "María": 22}
for nombre, edad in edades.items():
    print(f"{nombre} tiene {edad} años.")
```

Salida: Ana tiene 25 años.

Luis tiene 30 años.

María tiene 22 años.

5.6 Procesamiento de datos hasta una entrada del usuario

Las sentencias iterativas se utilizan cuando un programa depende de entradas dinámicas para detenerse.

En el siguiente ejemplo, el bucle continúa hasta que el usuario escriba "salir".

```
while True:
    entrada = input("Escribe 'salir' para finalizar: ")
    if entrada.lower() == "salir":
        break
    print(f"Escribiste: {entrada}")
```

***Salida* / Entrada:** Escribe 'salir' para finalizar: me quedo

Escribiste: me quedo

Escribe 'salir' para finalizar: salir

6 Sentencia iterativa «while»

La **sentencia while** ya se explicó en el tema 1. Este apartado servirá para refrescar lo que ya se expuso en dicho tema de modo que podamos afrontar el punto Estructuras de datos iterables en Python sin necesidad de consultar otro documento. Además, se han añadido dos apartados nuevos, Bucle infinito y Bucles «while» anidados.

Como ya se vio, el primer mecanismo que existe en Python para repetir instrucciones es usar la **sentencia while**. La semántica tras esta sentencia es: «*Mientras se cumpla la condición haz algo*».

Veamos un sencillo bucle que repite un saludo mientras así se desee:

```
quiere_saludar = 'S'  # importante dar un valor inicial


while quiere_saludar == 'S':
    print('Hola qué tal!')
    quiere_saludar = input('¿Quiere otro saludo? [S/N] ')
print('Que tenga un buen día')
```

***Salida / Entrada**:* *Hola qué tal!*

¿Quiere otro saludo? [S/N] S

Hola qué tal!

¿Quiere otro saludo? [S/N] n

Que tenga un buen día

La condición del bucle se comprueba en cada nueva repetición. En este caso chequeamos que la variable *quiere_saludar* sea igual a 'S'. Dentro del cuerpo del bucle estamos mostrando un mensaje y pidiendo la opción al usuario.

6.1 Romper un bucle while

Python ofrece la posibilidad de romper o finalizar un bucle antes de que se cumpla la condición de parada mediante la **sentencia break**.

Supongamos que, en el ejemplo anterior, establecemos un máximo de 4 saludos:

```
MAX_SALUDOS = 4
num_saludos = 0
quiere_saludar = 'S'
while quiere_saludar == 'S':
    print('Hola qué tal!')
    num_saludos += 1
    if num_saludos == MAX_SALUDOS:
        print('Máximo número de saludos alcanzado')
        break
    quiere_saludar = input('¿Quiere otro saludo? [S/N] ')
print('Que tenga un buen día')
```

***Salida / Entrada**:* *Hola qué tal!*

¿Quiere otro saludo? [S/N] S

Hola qué tal!

¿Quiere otro saludo? [S/N] S

Hola qué tal!

¿Quiere otro saludo? [S/N] S

Hola qué tal!

Máximo número de saludos alcanzado

Que tenga un buen día

Como hemos visto en este ejemplo, **«break»** nos permite finalizar el bucle una vez que hemos llegado al máximo número de saludos. Pero si no hubiéramos llegado a dicho límite, el bucle habría seguido hasta que el usuario indicara que no quiere más saludos.

Otra forma de resolver este ejercicio sería incorporar una condición al bucle:

```
while quiere_saludar == 'S' and num_saludos < MAX_SALUDOS:
...
```

6.2 Comprobar la rotura

Python nos ofrece la posibilidad de detectar si el bucle ha acabado de forma ordinaria, esto es, ha finalizado por no cumplirse la condición establecida. Para ello podemos hacer uso de la sentencia **else** como parte del propio bucle. Si el bucle **while** finaliza normalmente (sin llamada a *break*) el flujo de control pasa a la sentencia opcional **else**.

```
MAX_SALUDOS = 4


num_saludos = 0
quiere_saludar = 'S'


while quiere_saludar == 'S':
    print('Hola qué tal!')
    num_saludos += 1
    if num_saludos == MAX_SALUDOS:
        print('Máximo número de saludos alcanzado')
        break
    quiere_saludar = input('¿Quiere otro saludo? [S/N] ')
else:
    print('Usted no quiere más saludos')
print('Que tenga un buen día')
```

Salida / Entrada: *Hola qué tal!*

¿Quiere otro saludo? [S/N] S

Hola qué tal!

¿Quiere otro saludo? [S/N] n

Usted no quiere más saludos

Que tenga un buen día

Es importante saber que, si hubiéramos agotado el número de saludos **NO se habría ejecutado la cláusula else** del bucle ya que habríamos roto el flujo con un ***break***. Es por esto que la sentencia ***else*** sólo tiene sentido en aquellos bucles que contienen un ***break***.

6.3 Continuar un bucle

Hay situaciones en las que, en vez de romper un bucle, nos interesa saltar adelante hacia la siguiente repetición. Para ello Python nos ofrece la **sentencia continue** que hace precisamente eso, descartar el resto del código del bucle y saltar a la siguiente iteración.

Continuamos con el ejemplo anterior y vamos a contar el número de respuestas válidas:

```
quiere_saludar = 'S'
opciones_validas = 0

while quiere_saludar == 'S':
    print('Hola qué tal!')
    quiere_saludar = input('¿Quiere otro saludo? [S/N] ')
    if quiere_saludar not in 'SN':
        print('No le he entendido pero le saludo')
        quiere_saludar = 'S'
        continue
    opciones_validas += 1
print(f'{opciones_validas} respuestas válidas')
print('Que tenga un buen día')
```

***Salida / Entrada**:* *Hola qué tal!*

¿Quiere otro saludo? [S/N] S

Hola qué tal!

¿Quiere otro saludo? [S/N] rt

No le he entendido pero le saludo

Hola qué tal!

¿Quiere otro saludo? [S/N] S

Hola qué tal!

¿Quiere otro saludo? [S/N] N

3 respuestas válidas

Que tenga un buen día

6.4 Bucle infinito

Si no establecemos correctamente la **condición de parada** o bien el valor de alguna variable está fuera de control, es posible que lleguemos a una situación de bucle infinito, del que nunca podamos salir. Veamos un ejemplo de esto:

```
num = 1


while num != 10:
    num += 2
    ...
```

Este bucle, en teoría, nunca se parará por lo que deberemos pulsar **CTRL-C** para cancelar la ejecución del programa. El intérprete nos mostrará en consola esta "parada obligada" mediante el mensaje ***KeyboardInterrupt*.**

El problema de este bucle surge de que la variable *num* toma los valores *1, 3, 5, 7, 9, 11, ...* por lo que nunca se cumple la condición de parada del bucle. Esto hace que se repita eternamente la instrucción de incremento.

Una posible solución a este error es reescribir la condición de parada en el bucle:

```
num = 1


while num < 10:
    num += 2
    ...
```

Hay veces que un supuesto bucle infinito puede ayudar a resolver un problema. Imaginemos que se desea escribir un programa que ayude al profesorado a introducir las notas de un examen. Si la nota no está en el intervalo [0, 10] se deberá mostrar un mensaje de error, en otro caso debe seguir pidiendo valores:

```
while True:
    nota = float(input('Introduzca nueva nota: '))
    if not(0 <= nota <= 10):
        print('Nota fuera de rango')
        break
    print(nota)
```

Salida / Entrada: *Introduzca nueva nota: 7*

7.0

Introduzca nueva nota: 5

5.0

Introduzca nueva nota: 9.2

9.2

Introduzca nueva nota: 12

Nota fuera de rango

A modo de curiosidad, el código anterior se podría enfocar haciendo uso del operador morsa que vimos en el tema anterior:

```
while 0 <= (nota := float(input('Introduzca una nueva nota: '))) <= 10:
    print(nota)
print('Nota fuera de rango')
```

6.5 Bucles «while» anidados

Al igual que se mostró en el tema anterior en sentencias condicionales, el anidamiento es una técnica por la que incluimos distintos niveles de encapsulamiento de sentencias, unas dentro de otras, con mayor nivel de profundidad. En el caso de los bucles también es posible hacer anidamiento.

Los **bucles while anidados** se utilizan cuando necesitas realizar iteraciones dentro de otras iteraciones, es decir, un bucle *while* contenido dentro de otro. Esto es útil para problemas como recorrer matrices, generar patrones, o manejar estructuras con múltiples dimensiones.

A continuación, se muestran un par de ejemplos.

Programa que genera una tabla de multiplicar:

```
i = 1
while i <= 5:  # Bucle externo
    j = 1
    while j <= 5:  # Bucle interno
        print(f"{i} x {j} = {i * j}", end="\t")
        j += 1
    print()  # Salto de línea al final de cada fila
    i += 1
```

Salida:

1 x 1 = 1 1 x 2 = 2 1 x 3 = 3 1 x 4 = 4 1 x 5 = 5

2 x 1 = 2 2 x 2 = 4 2 x 3 = 6 2 x 4 = 8 2 x 5 = 10

3 x 1 = 3 3 x 2 = 6 3 x 3 = 9 3 x 4 = 12 3 x 5 = 15

4 x 1 = 4 4 x 2 = 8 4 x 3 = 12 4 x 4 = 16 4 x 5 = 20

5 x 1 = 5 5 x 2 = 10 5 x 3 = 15 5 x 4 = 20 5 x 5 = 25

Programa que genera un patrón de asteriscos:

```
n = 5
i = 1
while i <= n:  # Bucle externo controla las filas
    j = 1
    while j <= i:  # Bucle interno controla las columnas
        print("*", end=" ")
        j += 1
    print()  # Salto de línea después de cada fila
    i += 1
```

Salida:

```
*
* *
* * *
* * * *
* * * * *
```

Como vemos, el funcionamiento de estas anidaciones de bucles es similar:

1. El bucle externo controla la cantidad de iteraciones principales (filas, en estos ejemplos).
2. El bucle interno se ejecuta completamente en cada iteración del bucle externo y maneja las sub-operaciones (como columnas o elementos dentro de una fila).
3. Ambos bucles tienen condiciones y actualizaciones independientes para evitar bucles infinitos.

Cabe reseñar que es perfectamente posible la anidación de más de dos bucles *while*.

7 Sentencia iterativa «for»

La **sentencia for** se introdijo en el tema 1 y, en este apartado, se refrescará lo ya expuesto de modo que podamos afrontar con solvencia el contenido del punto Estructuras de datos iterables en Python.

Como se vio, Python permite recorrer aquellos tipos de datos que son *iterables*, es decir, que admiten iterar sobre ellos mediante el bucle «**for**». Algunos ejemplos de tipos y estructuras de datos que permiten ser iteradas (recorridas) son: cadenas de texto, listas, diccionarios, ficheros, ….

A continuación, se plantea un ejemplo en el que vamos a recorrer una cadena de texto:

```
palabra = 'Bicho!'

for letra in palabra:
    print(letra)
```

Salida: *B*

i

c

h

o

!

La clave aquí está en darse cuenta que el bucle va tomando, en cada iteración, cada uno de los elementos de la variable que especifiquemos. En este caso concreto *letra* va tomando cada una de las letras que existen en *palabra*, porque una cadena de texto está formada por elementos que son caracteres.

Una sentencia ***break*** dentro de un ***for*** rompe el bucle, igual que veíamos para los bucles *while*. Veamos un ejemplo con el código anterior. En este caso vamos a recorrer una cadena de texto y pararemos el bucle cuando encontremos una letra 'c':

```
palabra = 'Bicho!'

for letra in palabra:
    if letra == 'c':
        break
    print(letra)
```

Salida*: B*

i

Al igual que en los bucles *while*, también se puede utilizar la sentencia ***continue*** en bucles *for*. Esta sentencia se utiliza para omitir el resto del bloque de código de la iteración actual del bucle *for* y pasar directamente a la siguiente iteración. Esto es útil cuando necesitas saltarte ciertas condiciones específicas sin detener completamente la ejecución del bucle.

Veamos un **ejemplo** en el que el bucle imprime solo los números impares dentro de un rango. Si el número es par, se usa «*continue*» para omitir el resto del código y pasar al siguiente número:

```
for numero in range(1, 10):  # Itera del 1 al 9
    if numero % 2 == 0:  # Si el número es par
        continue  # Salta al siguiente número
    print(numero)
```

***Salida**: 1*

3

5

7

9

Es muy habitual hacer uso de **secuencias de números** en bucles. Python no tiene una instrucción específica para ello. Lo que sí aporta es una función ***range()*** que devuelve un flujo de números en el rango especificado. Una de las grandes ventajas es que la «lista» generada no se construye explícitamente, sino que cada valor se genera bajo demanda. Esta técnica mejora el consumo de recursos, especialmente en términos de memoria.

La función ***range*** tiene tres parámetros:

- **start**: Indica el primer valor que se asignará a la variable del bucle. Es opcional y tiene valor por defecto 0.
- **stop**: Indica hasta qué valor llegará la variable. Es obligatorio (siempre se llega a 1 menos que este valor).
- **step**: Indica en cuánto se incrementará/decrementará el valor de la variable en la siguiente iteración. Es opcional y tiene valor por defecto 1.

Veamos algunos ejemplos para aclarar el suso de *range*:

```
for i in range(0, 3):
    print(i)
```

Salida:

0

1

2

El mismo resultado se obtendría utilizando *range(3).*

```
for i in range(1, 6, 2):
    print(i)
```

Salida:

1

3

5

```
for i in range(2, -1, -1):
    print(i)
```

Salida:

2

1

0

Se suelen utilizar nombres de variables i, j, k para lo que se denominan **contadores**. Este tipo de variables toman valores numéricos enteros como en los ejemplos anteriores. No conviene generalizar el uso de estas variables a situaciones en las que, claramente, tenemos la posibilidad de asignar un nombre semánticamente más significativo.

Hay situaciones en las que **no necesitamos usar la variable** que toma valores en el rango, sino que únicamente queremos repetir una acción un número determinado de veces. Para estos casos se suele recomendar usar el guion bajo '_' como nombre de variable, que da a entender que no estamos usando esta variable de forma explícita:

```
for _ in range(10):
    print('se repite 10 veces')
```

7.1 Bucles «for» anidados

Los **bucles for**, al igual que los bucles *while*, también pueden anidarse. Son útiles cuando se necesita iterar sobre estructuras más complejas, como matrices, combinaciones de valores, o patrones. El bucle interno ejecuta su conjunto de instrucciones completamente para cada iteración del bucle externo.

Veamos un ejemplo de 2 bucles anidados en el que generamos las tablas de multiplicar del 1 al 9:

```
for tabla in range(1, 10):
    for factor in range(1, 10):
        resultado = tabla * factor
        print(f'{tabla} * {factor} = {resultado}')
```

Salida:

*1 * 1 = 1*

*1 * 2 = 2*

*1 * 3 = 3*

…

*9 * 7 = 63*

*9 * 8 = 72*

*9 * 9 = 81*

El siguiente ejemplo, muestra el mismo patrón de asteriscos que se conseguía en el ejemplo mostrado en la anidación de bucles *while*:

```
n = 5
for i in range(1, n + 1):  # Bucle externo controla las filas
    for j in range(1, i + 1):  # Bucle interno controla las columnas
        print("*", end=" ")
    print()  # Salto de línea después de cada fila
```

Salida:

```
*
* *
* * *
* * * *
* * * * *
```

Algunas consideraciones:

- Cuando se ejecutan instrucciones como *break* o *continue* dentro de bucles anidados, las sentencias solo afectan al bucle en el que está insertados, no a todos los bucles.
- Es perfectamente posible la anidación de más de dos bucles *for*.

8 Estructuras de datos iterables en Python

En temas anteriores, ya se han visto los tipos de datos simples en Python y también se ha dado alguna información sobre tipos más complejos, como son las listas, tuplas o diccionarios.

Este tipo de datos más complejos se denominan **estructuras de datos** y, aunque el objetivo final del módulo de "*Estructuras de control en Python*" no es profundizar sobre estas estructuras de datos, sí es conveniente tener un conocimiento de las mismas para sacar el máximo partido al uso de **sentencias iterativas**.

En Python, las **estructuras de datos iterables** son aquellas que permiten recorrer sus elementos uno por uno, generalmente utilizando un **bucle «for».** Este comportamiento se debe a que implementan el protocolo de iteración de Python. Comprender estas estructuras y su uso es esencial para manejar colecciones de datos de manera eficiente y realizar operaciones repetitivas o automatizadas.

En este apartado no se profundizará en exceso sobre las características de los objetos iterables, pero sí vamos a describir estas estructuras de datos y se expondrán ejemplos de cómo usarlas e iterar con ellas.

8.1 Listas

Las **listas** son colecciones **ordenadas y mutables** que pueden contener elementos de cualquier tipo de dato.

Ejemplo:

```
frutas = ["manzana", "plátano", "cereza"]
for fruta in frutas:
    print(fruta)
```

Salida:

manzana

plátano

cereza

Aquí vemos otros ejemplos de definición de listas:

```
empty_list = []
languages = ['Python', 'Ruby', 'Javascript']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]
data = ['Tenerife', {'cielo': 'limpio', 'temp': 24}, 3718, (28.29, -16.52)]
```

Para convertir otros tipos de datos en una lista podemos usar la función ***list():***

```
# conversión desde una cadena de texto
>>> list('Python')
['P', 'y', 't', 'h', 'o', 'n']
```

Veamos algunas operaciones que se pueden realizar con las listas.

8.1.1 Obtener un elemento

Igual que en el caso de las cadenas de texto, podemos obtener un elemento de una lista a través del **índice** (lugar) que ocupa. Veamos un ejemplo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'
>>> shopping[1]
'Huevos'
>>> shopping[2]
'Aceite'
>>> shopping[-1]  # acceso con índice negativo
'Aceite'
```

El índice que usemos para acceder a los elementos de una lista tiene que estar comprendido entre los límites de la misma. Si usamos un índice antes del comienzo o después del final obtendremos un error (excepción) del tipo "*IndexError: list index out of range*".

8.1.2 Trocear una lista

El troceado de listas funciona de manera totalmente análoga al troceado de cadenas. Veamos algunos ejemplos:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[0:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[2:4]
['Aceite', 'Sal']

>>> shopping[-1:-4:-1]
['Limón', 'Sal', 'Aceite']

>>> # Equivale a invertir la lista
>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

8.1.3 Invertir una lista

Python nos ofrece, al menos, tres mecanismos para invertir los elementos de una lista:

1. **Conservando la lista original**. Mediante troceado de listas con step negativo:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

2. **Conservando la lista original**. Mediante la función ***reversed():***

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> list(reversed(shopping))
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

3. **Modificando la lista original:** Utilizando la función **reverse()** (nótese que es sin «d» al final):

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.reverse()

>>> shopping
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

8.1.4 Añadir al final de la lista

Una de las operaciones más utilizadas en listas es añadir elementos al final de las mismas. Para ello Python nos ofrece la función ***append().*** Se trata de un método «destructivo» que modifica la lista original:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.append('Atún')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Atún']
```

8.1.5 Creando desde vacío

Una forma muy habitual de trabajar con listas es empezar con una vacía e ir añadiendo elementos poco a poco. Se podría hablar de un patrón creación.

Supongamos un ejemplo en el que queremos construir una lista con los números pares del 0 al 20:

```
>>> even_numbers = []

>>> for i in range(20):
...     if i % 2 == 0:
...         even_numbers.append(i)
...

>>> even_numbers
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

8.1.6 Añadir en cualquier posición de una lista

Ya hemos visto cómo añadir elementos al final de una lista. Sin embargo, Python ofrece una función ***insert()*** que vendría a ser una generalización de la anterior, para incorporar elementos en cualquier posición. Simplemente debemos especificar el índice de inserción y el elemento en cuestión. También se trata de una función destructiva:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.insert(1, 'Jamón')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Aceite']

>>> shopping.insert(3, 'Queso')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Queso', 'Aceite']
```

8.1.7 Repetir elementos

Al igual que con las cadenas de texto, el operador de multiplicación, «*», nos permite repetir los elementos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping * 3
['Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite']
```

8.1.8 Combinar listas

Python nos ofrece dos aproximaciones para combinar listas:

1. **Conservando la lista original:** Mediante el operador «+» o «+=»:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping + fruitshop
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

2. **Modificando la lista original**: Mediante la función ***extend()***:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping.extend(fruitshop)

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

8.1.9 Modificar una lista

Del mismo modo que se accede a un elemento utilizando su índice, también podemos modificarlo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'
>>> shopping[0] = 'Zumo'

>>> shopping
['Zumo', 'Huevos', 'Aceite']
```

8.1.10 Modificar con troceado

No sólo es posible modificar un elemento de cada vez, sino que podemos asignar valores a trozos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[1:4]
['Huevos', 'Aceite', 'Sal']
>>> shopping[1:4] = ['Atún', 'Pasta']

>>> shopping
['Agua', 'Atún', 'Pasta', 'Limón']
```

8.1.11 Borrar elementos

Python nos ofrece, al menos, cuatro formas para borrar elementos en una lista:

1. **Por su índice**. Mediante la sentencia ***del***:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> del shopping[3]

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

2. **Por su valor.** Mediante la función ***remove()***:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.remove('Sal')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

3. **Por su índice (con extracción).** La sentencia *del* y la función *remove*()borran el elemento indicado de la lista, pero no devuelven nada. Sin embargo, Python nos ofrece la función ***pop()*** que, además de borrar, nos recupera el elemento; algo así como una extracción. Lo podemos ver como una combinación de *acceso + borrado*:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> product = shopping.pop()  # shopping.pop(-1)
>>> product
'Limón'

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal']

>>> product = shopping.pop(2)
>>> product
'Aceite'

>>> shopping
['Agua', 'Huevos', 'Sal']
```

4. **Por su rango:** Mediante troceado de listas:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[1:4] = []

>>> shopping
['Agua', 'Limón']
```

8.1.12 Borrado completo de la lista

Python nos ofrece, al menos, dos formas para borrar una lista por completo:

1. Utilizando la función **clear()**:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.clear()  # Borrado in-situ

shopping
[]
```

2. Reinicializando la lista a vacío con **[]:**

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping = []  # Nueva zona de memoria

shopping
[]
```

8.1.13 Encontrar un elemento

Si queremos descubrir el índice que corresponde a un determinado valor dentro la lista podemos usar la función ***index()*** para ello:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.index('Huevos')
1
```

Debemos tener en cuenta que, si el elemento que buscamos no está en la lista, obtendremos un error del tipo "***ValueError***":

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.index('Pollo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'Pollo' is not in list
```

8.1.14 Pertenencia de un elemento

Si queremos comprobar la existencia de un determinado elemento en una lista, podríamos buscar su índice, pero la forma *pitónica* de hacerlo es utilizar el operador «***in***», que siempre nos devolverá *True* o *False*:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

'Aceite' in shopping
True

'Pollo' in shopping
False
```

8.1.15 Obtener la longitud de una lista

Podemos conocer el número de elementos que tiene una lista con la función ***len():***

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

len(shopping)
5
```

8.1.16 Obtener el número de ocurrencias

Para contar cuántas veces aparece un determinado valor dentro de una lista podemos usar la función ***count():***

```
sheldon_greeting = ['Penny', 'Penny', 'Penny']

sheldon_greeting.count('Howard')
0

sheldon_greeting.count('Penny')
3
```

8.1.17 Dividir una cadena de texto en lista

Una tarea muy común al trabajar con cadenas de texto es dividirlas por algún tipo de separador. En este sentido, Python nos ofrece la función ***split(),*** que debemos usar anteponiendo el «*string*» que queramos dividir:

```
proverb = 'No hay mal que por bien no venga'
proverb.split()
['No', 'hay', 'mal', 'que', 'por', 'bien', 'no', 'venga']

tools = 'Martillo,Sierra,Destornillador'
tools.split(',')
['Martillo', 'Sierra', 'Destornillador']
```

Como vemos en el ejemplo, si no se especifica un separador, *split()* usa por defecto cualquier secuencia de espacios en blanco, tabuladores y saltos de línea.

La función *split*() devuelve una lista donde cada elemento es una parte de la cadena de texto original:

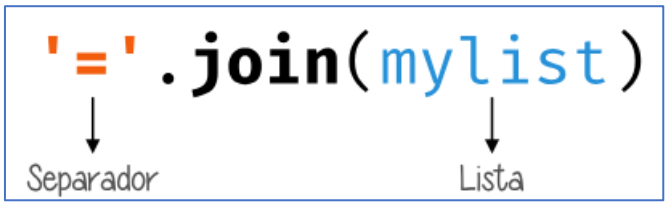
```
game = 'piedra-papel-tijera'

type(game_tools := game.split('-'))
list

game_tools
['piedra', 'papel', 'tijera']
```

8.1.18 Unir una lista en cadena de texto

Dada una lista, podemos convertirla a una cadena de texto, uniendo todos sus elementos mediante algún separador. Para ello hacemos uso de la función ***join()*** con la siguiente estructura:



*Estructura de llamada a la función **join()***

Veamos un ejemplo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> ','.join(shopping)
'Agua,Huevos,Aceite,Sal,Limón'

>>> ' '.join(shopping)
'Agua Huevos Aceite Sal Limón'

>>> '|'.join(shopping)
'Agua|Huevos|Aceite|Sal|Limón'
```

8.1.19 Ordenar una lista

Python proporciona, al menos, dos formas de ordenar los elementos de una lista:

1. **Conservando lista original**. Mediante la función ***sorted***() que devuelve una nueva lista ordenada:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

sorted(shopping)
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

2. **Modificando la lista original**. Mediante la función ***sort():***

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

shopping.sort()

shopping
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

Ambos métodos admiten un parámetro «booleano» reverse para indicar si queremos que la ordenación se haga en sentido inverso:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

sorted(shopping, reverse=True)
['Sal', 'Limón', 'Huevos', 'Agua', 'Aceite']
```

8.1.20 Iterar sobre una lista

Al igual que hemos visto con las cadenas de texto y los rangos numéricos, también podemos iterar sobre los elementos de una lista utilizando la sentencia ***for***:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

for product in shopping:
    print(product)

Agua
Huevos
Aceite
Sal
Limón
```

8.1.21 Iterar usando enumeración

Hay veces que no sólo nos interesa «visitar» cada uno de los elementos de una lista, sino que también queremos saber su índice dentro de la misma. Para ello Python nos ofrece la función ***enumerate***():

```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

for i, product in enumerate(shopping):
    print(i, product)

0 Agua
1 Huevos
2 Aceite
3 Sal
4 Limón
```

8.1.22 Iterar sobre múltiples listas

Python ofrece la posibilidad de iterar sobre múltiples listas en paralelo utilizando la función ***zip().*** Se basa en ir uniendo ambas listas elemento a elemento:

```
shopping = ['Agua', 'Aceite', 'Arroz']
details = ['mineral natural', 'de oliva virgen', 'basmati']

for product, detail in zip(shopping, details):
    print(product, detail)

Agua mineral natural
Aceite de oliva virgen
Arroz basmati
```

8.2 Tuplas

Las tuplas son colecciones ordenadas e ***inmutables*** (las listas son mutables). Al igual que las listas, pueden contener diferentes tipos de datos.

Ejemplo:

```
colores = ("rojo", "verde", "azul")

for color in colores:

    print(color)
```

Salida:

rojo

verde

azul

8.2.1 Crear tuplas

Podemos pensar en crear tuplas tal y como lo hacíamos con listas, pero usando **paréntesis** en lugar de corchetes:

```
empty_tuple = ()

three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')
```

Es posible también, crear las tuplas sin paréntesis:

```
three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'
```

Las tuplas, al ser inmutables, **no pueden modificarse** por lo que, si se intenta, no dará un error del tipo ***TypeError***:

```
three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'

three_wise_men[0] = 'Tom Hanks'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

8.2.2 Convertir un iterable en tupla

Para convertir otros tipos de datos en una tupla podemos usar la función ***tuple():***

```
shopping = ['Agua', 'Aceite', 'Arroz']

tuple(shopping)
('Agua', 'Aceite', 'Arroz')
```

Esta conversión es válida para aquellos tipos de datos que sean iterables: cadenas de caracteres, listas, diccionarios, conjuntos, etc. Un ejemplo que no funcionaría sería el intentar convertir un número en una tupla.

8.2.3 Otras operaciones con tuplas

Con las tuplas se pueden realizar muchas de las operaciones que se han visto con las listas, salvo las que conllevan una modificación de la misma. Por lo tanto, podríamos utilizar:

- ***reverse()***
- ***append()***
- ***extend()***
- ***remove()***
- ***clear()***
- ***sort()***

8.3 Diccionarios

Los diccionarios almacenan pares clave-valor y permiten iterar sobre sus claves, valores o ambos.

Ejemplo:

```
edades = {"Ana": 25, "Luis": 30, "María": 22}

for nombre, edad in edades.items():

    print(f"{nombre} tiene {edad} años")
```

Salida:

Ana tiene 25 años

Luis tiene 30 años

María tiene 22 años

Los diccionarios en Python tienen las siguientes características:

- Mantienen el orden en el que se insertan las claves.
- Son **mutables**, con lo que admiten añadir, borrar y modificar sus elementos.
- Las **claves deben ser únicas**. A menudo se utilizan las cadenas de texto como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
- Tienen un acceso muy rápido a sus elementos, debido a la forma en la que están implementados internamente.

8.3.1 Crear diccionarios

Para crear un diccionario se usan **llaves {}** rodeando asignaciones ***clave: valor*** que están separadas por comas.

En el siguiente código se observa la creación de un diccionario vacío, otro donde sus claves y sus valores son cadenas de texto y otro, donde las claves y los valores son valores enteros.

```
empty_dict = {}

rae = {
    'bifronte': 'De dos frentes o dos caras',
    'anarcoide': 'Que tiende al desorden',
    'montuvio': 'Campesino de la costa'
}

population_can = {
    2015: 2_135_209,
    2016: 2_154_924,
    2017: 2_177_048,
    2018: 2_206_901,
    2019: 2_220_270
}
```

8.3.2 Convertir iterables en diccionarios

Para convertir otros tipos de datos en un diccionario podemos usar la función ***dict():***

```
# Diccionario a partir de una lista de cadenas de texto
dict(['a1', 'b2'])
{'a': '1', 'b': '2'}

# Diccionario a partir de una tupla de cadenas de texto
dict(('a1', 'b2'))
{'a': '1', 'b': '2'}

# Diccionario a partir de una lista de listas
dict([['a', 1], ['b', 2]])
{'a': 1, 'b': 2}
```

Es posible utilizar la función *dict*() para crear diccionarios sin tener que utilizar llaves y comillas. Para ello, se deben pasar la clave y el valor como **argumentos** de la función:

```
>>> person = dict(
     name='Guido',
     surname='Van Rossum',
     job='Python creator'
)

>>> person
{'name': 'Guido', 'surname': 'Van Rossum', 'job': 'Python creator'}
```

8.3.3 Obtener un elemento

Para obtener un elemento de un diccionario basta con escribir la clave entre corchetes. Veamos un ejemplo:

```
rae = {
    'bifronte': 'De dos frentes o dos caras',
    'anarcoide': 'Que tiende al desorden',
    'montuvio': 'Campesino de la costa'
}

rae['anarcoide']
'Que tiende al desorden'
```

Si intentamos acceder a una clave que no existe, obtendremos un error ***KeyError***.

Para evitar este tipo de errores se puede utilizar la función ***get()*** que, si no encuentra la clave devuelve *None*. Veámoslo con un ejemplo:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}

>>> rae.get('bifronte')
'De dos frentes o dos caras'

>>> rae.get('programación')

>>> rae.get('programación', 'No disponible')
'No disponible'
```

8.3.4 Añadir o modificar un elemento

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la clave y asignarle un valor:

- Si la **clave ya existía** en el diccionario, se **modifica** el valor existente por el nuevo.
- Si la **clave es nueva**, se **añade** al diccionario con su valor. No vamos a obtener un error a diferencia de las listas.

Utilizando el mismo diccionario de ejemplos anteriores, veamos cómo se añadiría y modificaría un elemento al diccionario:

- **Añadir:**

```
>>> rae['enjuiciar'] = 'Someter una cuestión a examen, discusión y
juicio'

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'}
```

- **Modificar:**

```
>>> rae['enjuiciar'] = 'Instruir, juzgar o sentenciar una causa'

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

8.3.5 Utilizar un patrón de creación de diccionarios

Una forma muy habitual de trabajar con diccionarios es utilizar el **patrón creación** partiendo de uno vacío e ir añadiendo elementos poco a poco.

Supongamos un ejemplo en el que queremos construir un diccionario donde las claves son las letras vocales y los valores son sus posiciones:

```
VOWELS = 'aeiou'

enum_vowels = {}

for i, vowel in enumerate(VOWELS, start=1):
    enum_vowels[vowel] = i


enum_vowels
{'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5}
```

8.3.6 Comprobar si una clave existe

La forma *pitónica* de comprobar la existencia de una clave dentro de un diccionario, es utilizar el operador «***in***»:

```
>>> 'bifronte' in rae
True

>>> 'almohada' in rae
False

>>> 'montuvio' not in rae
False
```

8.3.7 Obtener la longitud

Podemos conocer el número de elementos («clave-valor») que tiene un diccionario con la función ***len()***:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}

>>> len(rae)
4
```

8.3.8 Obtener todos los elementos

Python ofrece mecanismos para obtener todos los elementos de un diccionario. Partimos del siguiente diccionario:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

1. **Obtener todas las claves de un diccionario.** Mediante la función ***keys***():

```
>>> rae.keys()
dict_keys(['bifronte', 'anarcoide', 'montuvio', 'enjuiciar'])

Obtener todos los valores de un diccionario. Mediante la función
values():
>>> rae.values()
dict_values([
    'De dos frentes o dos caras',
    'Que tiende al desorden',
    'Campesino de la costa',
    'Instruir, juzgar o sentenciar una causa'
])
```

2. **Obtener todos los valores de un diccionario:** Mediante la función ***values***():

```
>>> rae.values()
dict_values([
    'De dos frentes o dos caras',
    'Que tiende al desorden',
    'Campesino de la costa',
    'Instruir, juzgar o sentenciar una causa'
])
```

3. **Obtener todos los pares «clave-valor» de un diccionario.** Mediante la función ***items***():

```
>>> rae.items()
dict_items([
    ('bifronte', 'De dos frentes o dos caras'),
    ('anarcoide', 'Que tiende al desorden'),
    ('montuvio', 'Campesino de la costa'),
    ('enjuiciar', 'Instruir, juzgar o sentenciar una causa')
])
```

8.3.9 Borrar elementos

Python nos ofrece, al menos, tres formas para borrar elementos en un diccionario:

1. **Por su clave.** Mediante la sentencia ***del***:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> del rae['bifronte']

>>> rae
{'anarcoide': 'Que tiende al desorden', 'montuvio': 'Campesino de la
costa'}
```

2. **Por su clave (con extracción):** Mediante la función ***pop***() podemos extraer un elemento del diccionario por su clave. Vendría a ser una combinación de *get() + del*:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae.pop('anarcoide')
'Que tiende al desorden'

>>> rae
{'bifronte': 'De dos frentes o dos caras', 'montuvio': 'Campesino de
la costa'}
```

3. **Borrado completo del diccionario.** Utilizando la función **clear():**

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae.clear()

>>> rae
{}
```

8.3.10 Iterar sobre un diccionario

En base a los elementos que podemos obtener, Python nos proporciona tres maneras de iterar sobre un diccionario.

1. **Iterar sobre claves:**

```
>>> for word in rae.keys():
...     print(word)
...
bifronte
anarcoide
montuvio
enjuiciar
```

2. **Iterar sobre valores:**

```
>>> for meaning in rae.values():
...     print(meaning)
...
De dos frentes o dos caras
Que tiende al desorden
Campesino de la costa
Instruir, juzgar o sentenciar una causa
```

3. **Iterar sobre «clave-valor»:**

```
>>> for word, meaning in rae.items():
...     print(f'{word}: {meaning}')
...
bifronte: De dos frentes o dos caras
anarcoide: Que tiende al desorden
montuvio: Campesino de la costa
enjuiciar: Instruir, juzgar o sentenciar una causa
```

8.4 Conjuntos (Sets)

Como estamos viendo, estructuras iterables hay muchas y diversas. De hecho, a lo largo del curso y en este mismo tema, ya se ha trabajado con otros tipos de estructuras iterables, como son las cadenas (*strings*) y los rangos numéricos (*range*) y, en este apartado, nos centraremos en una última estructura de datos, los conjuntos.

Los **conjuntos** son colecciones **desordenadas y no permiten elementos duplicados**. Mantiene muchas similitudes con el concepto matemático de conjunto.

Ejemplo:

```
numeros = {4, 2, 3, 1}

for numero in numeros:

    print(numero)
```

Salida:

1

2

3

4

Nótese que, aunque el conjunto se ha definido con sus elementos desordenados, al recorrerlos mediante un bucle *for*, los elementos se recorren ordenados.

Para convertir otros tipos de datos en un conjunto podemos usar la función ***set()*** sobre cualquier iterable:

```
>>> set('aplatanada')
{'a', 'd', 'l', 'n', 'p', 't'}

>>> set([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5])
{1, 2, 3, 4, 5}

>>> set(('ADENINA', 'TIMINA', 'TIMINA', 'GUANINA', 'ADENINA',
'CITOSINA'))
{'ADENINA', 'CITOSINA', 'GUANINA', 'TIMINA'}

>>> set({'manzana': 'rojo', 'plátano': 'amarillo', 'kiwi': 'verde'})
{'kiwi', 'manzana', 'plátano'}
```

En un conjunto no existe un orden establecido para sus elementos, por lo tanto, **no podemos acceder a un elemento en concreto**. De este hecho se deriva igualmente que **no podemos modificar un elemento** existente, ya que ni siquiera tenemos acceso al mismo. Python sí nos permite añadir o borrar elementos de un conjunto usando las funciones ***add()*** y ***remove(),*** respectivamente.

La forma de recorrer los elementos de un conjunto es utilizar la sentencia *for*:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> len(beatles)
4

>>> for beatle in beatles:
...     print(beatle)
...
Harrison
McCartney
Starr
Lennon
```

Al igual que con otros tipos de datos, Python nos ofrece el operador «***in***» para determinar si un elemento pertenece a un conjunto:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> 'Lennon' in beatles
True

>>> 'Fari' in beatles
False
```

8.5 Funciones para estructuras iterables

Python también ofrece funciones integradas que trabajan con iterables. Algunas de ellas ya se han visto en untos anteriores y otras no. Son las siguientes:

- **len()**: Devuelve la cantidad de elementos.
- **enumerate()**: Proporciona un índice junto con cada elemento.
- **zip()**: Combina dos o más iterables en pares.
- **map()**: Aplica una función a cada elemento del iterable.
- **filter()**: Filtra elementos de un iterable según una condición.

Ejemplo con enumerate:

```
frutas = ["manzana", "plátano", "cereza"]
for indice, fruta in enumerate(frutas):
    print(f"Fruta {indice + 1}: {fruta}")
```

Salida:

Fruta 1: manzana

Fruta 2: plátano

Fruta 3: cereza

8.6 Creación de iterables personalizados

Aunque en este módulo no se trabajará la creación de **objetos iterables**, a modo informativo cabe indicar que en Python, al igual que en otro lenguajes orientados a objetos, se pueden crear objetos iterables personalizados implementando los métodos __iter__() y __next__() en una clase.

Ejemplo:

```
class Contador:
    def __init__(self, limite):
        self.limite = limite
        self.actual = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.actual < self.limite:
            self.actual += 1
            return self.actual
        else:
            raise StopIteration

contador = Contador(5)
for numero in contador:
    print(numero)
```

9 Web para explorar

Aquí podéis consultar algunas Web con las que podréis profundizar en los concetos vistos en esta Unidad de trabajo.

El tutorial de Python — documentación de Python - 3.13.0

Operadores Básicos en Python con ejemplos

Tupla en Python - Aprende cómo crear, acceder y agregar elementos

Diccionarios en Python - Blog de Código Facilito