4 BUCLES

Con frecuencia, las condiciones se deberán **comprobar de forma repetida.** Estas estructuras repetitivas se conocen con el nombre de ***bucles.***

Python ofrece varias alternativas para plantear programas que las empleen, como la orden **while** y la orden **for**.

4.1. while

Si se desea que una **sección** de un programa **se repita mientras** se cumpla una cierta condición, se deberá emplear **while** en lugar de **if**.

```
# Repetición con while: pedir un número positivo

numero = int ( input("Escribe un numero positivo: ") )
while numero <= 0:
  print ("Ese numero no es positivo.")
  numero = int ( input("Escribe otro numero positivo: ") )
```

```
Escribe un numero positivo: -5
Ese numero no es positivo.
Escribe otro numero positivo: -2
Ese numero no es positivo.
Escribe otro numero positivo: 6
```

4.2. Contadores

Se puede usar la orden **while** para crear un **contador,** esto es, una **variable** que aumenta de un valor a otro, usado para descubrir cuántas veces ha ocurrido algo:

```
# Contador con while

numero = 1
while numero <= 10:
  print (numero , " ", end="")
  numero = numero + 1
```

```
1 2 3 4 5 6 7 8 9 10
```

La última línea del programa anterior puede resultar desconcertante. Conviene recordar que una orden, como **numero = numero + 1**, no es una igualdad matemática (que no tendría sentido), sino una **asignación de valor,** que se podría leer como «quiero que el nuevo valor de la variable **numero** sea igual al antiguo valor de la variable **numero**, incrementado en una unidad».

El indicador de fin de línea, **end**, no solo puede ser cadena vacía, también sería aceptable casi cualquier otro carácter, de modo que la orden **print** anterior podría ser:

```
print (numero , end=" ")
```

4.3. Incremento, decremento y otras operaciones aritméticas abreviadas

La operación incrementar una variable **(numero = numero+1)** es tan habitual en programación que, algunos lenguajes como Python, tienen **una notación especial,** para indicarla de forma abreviada (**numero += 1**), tal como se puede observar en esta variante del ejemplo anterior:

```
# Contador con while, operador de incremento

numero = 1
while numero <= 10:
  print (numero , end=" ")
  numero += 1
```

```
1 2 3 4 5 6 7 8 9 10
```

Si se desea **incrementar** en más de una unidad, basta con cambiar ese valor numérico por otro (**n += 2**). Existe una notación abreviada para **decrementar** el valor (**n -= 5**), para **multiplicarla** (**n *= 3**) o **dividirla** (**n /= 2**).

```
# Ejemplo de operaciones abreviadas

n = 10
print("Valor inicial:", n)
n += 2
print("Si lo incrementamos en 2:", n)
n -= 5
print("Si lo decrementamos en 5:", n)
n *= 3
print("Si lo multiplicamos por 3:", n)
n /= 2
print("Si lo convertimos en su mitad:", n)
```

```
Valor inicial: 10
Si lo incrementamos en 2: 12
Si lo decrementamos en 5: 7
Si lo multiplicamos por 3: 21
Si lo convertimos en su mitad: 10.5
```

4.4. for

Cuando se desea **crear un contador,** se puede emplear la orden **for**, cuyo uso habitual es **recorrer una lista de datos,** pero que, con la ayuda de la función **range**, permite **obtener** los **valores** que hay **dentro de un rango,** uno a uno, como en el siguiente ejemplo:

```
# Contador con "for"

for numero in range(1,11):
  print (numero , end=" ")
```

```
1 2 3 4 5 6 7 8 9 10
```

El **primer** valor de **range()** indica el **valor inicial,** y el **último** señala el **punto final,** que no se alcanza, por lo que la orden actual saldría a ser «para los valores que están en el rango que comienza en 1 y que llega casi hasta 11...».

Además, **range** permite **especificar el incremento** que se desea aplicar en cada iteración de la orden **for**. Por ejemplo, se pueden mostrar los números impares que hay entre el 1 y el 10, comenzando en el 1 e incrementando de 2 en 2, así:

```
# Avanzar de 2 en 2 con "for"

for numero in range(1,11,2):
  print (numero , end=" ")
```

```
1 3 5 7 9
```

De la misma forma, se podría aplicar un **incremento negativo.** Conviene recordar que **el primer valor se incluye, pero el último no.** Así, si se desea hacer una cuenta atrás desde el 10 hasta el 0, el valor límite de **range** deberá ser –1:

```
# Contar hacia atrás con "for"

inicio = 10
limite = -1
incremento = -1
for numero in range(inicio, limite, incremento):
  print (numero , end=" ")
```

```
10 9 8 7 6 5 4 3 2 1 0
```

4.5. Bucles sin fin

En ocasiones, ya sea por error o intencionadamente, se puede provocar que un **bucle no tenga salida,** bien porque se plantee mal la condición de salida, bien porque sea incorrecto el incremento de la variable que lo controla.

Un primer ejemplo puede ser esta versión del contador con **while**, que tiene una condición incorrecta, porque el valor inicial de la variable **numero** no es 0 y, en cada iteración (vuelta del bucle), se va alejando aún más de ese valor 0:

```
# Contador con while, condición incorrecta

numero = 1
while numero != 0:
  print (numero , end=" ")
  numero += 1
```

Un segundo ejemplo puede ser esta variante, que no incrementa la variable **numero**, por lo que nunca se acercará al valor límite 10:

```
# Contador con while, sin incremento (incorrecto)

numero = 1
while numero <= 10:
  print (numero , end=" ")
```

4.6. Interrumpir un bucle

Es posible **salir** de un bucle **antes** de tiempo, empleando la orden **break**:

```
# Interrumpir un bucle "for" con "break"

for numero in range(1,11):
  if numero == 5:
    break
  print (numero , end=" ")
```

```
1 2 3 4
```

Se puede observar que, cuando se alcanza el valor 5, el bucle se interrumpe por completo, y no se procesan más valores.

La orden **continue** tiene un comportamiento parecido, pero **no abandona** por completo el bucle, sino que solo se **salta la iteración** (vuelta o pasada) actual.

```
# Saltar una iteración de un bucle "for" con "continue"

for numero in range(1,11):
  if numero == 5:
    continue
  print (numero , end=" ")
```

```
1 2 3 4 6 7 8 9 10
```

En este caso, al alcanzar el valor 5, se da por terminada esa vuelta del bucle (y, por tanto, no se procesa la orden **print**), pero sí se continúa posteriormente con el siguiente valor, el 6.

Tanto la orden **break** como **continue** están mal vistas, porque **interrumpen el flujo esperable del programa y lo hacen menos legible.**

4.7. Bucles anidados

Los bucles se pueden anidar, es decir, **incluir uno dentro de otro.** De este modo, por ejemplo, es posible escribir las tablas de multiplicar del 1 al 5:

```
# Tablas de multiplicar

for tabla in range(1,6):
    for numero in range(1,11):
        print(tabla, "por", numero, "es", tabla*numero)
    print()  # Línea en blanco entre tablas
```

```
1 por 1 es 1
1 por 2 es 2
1 por 3 es 3
(...)
5 por 8 es 40
5 por 9 es 45
5 por 10 es 50
```

4.8. Nombres de las variables

Cuando una variable se utiliza solo para contar, es habitual emplear como nombre la letra **i** (abreviatura de *index*, 'índice'), así:

```
for i in range(1,11):
    print(i, end=" ")
```

```
1 2 3 4 5 6 7 8 9 10
```

Si se trata de dos o más bucles anidados, se pueden emplear **j**, **k** y letras sucesivas para cada uno de los siguientes niveles, pero solo en casos en los que la **lógica sea muy simple.** Por ejemplo, el siguiente programa muestra los pares de números (1,1), (1,2) y siguientes hasta llegar a (3,3):

```
for i in range(1,4):
    for j in range(1,4):
        print("(", i, ",", j, ") ", end=" ")
```

```
( 1 , 1 ) ( 1 , 2 ) ( 1 , 3 ) ( 2 , 1 ) ...
( 3 , 3 )
```

Por el contrario, si las variables no son solo contadores, sino que tienen un significado real en el problema, se deberían buscar términos más representativos, como **tabla** y **numero**, empleados en el ejemplo de las tablas de multiplicar; o **fila** y **columna**, utilizados en el siguiente código, que muestra un rectángulo de asteriscos:

```
for fila in range(1,4):
    for columna in range(1,9):
        print("*", end=" ")
    print()
```

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

Actividades

21 Crea un programa que pida a la persona usuaria su contraseña, tantas veces como sea necesario, hasta que introduzca el número 7890.

22 Usando **while**, escribe en pantalla, de mayor a menor, los números pares del 26 al 10.

23 Prepara un programa que pida al usuario o la usuaria una serie de números positivos y, además, calcule la suma de todos ellos (este deberá finalizar cuando se introduzca un número negativo o 0).

24 Elabora un programa que pida, al usuario o la usuaria, su código de usuario y su contraseña, y que no le permita finalizar hasta que introduzca el código de usuario 1024 y la contraseña 7890.

25 **Comprobamos.** Compón el código de un programa que otorgue a, la persona usuaria, la oportunidad de adivinar un número del 1 al 100, en un máximo de seis intentos. Ten en cuenta que, en cada intento, el programa deberá avisar de si se ha pasado o se ha quedado corto o corta.

5
ESTRUCTURAS BÁSICAS DE DATOS

5.1. Nociones básicas sobre arrays

Un **array** o tabla es un **conjunto de elementos,** normalmente todos ellos **del mismo tipo** (por ejemplo, todos números o todos textos). Estos elementos compartirán todos el **mismo nombre** y ocuparán un **espacio contiguo** en la memoria.

Si se conocen los valores iniciales de un array, se pueden detallar entre **corchetes** y **separados por comas,** de la siguiente forma:

```
datos = [ 10, 20, 30, 40, 50]
```

Esa variable **datos** sería un array, que contendría cinco elementos prefijados. La forma habitual de acceder a ellos es a partir de su posición, indicada de forma numérica, contando a partir de 0. Así, podríamos mostrar el primer y el último dato de ese array con:

```
# Primer y último dato de un array

datos = [ 10, 20, 30, 40, 50]
print(datos[0])
print(datos[4])
```

```
10
50
```

Es habitual recorrer esos cinco valores con la ayuda de un **for**. En la mayoría de los lenguajes de programación, esto se conseguiría recorriendo los índices (desde la posición 0, hasta la *n*-1), así:

```
# Contenido de un array, por posición

datos = [ 10, 20, 30, 40, 50]
for posicion in range(0,5):
  print(datos[posicion])
```

```
10
20
30
40
50
```

Pero Python, al igual que otros lenguajes modernos, **permite** también **recorrer toda la estructura de datos,** consultando sus datos **uno a uno,** sin necesidad de pensar en qué posición se encuentran, de la siguiente manera:

```
# Contenido de un array, de la forma "natural"

datos = [ 10, 20, 30, 40, 50]
for numero in datos:
  print(numero)
```

```
10
20
30
40
50
```

En Python, comparado con otros lenguajes, no es especialmente fácil crear un **array «vacío»**: no se puede «reservar espacio para tres datos», con la intención de rellenarlos posteriormente, ni se puede acceder a posiciones que aún no estén utilizadas. Por ejemplo, el siguiente programa da error:

```
# Error: no se puede acceder a posiciones inexistentes

datos = [ ]
datos[0] = 9
```

La alternativa es rellenarlo con una cierta cantidad de ceros:

```
# Rellenar con "n" ceros

datos = [0] * 3
datos[0] = 9
print(datos)
```

```
[9, 0, 0]
```

5.2. Listas frente a arrays

En la mayoría de los lenguajes, un **array** es una **estructura que tiene un tamaño prefijado** y que **no se puede cambiar.** En caso de necesitar una estructura de datos que pueda aumentar su tamaño cuando se quiera añadir nuevos elementos, no es aconsejable usar un array, ya que estos no pueden crecer.

Por eso, la mayoría permiten también crear **listas,** capaces de crecer, pero que, **en ocasiones, no dejan acceder directamente a una cierta posición,** a no ser que se recorran antes todas las que le preceden, de forma secuencial.

En Python, realmente, los arrays están implementados como **listas,** en el sentido de que se les puede añadir nuevos datos, pero conservan la ventaja de que facilitan acceder a cualquiera, indicando su posición.

La forma más habitual de **añadir datos** al final de una lista o array es mediante el uso de **.append()**:

```
# Añadir a una lista
datos = [ 10, 20, 30, 40, 50]
datos.append(60)
datos.append(70)
for numero in datos:
  print(numero)
```

```
10
20
30
40
50
60
70
```

Cuando **no haya datos iniciales** (por ejemplo, si todos ellos los va a introducir la persona usuaria, se van a leer de fichero, o descargar desde Internet), se indicará dejando los **corchetes vacíos:**

```
# Guardar datos en una lista
lista = [ ]
n = int ( input("Dime un dato (0 para terminar): ") )
while n != 0:
  lista.append(n)
  n = int ( input("Dime otro dato (0 para terminar): ") )
for numero in lista:
  print(numero)
```

```
Dime un dato (0 para terminar): 2
Dime otro dato (0 para terminar): 3
Dime otro dato (0 para terminar): 4
Dime otro dato (0 para terminar): 0
2
3
4
```

Si se desea **saber cuántos datos hay** (por ejemplo, para recorrerlos del último al primero, por posición), se empleará **len()**:

```
# Mostrar datos de una lista al revés
lista = [ ]
n = int ( input("Dime un dato (0 para terminar): ") )
while n != 0:
  lista.append(n)
  n = int ( input("Dime otro dato (0 para terminar): ") )
ultimaPosicion = len(lista)-1
limite = -1
incremento = -1
for posicion in range(ultimaPosicion, limite, incremento):
  print( lista[posicion] )
```

```
Dime un dato (0 para terminar): 2
Dime otro dato (0 para terminar): 3
Dime otro dato (0 para terminar): 4
Dime otro dato (0 para terminar): 0
4
3
2
```

5.3. Arrays y física: vectores

Muchas magnitudes físicas se formulan mediante **vectores,** que se suelen expresar con un **array:** al analizar una fuerza, no solo es importante cuán intensa es, sino, también, la dirección y el sentido en los que esta se aplica. Por ejemplo, una fuerza en el espacio de tres dimensiones se podría expresar como un array de tres números reales:

```
# Vector fuerza

import math
fuerza = [ ]

x = float ( input("Dime la componente X de la fuerza: ") )
fuerza.append(x)
y = float ( input("Dime la componente Y de la fuerza: ") )
fuerza.append(y)
z = float ( input("Dime la componente Z de la fuerza: ") )
fuerza.append(z)
print("El vector fuerza es:", fuerza)
print("O bien: (", fuerza[0], ",", fuerza[1], ",", fuerza[2], ")")
print("Y su modulo es ",
   math.sqrt(fuerza[0]**2 + fuerza[1]**2 + fuerza[2]**2))
```

```
Dime la componente X de la fuerza: 2
Dime la componente Y de la fuerza: 3
Dime la componente Z de la fuerza: -1.5
El vector fuerza es: [2.0, 3.0, -1.5]
O bien: ( 2.0 , 3.0 , -1.5 )
Y su modulo es 3.905124837953327
```

Existe otra forma más elegante de plantear este problema, usando **clases,** tal como se verá más adelante.

5.4. Arrays bidimensionales

Es posible declarar un **array de dos o más dimensiones** (listas que contienen listas). Por ejemplo, si se desea medir tiempos en pruebas deportivas y guardar datos de dos grupos de tres deportistas cada uno, hay dos alternativas básicas:

- Crear un bloque de seis datos y recordar que los tres primeros corresponden realmente a un grupo de deportistas, y los tres siguientes, a otro.
- Crear un bloque con dos subbloques, cada uno de los cuales tendrá tres elementos, de modo que los datos de la forma **tiempos[0][i]** sean los del primer grupo, y los de la forma **tiempos[1][i]**, del segundo. Esta alternativa es la más natural.

Si se conocen los valores iniciales, se pueden indicar de la siguiente manera:

```
# Ejemplo de array bidimensional (lista de listas)

tiempos = [ [11.5, 12.3, 10.8], [11.3, 12.2, 10.5]]
print("Primer bloque de tiempos: ", tiempos[0])
print("Primer tiempo del segundo bloque: ", tiempos[1][0])
```

```
Primer bloque de tiempos: [11.5, 12.3, 10.8]
Primer tiempo del segundo bloque: 11.3
```

En Python (sin usar bibliotecas externas), es relativamente complejo crear un array bidimensional «vacío» —especificando su tamaño, pero sin indicar los valores de sus datos—. Se puede rellenar con ceros, pero resulta aún menos legible que en el caso de un array unidimensional:

```
# Inicializar con ceros un array bidimensional (lista de listas)

columnas = 3 ; filas = 4
datos = [[0] * columnas for i in range(filas)]
datos[1][2] = 25.3
print(datos)
```

```
[[0, 0, 0], [0, 0, 25.3], [0, 0, 0], [0, 0, 0]]
```

5.5. Arrays y matemáticas: matrices

Los **arrays bidimensionales** también se emplean para **guardar matrices,** cuando es necesario resolver problemas matemáticos más complejos. Por ejemplo, un programa que pida los datos de dos matrices de 3x3 y que muestre su suma —nuevamente, sin emplear ninguna biblioteca externa— podría componerse así:

```
# Suma de dos matrices 3x3
columnas = 3
filas = 3
matrices = 2

# Preparamos las matrices
datos = []
datos.append( [[0] * columnas for i in range(filas)] )
datos.append( [[0] * columnas for i in range(filas)] )
matrizSuma = [[0] * columnas for i in range(filas)]

# Mostramos los datos iniciales (vacíos)
for m in range(0, matrices):
  print("Matriz ", m+1, datos[m] )

# Pedimos los datos reales
for m in range(0, matrices):
  for fila in range(0, filas):
    for columna in range(0, columnas):
      print ("En la matriz ", m+1,
        ", dime el dato de la fila ", fila+1,
        " y la columna ", columna+1, ": ")
      datos[m][fila][columna] = float(input())

# Calculamos la suma
for fila in range(0, filas):
  for columna in range(0, columnas):
    matrizSuma[fila][columna] = datos[0][fila][columna] +\
      datos[1][fila][columna]

# Y mostramos el resultado
for m in range(0, matrices):
  print("La matriz", m+1, "era:", datos[m])
print("La suma es:", matrizSuma)
```

```
Matriz 1 [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
Matriz 2 [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
En la matriz 1 , dime el dato de la fila 1 y la columna 1 :
1
(...)
La matriz 1 era: [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
La matriz 2 era: [[1.1, 1.2, 1.3], [2.1, 2.2, 2.3], [3.1, 3.2, -3.3]]
La suma es: [[2.1, 3.2, 4.3], [6.1, 7.2, 8.3], [10.1, 11.2, 5.7]]
```

Las **matrices** son una **estructura de datos** tan habitual en la manipulación de datos numéricos que existen **bibliotecas específicas** para realizar operaciones con ellas, de forma más sencilla que empleando solo Python estándar. La más extendida es, posiblemente, **numpy**, cuya instalación se verá en la siguiente unidad. Con ella, sumar dos vectores se llevaría a cabo simplemente así:

```
import numpy
vector1 = numpy.array([1, 2, 3])
vector2 = numpy.array([0, 6, -5])
suma = numpy.add(vector1, vector2)
print(suma)
```

```
[ 1  8 -2]
```

Así, un programa equivalente al anterior, que sume dos matrices de tamaño 3x3, en esta ocasión, empleando las estructuras básicas de **numpy** y guardando cada matriz en una variable independiente, podría ser el siguiente:

```
import numpy
columnas = 3
filas = 3
matriz1 = numpy.zeros((filas, columnas))
matriz2 = numpy.zeros((filas, columnas))
for fila in range(0, filas):
  for columna in range(0, columnas):
    print ("En la matriz 1, dime dato de fila", fila+1,
      "y columna", columna+1, ": ")
    matriz1[fila][columna] = float(input())
for fila in range(0, filas):
  for columna in range(0, columnas):
    print ("En la matriz 2, dime dato de fila", fila+1,
      "y columna", columna+1, ": ")
    matriz2[fila][columna] = float(input())
matrizSuma = numpy.add(matriz1, matriz2)
print("La matriz 1 es:", matriz1 )
print("La matriz 2 es:", matriz2 )
print("La suma es:", matrizSuma)
```

```
La matriz 1 es: [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
La matriz 2 es: [[13. 12. 11.]
 [ 1.  2.  3.]
 [ 4.  5.  6.]]
La suma es: [[14. 14. 14.]
 [ 5.  7.  9.]
 [11. 13. 15.]]
```

Actividades

26 Crea un programa que pida cuatro números al usuario o la usuaria, los memorice (utilizando una lista), calcule su media aritmética y, finalmente, muestre en pantalla tanto la media como los datos tecleados.

27 Prepara un programa que almacene, en un array, el número de días que tiene cada mes (suponiendo que se trata de un año no bisiesto), pida a la persona usuaria que indique un mes determinado (1 = enero y 12 = diciembre) y muestre, en pantalla, el número de días que tiene el mes indicado. Deberá mostrar un mensaje como «El mes 3 tiene 31 días».

28 Diseña un programa que guarde, en un array, el número de días que tiene cada mes (de un año no bisiesto), pida a la persona usuaria que indique un mes (por ejemplo, febrero = 2) y un día determinado (por ejemplo, el día 15), y señale qué número de día es dentro de ese año (por ejemplo, el día 15 de febrero sería el día número 46, y el 31 de diciembre, el 365).

29 Realiza un programa que pida diez números enteros e indique cuál es el mayor.

30 Elabora un programa que pida, a la persona usuaria, los datos de dos vectores en un plano (dos coordenadas) y calcule su diferencia.

31 Diseña un programa que pida los componentes de dos vectores en el espacio (tres coordenadas) y calcule su producto escalar.

32 Crea un programa que pida, a la persona usuaria, dos vectores en el plano (dos coordenadas) y diga si son linealmente dependientes (es decir, si sus componentes son proporcionales).

33 Implementa un programa que pida los datos de una matriz de 2x2 y muestre su traspuesta (el resultado de intercambiar filas por columnas).

34 Compón un programa que pida los datos de una matriz de 3x3 y muestre su determinante.

35 Busca información sobre la «ordenación de burbuja» y crea un programa que pida diez datos numéricos y los muestre ordenados de menor a mayor.

6 FICHEROS

En numerosas ocasiones, será deseable no perder la información que maneja un determinado programa. Para ello, se pueden **volcar los datos en un fichero** antes de salir del programa, para recuperarlos en la siguiente sesión.

Las tres operaciones básicas para manejar ficheros son: **1) abrir; 2) leer o escribir datos;** y **3) cerrar el fichero.**

Además, será necesario **comprobar** posibles **errores;** por ejemplo, puede ocurrir que se esté intentando abrir un fichero que realmente no existe o tratando de escribir en un dispositivo de solo lectura.

6.1. Escritura en un fichero de texto

El manejo de ficheros de texto es muy parecido al de la consola de texto. Para guardar un dato, se usa una sintaxis similar a la de **print**, con la excepción de los avances de línea; en la consola, son automáticos, mientras que en el fichero de texto, no.

El primer paso será **abrir el fichero,** mediante la orden **open()**, a la que hay que **indicar** dos detalles **entre paréntesis:** el **nombre del fichero,** dentro de la unidad de disco, y el **modo de apertura.**

Los modos de apertura más habituales son **w**, abreviatura de *write,* para escribir datos, y **r**, de *read,* para leer desde un fichero existente:

```
fichero = open("ejemplo.txt", "w")
```

El segundo paso es **escribir cada dato,** usando la orden **write**:

```
fichero.write("Línea 1");
```

Pero es importante insistir en que el siguiente texto que se enviase al fichero quedaría en la misma línea, a continuación de ese texto. Para **forzar un salto de línea,** se deberá añadir un carácter especial, que se indica con el par de símbolos **\n** *(new line),* así:

```
fichero.write("Línea 1\n");
```

Y el último paso será **cerrar el fichero,** con la orden **close()**:

```
fichero.close()
```

Un programa completo, que crease un fichero de texto y escribiese dos líneas de texto en él, podría estar compuesto como el siguiente:

```
fichero = open("ejemplo.txt", "w")
fichero.write("Línea 1\n");
fichero.write("Línea 2\n");
fichero.close()
```

Cabe mencionar que los saltos de línea no se representan de igual forma en todos los sistemas operativos: el carácter especial **\n** es el formato de avance de línea que se emplea en Linux, y que será reconocido de forma correcta por Python en cualquier sistema operativo.

Sin embargo, si un fichero de texto como el que genera este programa se abre desde una utilidad específica de un cierto sistema operativo, como podría ser el Bloc de Notas de Windows, es probable que no se vea correctamente, sino que parezca estar todo en una misma línea.

Probar programas

Un programa como este no se podrá probar desde un entorno *online,* que por lo general bloqueará el acceso al sistema de ficheros. Será necesario tener Python instalado en el sistema local para poder probarlo.

6.2. Lectura de un fichero de texto

Para **leer** el contenido de un **fichero,** se deberá hacer con el modo **r**:

```
f = open("ejemplo.txt", "r")
```

A su vez, es posible **leer una línea,** con la orden **readline()**:

```
linea = f.readline()
print(linea)
```

Y, al terminar de usar el fichero, será recomendable (aunque no crítico, como en el caso de escribir datos) **cerrarlo:**

```
f.close()
```

Así, el siguiente programa mostraría la **primera línea** de un fichero:

```
f = open("ejemplo.txt", "r")
linea = f.readline()
f.close()
print(linea)
```

El **problema** de este planteamiento es que lo habitual es no saber, *a priori,* cuántas líneas forman el fichero, por lo que no se podrá usar un contador para leerlas. Por eso, la alternativa más habitual es recorrer el contenido del fichero con una orden **for**, así:

```
f = open("ejemplo.txt", "r")
for linea in f:
    print(linea)
f.close()
```

Esto tiene un **inconveniente adicional:** el carácter de salto de línea (**\n**), que marca el final de cada línea del fichero, queda formando parte de la cadena de texto, por lo que el programa anterior mostrará espacios en blanco entre ellas. Una solución sencilla, pero no ideal, es **mostrar las líneas sin ese salto** de línea adicional que añadiría la orden **print**.

```
f = open("ejemplo.txt", "r")
for linea in f:
    print(linea, end="")
f.close()
```

Otra solución más elegante pasa por **eliminar el salto de línea,** lo que se puede conseguir con **.rstrip()**, que elimina los **espacios** en blanco al final del texto *(right strip)* y el **salto de línea,** si lo hubiera:

```
f = open("ejemplo.txt", "r")
for linea in f:
    linea = linea.rstrip()
    print(linea)
f.close()
```

6.3. Lectura y escritura en bloque

También existe la opción de **leer todo el fichero,** con una única orden, **readlines()**, lo que creará una lista en la que cada elemento será una línea del fichero:

```
f = open("ejemplo.txt", "r")
lineas = f.readlines()
f.close()

print(lineas)
```

Asimismo, existe una orden **writelines**, que vuelca todo el contenido de una lista a un fichero, pero tiene el inconveniente de que no añade el salto de línea al final de cada elemento. Por eso, una alternativa más habitual para guardar el contenido de una lista de datos sería:

```
f = open("ejemplo.txt", "w")
for linea in lineas:
    f.write(f"{linea}\n")
f.close()
```

6.4. Errores en el acceso a ficheros: excepciones

La forma habitual de tratar los posibles **errores en tiempo** de ejecución en un lenguaje moderno es mediante el uso de **excepciones.** La idea detrás de estas es que no se incluye una orden **if**, para comprobar si cada paso de un proceso ha resultado correctamente, sino que se intenta (orden **try**) dar una serie de pasos y, a continuación, se indica qué debería hacerse, en caso de que haya existido algún problema. Este planteamiento permite separar la lógica del problema de la de gestión de errores. Así, una versión mejorada del programa anterior podría ser:

```
try:
    f = open("ejemplo.txt", "r")
    lineas = f.readlines()
    f.close()
    print(lineas)
except:
    print("No se ha podido leer el fichero")
```

Se pueden **interceptar** varias **excepciones distintas,** que se deberán **ordenar** de la más específica a la más general, e **indicar el nombre** de cada una de ellas (excepto, quizá, el de la última, para que el último caso atrape cualquiera de ellas):

```
try:
    f = open("ejemplo.txt", "r")
    lineas = f.readlines()
    f.close()
    print(lineas)
except FileNotFoundError:
    print("Fichero no encontrado")
except:
    print("Error de lectura")
```

También existen dos bloques opcionales: un bloque **else**, que se lanzaría en caso de **no detectarse errores** y que puede ser una alternativa más legible al programa anterior, que se emplea para **separar** el fragmento de programa que puede tener errores de aquel en el que no se espera que los haya:

```
try:
   f = open("ejemplo.txt", "r")
   líneas = f.readlines()
   f.close()
except FileNotFoundError:
   print("Fichero no encontrado")
except:
   print("Error de lectura")
else:
   print("El contenido del fichero es:")
   print(líneas)
```

Y un bloque **finally**, que se lanzaría **al final del proceso,** tanto si ha habido algún error como si todo ha funcionado correctamente:

```
print("Analizando el fichero...")
try:
   f = open("ejemplo.txt", "r")
   lineas = f.readlines()
   f.close()
except FileNotFoundError:
   print("Fichero no encontrado")
except:
   print("Error de lectura")
else:
   print("El contenido del fichero es:")
   print(lineas)
finally:
   print("Análisis del fichero terminado")
```

Actividades

36 Crea un programa que pida números y los guarde en un fichero **«numeros.txt»,** separados por espacios en blanco. Finalizará cuando se introduzca el número 0.

37 Desarrolla un programa que pida un nombre de fichero y muestre su contenido, haciendo una pausa cada 24 líneas. Pista: Deberás leer línea a línea y comprobar si **numeroLinea % 24 == 23**.

38 Diseña un programa que muestre el resultado de la suma de los números almacenados en **«numeros.txt».**

39 Elabora un programa que permita, al usuario o la usuaria, anotar el nombre, correo electrónico y teléfono móvil de sus contactos. Al comenzar, mostrará un menú que facilite escoger entre buscar y mostrar un contacto a partir del nombre, añadir un nuevo contacto o salir del programa. Cuando este termine, los datos se deberán guardar en un fichero llamado **«contactos.txt».** Cada vez que el programa se ponga en marcha, cargará los datos desde ese fichero, si existe (pero deberá analizar los posibles errores, por si no existe).

7 FUNCIONES

7.1. Los problemas de un código repetitivo

Es habitual que algunas tareas deban **repetirse varias veces,** en distintos puntos de un programa. En tales casos, volver a teclear varias veces el mismo fragmento de código no suele ser la solución óptima, ya que conlleva los siguientes **inconvenientes:**

- la escritura del programa llevará más tiempo;
- el código fuente final resultará menos legible;
- la posibilidad de cometer algún error será más elevada, tanto cada vez que se vuelva a teclear el fragmento repetitivo como cuando se decida hacer una modificación en uno de estos, por la probabilidad de olvidar incluirla en el resto.

Por ello, conviene **evitar que un programa contenga código repetitivo.** Una manera de lograrlo es descomponerlo en **bloques,** los cuales reciben el nombre de ***funciones.***

Un ejemplo de programa repetitivo podría ser el siguiente, que escribe varios textos subrayados:

```
# Primer acercamiento a las funciones: fuente repetitivo

print ("Primer ejemplo")
for i in range(0, 20):
   print ("-", end="")
print() # Avance de línea

print ("Segundo ejemplo")
for i in range(0, 20):
   print ("-", end="")
print() # Avance de línea

print ("Tercer ejemplo")
for i in range(0, 20):
   print ("-", end="")
print() # Avance de línea
```

```
Primer ejemplo
--------------------
Segundo ejemplo
--------------------
Tercer ejemplo
--------------------
```

Si bien el código anterior funciona, es muy mejorable. De hecho, es posible hacerlo algo más elegante con la creación de un bloque **subrayar()**, que dé esos pasos repetitivos, de tal forma que el cuerpo del programa principal quede de la manera que se muestra a continuación y respecto de lo que se estudiará en detalles del siguiente apartado:

```
print ("Primer ejemplo")
subrayar()
print ("Segundo ejemplo")
subrayar()
print ("Tercer ejemplo")
subrayar()
```

Esta nueva versión del programa resulta compacta y más legible, pero además será más fácil de mantener: si se decide alguna mejora para la función **subrayar()**, todos los puntos del programa que la emplean lo harán de forma automática, sin tener que hacer los cambios en varios distintos.

Escribir un texto repetitivo

Esa forma de subrayar es la que se emplearía en otros lenguajes de programación. En realidad, Python permite una forma abreviada de escribir un texto repetitivo, usando el operador de multiplicación, pero esa estructura no existe en la mayoría de lenguajes:

```
print ("-" * 20)
```

7.2. Una primera función

Crear un bloque con nombre es sencillo. Para ello, habrá que **elegir un nombre,** precederlo de **def** (abreviatura de *definition*) y **detallar,** tabulando a la derecha, los **pasos** que debe dar:

```
# Primer ejemplo de uso de funciones

# Función auxiliar para subrayar
def subrayar():
  for i in range(0, 20):
    print ("-", end="")
  print() # Avance de línea

# Cuerpo del programa
print ("Primer ejemplo")
subrayar()
print ("Segundo ejemplo")
subrayar()
print ("Tercer ejemplo")
subrayar()
```

Este programa tiene dos bloques: el primero, **subrayar()**, se encarga de **ejecutar la tarea** repetitiva. El segundo, es el cuerpo del programa, que se **ayuda** del anterior para que el resultado sea **más legible,** más **compacto y fácil** de mantener.

Estos bloques de programa con nombre se suelen llamar *funciones.* En realidad, según el lenguaje de programación que se utilice, se suele distinguir entre ***procedimientos*** o ***subrutinas,*** que dan una serie de pasos, pero no devuelven ningún resultado, como en el ejemplo anterior, y ***funciones «propiamente dichas»,*** que realizan varios pasos y, finalmente, devuelven un resultado, tal como ocurre con la función **math.sqrt** (raíz cuadrada), que ya se ha empleado y que es parte del sistema básico de Python.

La variable **i**, que se ha utilizado dentro de la función **subrayar()**, es lo que se conoce como una ***variable local;*** la cual solo es **accesible** dentro del bloque **donde está declarada,** de modo que su valor no se podrá consultar ni cambiar desde el cuerpo del programa ni desde ninguna otra función, por lo que el siguiente programa es incorrecto:

```
# Intento (incorrecto) de usar una variable local

def subrayar():
  for i in range(0, 20):
    print ("-", end="")
  print()

print ("Primer ejemplo")
subrayar()
print(i)
```

Lo contrario a una variable local es una ***variable global;*** una que estuviera **declarada fuera** de todas las funciones, **accesible por todas.** Aun así, **en Python,** al contrario que en la mayoría de lenguajes, se da por sentado que **todas** las variables que se utilicen dentro de una función **son locales.**

La importancia del orden

Una función deberá estar declarada antes de usarse. En general, esto supone que el cuerpo del programa tendrá que estar tras todas las funciones que este vaya a utilizar.

En caso de que alguna sea global (lo que no es deseable, en general), se deberá indicar con la palabra clave **global**, tal como se observa en el siguiente ejemplo:

```
# Ejemplo de variable global

n = 5

def duplicarN():
  global n
  n *= 2

print (n)
duplicarN();
print (n)
```

```
5
10
```

7.3. Parámetros de una función

Habitualmente, resultará práctico **indicar,** a la función, ciertos **datos** con los que se desea que esta trabaje. Por ejemplo, es posible mejorar la función **subrayar()** para que no escriba siempre 20 guiones, sino la cantidad exacta que se indique, tal como se muestra a continuación:

```
# Una función con un parámetro

# Función auxiliar para subrayar
def subrayar(n):
  for i in range(0, n):
    print ("-", end="")
  print() # Avance de línea

# Cuerpo del programa
print ("Primer ejemplo")
subrayar(16)
print ("Segundo ejemplo")
subrayar(17)
print ("Tercer ejemplo")
subrayar(16)
```

```
Primer ejemplo
----------------
Segundo ejemplo
-----------------
Tercer ejemplo
----------------
```

Estos **datos adicionales** se conocen como ***parámetros.*** Para cada uno, se deberá escoger un **nombre.** En caso de necesitar varios parámetros, sus nombres **se separarán mediante comas:**

```
# Una función con dos parámetros

def mostrarSuma(n1, n2):
  print (n1+n2)

n = 5
mostrarSuma(12, n)

a = 20
b = - 5
mostrarSuma(a, b)
```

Actividades

40 Desarrolla una función **saludarVariasVeces** que escriba el texto «Hola» en pantalla, separado por espacios, tantas veces como lo indique el parámetro.

41 Implementa una función **escribirTabla** que muestre en pantalla la tabla de multiplicar del número que indique el parámetro.

7.4. Valor devuelto por una función

Con frecuencia, es deseable que una función **realice** una serie de **cálculos** y **devuelva su resultado,** con el fin de poder usarlo desde cualquier parte del programa. Por ejemplo, se puede crear una función para elevar un número al cuadrado:

```
# Una función que devuelve un valor
def cuadrado(n):
  return n ** 2

print( cuadrado(5) )
```

```
25
```

Tal como se puede observar, una función que devuelva un valor debe terminar con una orden **return**, seguida del resultado.

Si una función puede **devolver un valor** de entre varios distintos, **podrá contener varias** órdenes **return** distintas. Por ejemplo, se puede crear una función que diga cuál es el mayor de dos números reales, así:

```
# Función con varios return
def mayor(n1, n2):
  if n1>n2:
    return n1
  else:
    return n2

numero1 = float(input("Dime un numero: "))
numero2 = float(input("Dime otro: "))
print( "El mayor es",
  mayor(numero1, numero2) )
```

```
Dime un numero: 5.9
Dime otro: -3.1
El mayor es 5.9
```

Pero, por legibilidad, en funciones que no sean tan sencillas, es preferible usar alguna **variable temporal,** de modo que exista un único **return**, que se encuentre al final de la función, como en esta versión alternativa del programa anterior:

```
# Un "return" en vez de varios
def mayor(n1, n2):
  if n1>n2:
    mayorTemporal = n1
  else:
    mayorTemporal = n2
  return mayorTemporal

numero1 = float(input("Dime un numero: "))
numero2 = float(input("Dime otro: "))
print( "El mayor es",
  mayor(numero1, numero2) )
```

Es importante recordar que las variables locales, como es **mayorTemporal**, solo son accesibles desde dentro de la función, no desde el cuerpo del programa ni desde ninguna otra.

Parámetros y variables

Tal como se muestra en estos dos ejemplos, los parámetros que se usan en una función no tienen por qué llamarse igual que las variables que existan en el cuerpo del programa

Actividades

42 Crea una función **suma** que devuelva la suma de tres números que se indiquen como parámetros.

43 Implementa una función **esPrimo** que reciba un número y devuelva el valor 1, si es primo, y el valor 0 si no lo es.

7.5. Modificar el valor de un parámetro

En caso de que se intente **modificar el valor de un dato** que una función reciba **como parámetro, los cambios no se conservan** cuando se sale de dicha función, tal como se muestra en el siguiente ejemplo:

```
# Intentar modificar un parámetro

def duplicar(n):
    n = n*2

numero = 5
print ("El numero vale", numero)
duplicar(numero);
print ("Tras duplicar, se convierte en", numero)
```

```
El numero vale 5
Tras duplicar, se convierte en 5
```

Esto se debe a que, en la mayoría de lenguajes, y a no ser que se indique lo contrario de forma explícita, los parámetros se «pasan por valor», es decir, se hace una **copia del dato original** y, dentro de la función, se trabaja con esa copia, desvinculándolo de esta.

Cuando se pasan datos más complejos, que se estudiarán más adelante, el comportamiento puede ser ligeramente distinto.

Por lo general, el hecho de que no se modifique el valor de un parámetro no supone una limitación importante ya que, si interesa que una función devuelva un determinado valor, basta con usar **return**.

7.6. Devolver dos valores

Una peculiaridad de Python, que no está presente en muchos otros lenguajes de programación, es que se puede conseguir que una función devuelva dos valores. Ambos se deberán separar por comas en la orden **return** y asignar a variables separadas por comas, cuando se llame a la función. Por ejemplo, el siguiente programa muestra una forma de devolver tanto el primer valor como el último de una lista:

```
def primeroYUltimo(lista):
    return lista[0], lista[-1]
datos = [4, 2, 8, 6]
prim, ult = primeroYUltimo(datos)
print("Primero = ",prim,"Último = ",ult)
```

```
Primero =  4 Último =  6
```

Actividades

44 Crea una función llamada **dibujarRectangulo,** que mostrará un rectángulo de asteriscos con la anchura y la altura que se indiquen como parámetros. Empléala en un programa que pida, a la persona usuaria, el ancho y el alto del rectángulo.

45 Implementa una función **sumaDeDigitos** que devuelva el resultado de sumar las cifras del número que se indique como parámetro. Pista: Puedes obtener la última cifra de un número si hallas su módulo entre 10, y luego eliminar esa cifra si calculas su división entera entre 10.

46 Haz una función que halle el máximo común divisor (MCD) de dos números. Deberás buscarlo probando todos los valores que haya entre 1 y el menor de ambos, y comprobando si cada uno de esos números es, a la vez, divisor de los dos datos indicados.