Sesión 3. Funciones. Listas y Diccionarios

Este documento forma parte del curso **Programación con Python** del CEFIRE CTEM.

Autoría: José Luis Tomás Navarro
Revisión: Mª Paz Segura Valero

ÍNDICE

1.- Funciones

Las **funciones en Python constituyen un elemento similar** a los **subprogramas en la *Programación Estructurada***.

Estos elementos nos van a permitir dividir un problema en partes más pequeñas, resolver cada una de estas partes y luego unirlas para resolver un problema mayor. También **recordemos que se introduce la idea de reutilización de código**, puesto que, una vez definida la función puede ser invocada tantas veces como necesitemos hacerlo. Además, también se organiza y estructura mejor el código de los programas, sobre todo cuanto más grandes son estos.

Las **funciones se pueden definir en cualquier parte de un programa**, aunque por estandarización y organización **vamos a tomar la decisión de situarlas siempre al principio de los programas**, antes de la primera instrucción del programa principal.

La sintaxis para definir una función la tenemos a continuación:

```
def nombre_funcion([parametros]):
    Bloque instrucciones de la función
    [return]
```

Ejemplos:

```
def visualiza_mensaje():
    print("Mensaje dentro texto de la función")
    return

visualiza_mensaje()
print("Programa terminado")
```

En el programa principal se llama a la función **visualiza_mensaje()**, que símplemente visualiza un mensaje de texto.

En este ejemplo la función suma dos variables que han sido inicializadas dentro de la misma. Como vemos no contiene la palabra return, puesto que, si la función no devuelve ningún valor esta palabra es opcional.

```
def suma_numeros():
        a = 2
        b = 5
        c = a + b
        print(c)
c = 15
suma_numeros()
print(c)
```

El resultado de la ejecución es:

```
7
15
```

Podemos fijarnos que **existe una variable con el nombre c** en el **programa principal** y **también otra variable c en la función. Esto nos lleva a preguntarnos si son la misma variable**. **La respuesta** a esta pregunta **es NO**, puesto que están en ámbitos de memoria diferentes. Además **podemos ver como el contenido de la variable c es diferente (7 en la función y 15 en el programa principal)**. **Este concepto lo abordaremos** un poco más abajo, **cuando hablemos del ámbito de las variables**.

Ámbito de las variables

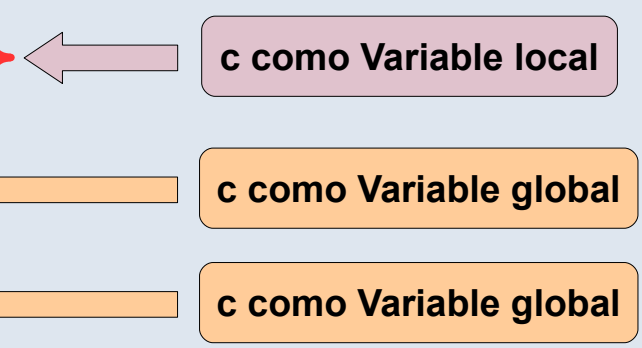
Para resolver el problema de los conflictos de nombres, que aparecen en los subprogramas o funciones, tal y como vimos en el ejemplo de la función suma_numeros, **los lenguajes de programación limitan lo que se llama el alcance o el ámbito de las variables**.

Es decir, **se permite que una variable exista únicamente en el interior de una función** y **no afecte a otras variables de mismo nombre situadas fuera de la función**, porque en estos casos el ámbito de las variables es diferente.

Recordemos el contenido de la **función suma_numeros()**:

```
def suma_numeros():
    a = 2
    b = 5
    c = a + b
    print(c)

c = 15
suma_numeros()
print(c)
```



El **ámbito de la variable c dentro de suma_numeros() es únicamente la función suma_numeros()**. A estas variables se les llama **variables locales**.

En cambio, **el ámbito de la variable c inicializada en el programa principal es únicamente el programa principal.** A estas variables se les llama **variables globales**.

Esta es la razón por la que las variables son consideradas diferentes.

En ocasiones nos puede interesar que una función pueda acceder y modificar las variables globales situadas en el programa principal. El mecanismo que proporciona Python **para poder hacer esto** es **declarar la variable en la función como global**, con la siguiente sintaxis: **global nombre_variable**

Veamos el mismo ejemplo de la función **suma_numeros()**, pero cambiando algunas cosas:

```
def suma_numeros():
        global c
        print(c)
        a = 2
        b = 5
        c = a + b

c = 15
suma_numeros()
print(c)
```

c como Variable global

c como Variable global

c como Variable global

c como Variable global

El resultado de la ejecución sería:

```
15
7
```

Si hacemos referencia a una variable, por ejemplo, en una instrucción condicional **if** o en una instrucción **print()** antes de que ésta sea inicializada con un valor, se producirá un error. Esto sucede independientemente de si estamos en el ámbito de una variable local a una función o en el ámbito de una variable global del programa principal.

A continuación podemos ver un ejemplo:

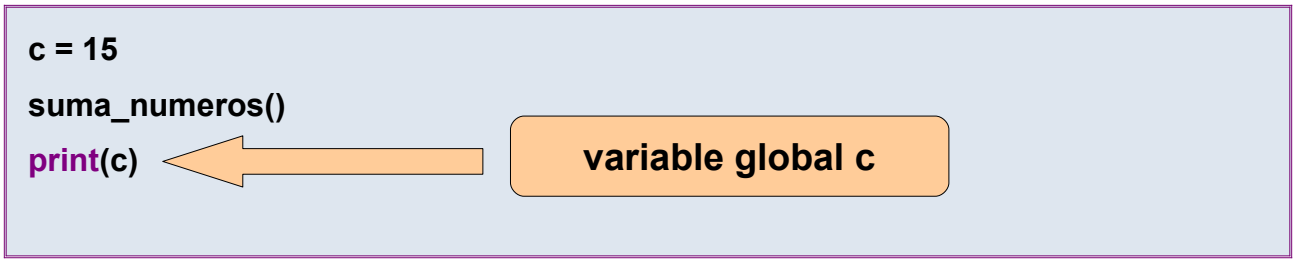
```
def suma_numeros():
        if c>0:
            print("Mayor de 0")
        a = 2
        b = 5
        c = a + b

print(c)
```

variable no inicializada

variable local c

variable no inicializada



En este caso se producirían dos errores: uno en la función al intentar consultar el valor de la **variable c** (instrucción condicional **if c>0:**) y otro en el programa principal al intentar visualizar el valor de la **variable c**.

Paso de Parámetros

Las **funciones, opcionalmente, pueden contener parámetros,** indicados entre paréntesis y separados por comas.

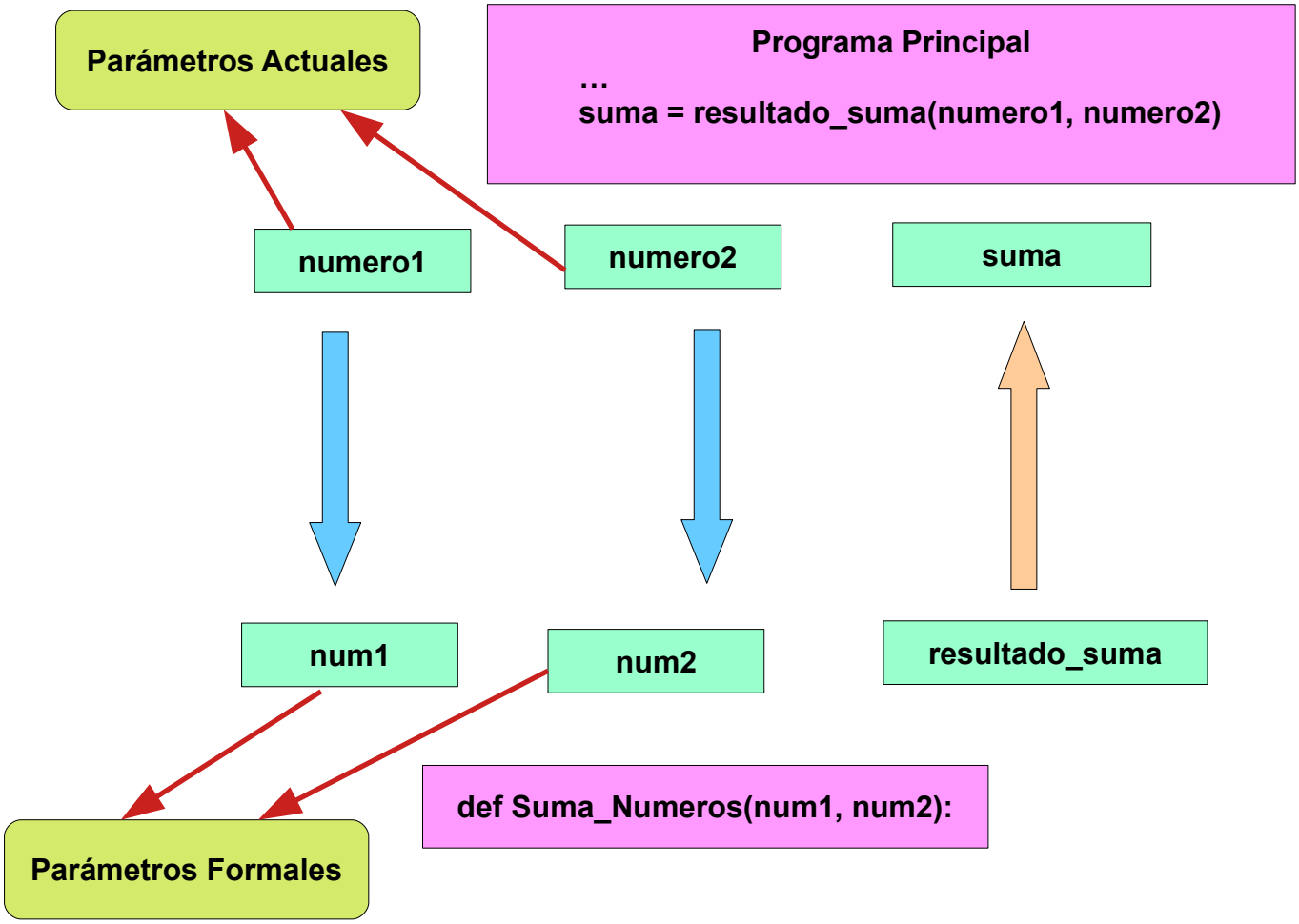
Además, puede aparecer en el cuerpo de instrucciones de la función la palabra **return**. Esta palabra **se suele poner al final de las instrucciones de la función aunque su uso sólo es obligatorio** cuando la función tenga que **devolver algún valor**.

Tradicionalmente, los lenguajes de programación distinguen entre **parámetros actuales** y **parámetros formales**. Los parámetros actuales son las variables del programa principal que actúan como parámetros y que nos permiten invocar a la función mientras que los parámetros formales se corresponden con los nombres de las variables que ejercen de parámetros en la función.

Tiene que haber un **acoplamiento perfecto entre los parámetros actuales y formales**, en el sentido de que **el número y tipo de datos de los mismos tiene que coincidir**. No ocurre lo mismo con el nombre de las variables, es decir, el nombre de los parámetros actuales y el nombre de los parámetros formales puede ser diferente aunque, lo cierto es que lo más sencillo es que sean iguales si no queremos complicarnos la vida.

En la siguiente figura, podemos ver el acoplamiento que existe en este caso entre los dos parámetros del programa principal y los de la función vista anteriormente.

La función calculará el resultado de la suma (resultado_suma) y lo devolverá al programa principal con la instrucción **return resultado_suma**.



En este caso vemos que **el nombre de los dos parámetros actuales** (**numero1 y numero2**) **no coincide con el de los dos primeros parámetros formales** (**num1 y num2**).

Cuando desde el programa principal se invoca a una función, **los valores existentes en los parámetros actuales se copian a los parámetros formales**, para que la función pueda conocer los datos con los que debe trabajar.

El **espacio de memoria de los parámetros actuales y los parámetros formales es diferente**. Por eso, **si en la función se modifica el valor de las variables asociadas a los parámetros formales, no afectará al valor almacenado en las variables que representan los parámetros actuales**.

Dependiendo de la naturaleza de la función, **en el caso de que se tenga que devolver algún valor, se incluirá en el cuerpo de la función la palabra return nombre_variable, para que el programa principal sea conocedor del valor devuelto por la función.**

En el siguiente esquema aparece la sintaxis empleada a la hora de especificar la lista de parámetros, junto con la declaración de la función:

def **nombre_función** **(param1, param2, …, param N):**

Los parámetros en Python siempre son de entrada, es decir, se copia el valor de la variable que representa el parámetro actual en la variable existente en la definición de la función (parámetro formal). Recordemos que en otros lenguajes de programación sí que existen parámetros de salida.

Python presenta una alternativa a los parámetros de salida. Se trata de **utilizar la palabra return junto con el nombre de una variable** para **indicar que se devuelve un valor al programa principal**.

Podemos ver el ejemplo modificado de suma_numeros(), para entenderlo mejor.

```
def suma_numeros(x, y):
    resultado = x + y
    return resultado

a = 3
b = 5
suma = suma_numeros(a, b)
print("La suma de", a, "y", b, "es:", suma)
```

Podemos ver que el programa principal tiene que recoger de alguna forma la variable devuelta por la función, en este caso la almacena en la variable **suma** (instrucción **suma = suma_numeros(a, b)**). También podíamos haber situado la llamada a la función directamente en la instrucción print(), por ejemplo:

```
print("La suma de", a, "y", b, "es:", suma_numeros(a, b))
```

Podemos pensar que como las variables del programa principal son accesibles por la funciones, en principio **no nos haría falta pasar parámetros a las funciones**. Sin embargo, **al utilizar los parámetros conseguimos que las funciones sean mucho más portables y el programa quede mucho más claro y estructurado**.

Evidentemente, la función **suma_numeros()** es muy trivial pero imaginemos que fuera mucho más compleja y quisiéramos reutilizarla en otros programas. Si la función no tuviera parámetros, la línea que calcula el resultado utilizaría las variables del programa principal, quedando por ejemplo:

```
...
resultado = a + b
```

Si tuviésemos otro programa principal que utilizara la función **suma_numeros()** y, en lugar de utilizar las **variables a y b** usase **numero1 y numero2**, la función daría un error y no funcionaría porque no conocería lo que son las **variables a y b**.

Return con varios datos

Aunque en principio pueda resultar un poco extraño, **Python cuenta con la posibilidad** de que **una función puede retornar más de un valor**.

Ejemplo:

```
def funcion():
...
return (a,b)
```

En realidad, aunque aparezca más de un valor en la palabra return, **realmente todos estos valores son tratados por Python como una única variable que es considerada una tupla**. Las tuplas son un tipo de datos estructurados parecidos a la listas. Pero, a diferencia de las listas, las **tuplas son inmutables. Esto quiere decir que una vez que se definen sus elementos, estos ya no pueden variar.**

Lógicamente, **la variable del programa principal** que recoja el resultado de la llamada a la función, **será una tupla**, **donde se almacenarán los valores devueltos**.

A continuación se muestra **un ejemplo completo**, en el cual **se devuelven 3 valores en una tupla al programa principal**, el cual los mostrará utilizando la estructura **for**:

```
def introduce_lados_triangulo()
    lado1 = int(input("Introduce el primer lado"))
    lado2 = int(input("Introduce el segundo lado"))
    lado3 = int(input("Introduce el tercer lado"))
    return (lado1,lado2,lado3)

lados_triangulo = introduce_lados_triangulo()
for i in range(len(lados_triangulo)):
    print("Lado",i+1,lados_triangulo[i])
```

Actividades Propuestas Voluntarias – Funciones

Actividad Calculadora

Se trata de simular mediante funciones, las cuatro operaciones aritméticas básicas: suma, resta, multiplicación y división.

En primer lugar el programa solicitará por pantalla dos números enteros (**numero1 y numero2**). A continuación, se solicitará al usuario que indique la operación que desea realizar, para lo cual el usuario introducirá por teclado **un carácter**.

Los valores posibles serán los siguientes:

LETRA	SIGNIFICADO
'+'	Suma
'-'	Resta
'*'	Multiplicación
'/'	División

Una vez que tenemos los operandos sobre los que hay que realizar la operación y cual es la operación a realizar, **el programa llamará a la función que calculará la operación en cuestión**. Ésta devolverá el cálculo al **programa principal**, quien **será el que muestre el resultado de la operación que se ha solicitado**.

Los nombres de las funciones que podemos utilizar, junto con los dos parámetros de entrada se muestran a continuación:

- **calcula_suma(numero1, numero2)**
- **calcula_resta(numero1, numero2)**
- **calcula_multiplicacion(numero1, numero2)**
- **calcula_division(numero1, numero2)**

Por ejemplo, si el usuario quiere **calcular la multiplicación de los números 5 y 3**, el mensaje que **tendrá que visualizar** el programa principal, será:

5 * 3 es igual a: 15

Hay que aclarar que la **función calcula_division(numero1, numero2) devolverá un dato de tipo tupla**. **formado por dos valores**. La composición de la tupla será la siguiente:

- Primer elemento, si se puede realizar la división => **division_posible (booleano)**

- Segundo elemento, el resultado de la división => **resultado_division (float)**, este último valor valdrá 0 en el caso de que no se pueda realizar la división

De manera que la función **calcula_division(numero1, numero2)** comprobará **si el divisor es distinto de 0** para poder realizar la división. Si es así, asignará el valor **cierto**

a division_posible y el resultado obtenido a resultado_division. En caso contrario, se asignará el valor **falso** a **division_posible** y un **cero** a **resultado_division**.

El nombre del archivo podría ser: **calculadora.py**.

Actividad Calculo Áreas Geométricas

Esta actividad es una variación de la que vimos como actividad obligatoria en la sesión anterior. La diferencia está en que, ahora, para calcular el área de cada una de las figuras emplearemos una función.

Los nombres de las funciones junto con los parámetros y la fórmula que hay que emplear para calcular el área son, respectivamente:

- **calcula_area_cuadrado(lado) => lado * lado**
- **calcula_area_circulo(radio) => Π * r^2**
- **calcula_area_triangulo(base, altura) => (base * altura) / 2**

Las funciones devolverán el cálculo al programa principal, que será el que muestre el resultado del área calculada.

Recordemos que es recomendable que la definición de estas tres funciones estén al principio del programa, después de la zona de **imports** y antes del programa principal.

Ahora desde el programa principal, cada vez que el usuario elija un tipo de figura, llamaremos a la función correspondiente y le pasaremos los parámetros adecuados.

Por ejemplo:

```
area_triangulo = calcula_area_triangulo(base, altura)
```

El nombre que tendrá el programa será:**calculo_areas_funciones.py**

2.- Listas

Las listas son un tipo de datos estructurados q**ue contienen un conjunto de elementos ordenados y mutables**. Por mutables entendemos que, una vez que definimos una lista con un número determinado de elementos, vamos a poder añadir o eliminar elementos a la lista.

Una característica destacable de las listas en Python es que **no es necesario que todos los elementos sean del mismo tipo**.

Al igual que con las cadenas, el operando que nos permite añadir o concatenar elementos es la suma **(+)**. Este operando necesita que los dos operandos sean listas, según podemos ver en los siguientes ejemplos:

```
#Primer ejemplo
vocales = [‘a’,’e’]
vocales = vocales + [‘i’]

#Segundo ejemplo
lista = [ ]
lista = lista + [12,14]
```

En el primer ejemplo, añadimos la **vocal i** a la lista **vocales** que ya tenía dos elementos, las **vocales a** y **e**.

En el segundo ejemplo, tenemos una lista inicialmente **vacía** a la que añadimos la lista formada por los **elementos 12** y **14**.

Si, en lugar de concatenar una lista con otra, intentásemos añadir directamente un elemento que no es una lista, Python daría un error. Por ejemplo:

```
vocales = [‘a’,’e’,’i’]
vocales = vocales + ‘o’
```

En los primeros ejemplos hemos añadido elementos por la parte de detrás de una lista pero también podemos añadirlos por delante, tal y como podemos ver aquí:

```
vocales = ['e','i']
vocales = ['a'] + vocales
```

Python ofrece el **método append()** para **añadir elementos al final de una lista**. Esto puede resultar **útil para automatizar determinados procesos** en los que, por ejemplo, tengamos el elemento que queremos añadir almacenado en una variable.

Ejemplo:

```
nombre = input("Introduce un nombre:")
lista_nombres.append(nombre)
```

La forma de manipular los elementos individuales en una lista es exactamente igual a como lo hacíamos con las cadenas. Recordemos que utilizábamos un índice para acceder a cada uno de los elementos individuales y que **el índice del primer elemento es el 0**.

Ejemplo:

```
print(vocales[0])
```

El resultado sería:

```
a
```

Ejemplo:

```
if vocales[indice] == 'e':
...
```

En el ejemplo de arriba, comprobamos si el elemento de la lista apuntado por la variable **índice** contiene la **vocal e**.

Manipulación de Sublistas

De una lista se pueden extraer sublistas (porciones de listas) utilizando la siguiente notación:

nombre_lista[inicio:límite]

donde **inicio y límite** juegan el mismo papel que en el tipo **range(inicio, límite)**, es decir el elemento apuntado por la posición límite no se incluye en la lista resultante.

Ejemplos:

```
dias_semana = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"]
print(dias_semana[1:3])   # Extraemos martes y miércoles
print(dias_semana[5:6])   # Extraemos el sábado
print(dias_semana[:])     # Extraemos todos los elementos
```

Los resultados serían:

```
['martes', 'miércoles']
['sábado']
['lunes', 'martes', 'miércoles', 'jueves', 'viernes', 'sabado', 'domingo']
```

También podemos acceder a un elemento de la lista para modificarlo.

Ejemplo:

```
lista_numeros_pares = [2,4,6,8,10]
lista_numeros_pares[2] = 8    #Tercer elemento de la lista
print(lista_numeros_pares)
```

La visualización por pantalla quedaría así:

```
[2,4,8,8,10]
```

Borrado de Elementos

El comando **del** nos permite borrar un elemento de la lista o varios, depende de como lo especifiquemos.

Sintaxis:

```
del nombre_lista[elemento]
del nombre_lista[elemento_inicial:limite]
del nombre_lista
```

En el primer caso estamos eliminando un elemento concreto, indicado por el índice elemento. En el segundo caso eliminamos un rango de elementos de la lista, tal y como hacíamos cuando manipulábamos sublistas. Finalmente, en el tercer caso eliminamos la lista entera.

Ejemplo:

```
...
del vocales[0]     # Eliminamos el primer elemento
del vocales[0:3]   # Eliminamos desde el primer elemento (0) hasta el tercero (2)
del vocales        # Eliminamos toda la lista
...
```

También contamos con el **método pop()** que nos permite borrar el último elemento de una lista, o bien el elemento especificado por el índice.

Ejemplos:

```
...
lista_numeros.pop()   # Borra el último elemento de la lista lista_numeros
lista_numeros.pop(2)  # Borra el elemento situado en la tercera posición (0,1,2,...)
...
```

Recorrido de una lista

Existen dos formas de recorrer los elementos de una lista:

- Primera forma: **la variable de control del bucle toma los valores de la lista que estamos recorriendo**.

  Ejemplo:

```
…
letras = ["a", "b", "c",”d”]
for contador in letras:
    print(contador, end=" ")
...
```

  La visualización sería:

```
a b c d
```

- Segunda forma: recorremos indirectamente los elementos de la lista, es decir, **la variable de control del bucle toma como valores los índices de la lista que estamos recorriendo (0,1 ,2 , etc.)**. En este caso, para acceder a los valores de la lista hay que utilizar el índice, tal y como podemos ver en el siguiente ejemplo:

```
…
letras = ["a", "b", "c",”d”]
for contador in range(len(letras)):
    print(letras[contador], end=" ")
...
```

  La **función len(lista)** devuelve el número de elementos existentes en una lista.

Pertenencia de un elemento a una lista

Al igual que ocurría con las cadenas, con el operador **in** vamos a poder determinar si un elemento pertenece a una lista o no, de una forma sencilla.

Ejemplo:

```
…
personas = ["Carlos", "María"]
nombre = input("Introduce un nombre: ")
if nombre in personas:
  print("La persona está en la lista")
else:
  print("La persona NO está en la lista")
...
```

Una variante del predicado **in** es el predicado **not in**, que niega la condición. Eso quiere decir que valdrá **True** si el elemento no pertenece a la lista, justo al contrario que el predicado **in**.

Ejemplo:

```
…
personas = ["Carlos", "María"]
nombre = “Antonio”
...
if nombre not in personas:
    print("La persona NO está en la lista")
else:
    print("La persona está en la lista")
...
```

En este caso el mensaje que se visualizaría es:

```
La persona NO está en la lista
```

puesto que Antonio no se encuentra en la lista.

En el caso de que tengamos una lista **con datos numéricos solamente**, **podemos sumarlos** sin necesidad de utilizar un bucle **for** que recorra toda la lista. Para ello utilizaremos la **función sum()** de la siguiente forma:

```
# Suponemos lista_numeros[ ] contiene una lista de datos numéricos
suma_numeros = sum(lista_numeros)
```

3.- Diccionarios

Los **Diccionarios** son un tipo de datos estructurados que se caracterizan porque permiten almacenar parejas de valores. Dichas parejas siguen el siguiente patrón:

clave : valor

Las claves pueden ser de cualquier tipo inmutable, preferiblemente cadenas de texto y números. Permiten identificar un par **clave:valor** determinado, así que deben ser únicas. Esto quiere decir que no puede haber dos pares **clave:valor** con la misma **clave** aunque sí podría darse el caso de tener dos pares **clave:valor** con idéntico **valor**.

A diferencia de las lista, cuyos elementos van encerrados entre corchetes […], los diccionarios emplean las **llaves { … } para delimitar sus elementos**. Las parejas de datos **clave : valor se separan por comas**.

En el siguiente ejemplo podemos ver con más claridad como se define un diccionario:

```
diccionario = {'nombre' : 'Juan', 'edad' : 20, 'aficiones': ['Pádel','Cine','Bici de Montaña']}
```

Como podemos observar, los datos almacenados pueden ser de cualquier tipo: cadenas de texto, números, listas, etc. Si nos fijamos en el tercer par, veremos que el valor es un dato estructurado, concretamente una lista de tres elementos (**['Pádel','Cine','Bici de Montaña']**).

La forma que tenemos para acceder a un elemento de un diccionario es a través de su clave, poniendo la misma delimitada por comillas simples y dentro de unos corchetes, según podemos ver en el siguiente ejemplo:

```
print(diccionario['nombre'])
print(diccionario['edad'])
print(diccionario['aficiones'])
```

El resultado sería:

```
Juan
20
['Pádel', 'Cine', 'Bici de Montaña']
```

Si uno de los elementos de nuestro diccionario es una lista y queremos acceder a sus elementos de forma individual, utilizaremos un índice, según podemos ver en el siguiente ejemplo:

```
...
print(diccionario['aficiones'][0])
print(diccionario['aficiones'][1])
print(diccionario['aficiones'][2])
...
```

El resultado sería:

```
Pádel
Cine
Bici de Montaña
```

Para recorrer todo un diccionario podemos hacer uso de la estructura repetitiva **for**. A continuación podemos ver un ejemplo:

```
...
for key in diccionario:
    print(key, ":", diccionario[key])
...
```

El resultado sería:

```
nombre : Juan
edad : 20
aficiones : ['Pádel', 'Cine', 'Bici de Montaña']
```

A continuación vamos a ver algunos de los métodos que podemos utilizar en los diccionarios:

=> dict(<representación de un diccionario>)

Recibe como parámetro la representación de un diccionario y, si es posible, devuelve un diccionario de datos.

Ejemplo:

```
diccionario_datos = dict(nombre='Pedro', altura=1.8, peso=70)
# Obtendremos: diccionario_datos => {'nombre':'Pedro',  'altura':1.8, 'peso':70}
```

=> keys()

Devuelve una lista de elementos con las claves del diccionario de datos.

Ejemplo:

```
diccionario_datos = {'nombre':'Pedro',  'altura':1.8, 'peso':70}
claves = diccionario_datos.keys()
# Obtendremos: claves => ['nombre','altura','peso']
```

=> values()

Devuelve una lista de elementos con las valores del diccionario de datos.

Ejemplo:

```
diccionario_datos = {'nombre':'Pedro',  'altura':1.80, 'peso':70}
valores = diccionario_datos.values()
# Obtendremos: valores => ['Pedro',1.8,70]
```

=> clear()

Elimina todos los ítems del diccionario y lo deja vacío.

Ejemplo:

```
diccionario_datos = {'nombre':'Pedro',  'altura':1.8, 'peso':70}
diccionario_datos.clear()
# Obtendremos: diccionario_datos => { }
```

=> get('clave')

Recibe como parámetro una clave y devuelve el valor de dicha clave. Si no la encuentra entonces devuelve un objeto **none**.

Ejemplo:

```
diccionario_datos = {'nombre':'Pedro',  'altura':1.8, 'peso':70}
valor = diccionario_datos.get('nombre')
print(valor)
```

Obtendremos:

```
Pedro
```

=> pop('clave')

Recibe como parámetro una clave, elimina el par clave:valor del diccionario asociado a dicha clave y devuelve su valor. Si no lo encuentra, entonces devuelve error.

Ejemplo:

```
diccionario_datos = {'nombre':'Pedro',  'altura':1.8, 'peso':70}
valor = diccionario_datos.pop('peso')
# El diccionario resultate quedaría:
diccionario_datos => {'nombre':'Pedro',  'altura':1.8}
```

=> setdefault('clave', 'valor')

Permite buscar una **clave** en un diccionario. Si la encuentra, devuelve el **valor** asociado. Si no la encuentra entonces agrega un nuevo elemento al diccionario formado por **clave:valor**.

Ejemplo:

```
diccionario = {'nombre':'Pedro',  'altura':1.8, 'peso':70}
diccionario.setdefault('poblacion', 'Valencia')
# El diccionario resultante sería:
#     diccionario => {'nombre':'Pedro', 'altura':1.8, 'peso':70, 'poblacion':'Valencia}
```

=> update(diccionario)

Permite fusionar dos diccionario. Recibe como parámetro el diccionario que quiere añadir al diccionario original. Si hay claves iguales, actualiza el **valor** correspondiente a la clave repetida con el valor del diccionario pasado como parámetro. Las claves nuevas, son añadidas al diccionario directamente.

Ejemplo:

```
diccionario = {'nombre':'Pedro',  'altura':1.8, 'peso':70}
diccionario_dos = {'nombre':'Antonio',  'poblacion':'Valencia'}
diccionario_datos.update(diccionario_datos_dos)
# El diccionario resultante sería:
# diccionario => {'nombre':'Antonio',  'altura':1.8, 'peso':70, 'poblacion':'Valencia}
```