5

PROGRAMACIÓN ESTRUCTURADA

¿Qué es un programa?

Un programa es una secuencia de instrucciones para un ordenador.

Los primeros lenguajes de «alto nivel» (similares al lenguaje humano) permitían crear programas formados por una lista de órdenes que se procesaban de principio a fin, salvo que, en algún instante, se indicase que se debía saltar a otra posición del programa:

```
10 PRINT "Hola"
20 GOTO 10
```

Esta forma de trabajar provocaba que, a medida que el tamaño de los programas iba aumentando, estos resultaran cada vez más difíciles de leer y corregir, puesto que era necesario analizar paso a paso de qué punto a qué otro punto del programa se saltaba: es lo que se llamó *código espagueti*.

Como alternativa, a mediados de los años sesenta se diseñaron lenguajes que incorporaban estructuras más legibles, que permitían que un fragmento de un programa se repitiera mientras se cumpliera una cierta condición o descomponer un programa en bloques, denominados *subrutinas*. Este hecho supuso el comienzo de la programación estructurada, que se aplicó a lenguajes como ALGOL, PL/I o Pascal.

Con el paso de los años, se han diseñado otras metodologías más apropiadas para crear programas de gran tamaño (formados por miles de órdenes), como la programación orientada a objetos.

COMPROMISO ODS

Un programa informático puede ser una valiosa herramienta paradar a conocer los Objetivos de Desarrollo Sostenible.

1. Crea, al finalizar esta unidad, un programa sencillo que muestre una lista numerada del 1 al 17, con los nombres de los objetivos. Al introducir el número correspondiente, deberá mostrar un resumen breve sobre a qué se refiere ese objetivo concreto, empleando para ello dos listas de datos.
2. **Mesa redonda.** ¿Qué ventaja tendría la programación de un juego para conocer mejor los ODS frente a un texto que los explique?

En anayaeducacion.es encontrarás información y vídeos sobre las principales metas de los 17 Objetivos de Desarrollo Sostenible.

LENGUAJES, COMPILADORES E INTÉRPRETES

1. Lenguajes de bajo nivel y alto nivel

programa consiste en una **secuencia de instrucciones** para un **ordenador.** Al
al que las personas utilizamos distintos idiomas para comunicarnos, existen
ersos lenguajes con los que podemos dar órdenes a un dispositivo.

roblema radica en que el lenguaje que un ordenador entiende, esto es, el
digo máquina», que es el ejemplo más claro de lenguaje de bajo nivel, es muy
ente a los idiomas hablados por las personas. Resulta difícil de aprender
s fácil cometer errores con él. Por eso, es habitual emplear otros más pare-
os a los de los humanos (los denominados ***lenguajes de alto nivel***) y utilizar
amientas que conviertan el programa a código máquina. Normalmente,
os son muy similares al inglés, aunque más estrictos.

```
PRINT "Hola"
```

Fig. 1. Escritura en pantalla usando un lenguaje de alto nivel (BASIC).

concepto de *algoritmo* está muy relacionado con el de *programa,* aunque, a
ces, se confunden al pensar que son lo mismo. En este caso, un **algoritmo** es
secuencia de pasos (finita y sin ambigüedades) necesaria para resolver un
rto problema. Esta expresión no se usa solo en informática, sino también
otras ciencias, como las matemáticas y la lógica. Un **programa** es el **resulta-**
de implementar un algoritmo, empleando un sistema informático.

2. Compiladores e intérpretes

s **herramientas** encargadas de convertir nuestro programa escrito en lengua-
de alto nivel (lo que se conoce como ***programa fuente***) en código máquina,
bteniendo un programa **ejecutable,** son los **compiladores.** Si este detecta al-
un error en el programa fuente, mostrará el pertinente mensaje de aviso y no
creará ningún ejecutable.

```
PRINT "Hola"
Hola

ESCRIBE "Hola"
Syntax error
```

Fig. 2. Comportamiento de un intérprete: análisis línea a línea en tiempo real.

or su parte, un **intérprete** es otro tipo de **traductor.** La diferencia con el compi-
dor es que, en los intérpretes, no se crea ningún programa ejecutable capaz de
ncionar por sí solo. En el momento de ponerlo en funcionamiento, el intérprete
nvierte cada orden del programa fuente en su equivalente en código máquina
la pone en marcha, y luego la siguiente, y así sucesivamente. Si encuentra un
ror, el programa se detiene en ese punto, pero las órdenes anteriores ya se
abrán ejecutado.

sto supone que, normalmente, un **programa interpretado comenzará** a fun-
onar **antes que un programa compilado** (pues no es necesario traducirlo todo
ara empezar, sino solo la primera orden), **pero será más lento** en los procesos
e cálculo intensivo (porque cada orden se puede tener que traducir varias ve-
es: tantas como se ejecute dicha orden).

3. Pseudocódigo

unque los lenguajes de alto nivel se asemejan al lenguaje natural, es habitual
usar ninguno concreto al plantear, inicialmente, los pasos necesarios para
solver un problema, sino **emplear uno ficticio,** no tan estricto, que puede es-
birse, incluso, en castellano. Este recibe el nombre de ***pseudocódigo:***

```
PEDIR numero1
PEDIR numero2
SI numero2 ≠ 0
   ESCRIBIR "Su división es ", numero1/numero2
SI NO
   ESCRIBIR "No se puede dividir entre cero"
```

1.4. Lenguajes más extendidos

Existen multitud de lenguajes de programación. Muchos de ellos son de **propósito general** (sirven para crear programas de cualquier tipo), mientras que otros están **especialmente diseñados para ciertas tareas,** por lo que pueden resultar menos adecuados para realizar otras distintas. Por ejemplo, hay lenguajes específicos para hacer cálculos matemáticos complejos, para representar y manipular imágenes, para programar robots, etc.

Existen *rankings* de valoraciones, como el índice TIOBE, que **ordenan** los lenguajes de mayor a menor **popularidad,** a partir de ciertos criterios, como la cantidad de búsquedas realizadas sobre ellos en Google y otros buscadores, o la cantidad de programas escritos en ese lenguaje que se pueden encontrar en almacenes de programas fuente, como GitHub y SourceForge.

Algunos de los **lenguajes de programación más extendidos** son **C, C++, C#, Java, JavaScript, PHP** y **Python. C** se suele usar para **crear sistemas operativos y programas** que deban acceder al *hardware;* **Java,** para **aplicaciones** en **servidores web** de gran tamaño y para el sistema **Android; C++,** en **aplicaciones de escritorio y videojuegos,** y **Python** se emplea, cada vez más, como **lenguaje** de propósito **general,** pero también en sectores como el del **análisis de datos.** Se trata de un lenguaje potente, con una sintaxis razonablemente sencilla, y para el que hay multitud de **bibliotecas,** que permiten aplicarlo a muy distintos ámbitos. Además, su intérprete está disponible para la gran mayoría de sistemas operativos actuales, de modo que se pueden crear aplicaciones para LliureX, macOS o Windows, entre otros.

1.5. Hola, mundo

El primer programa que se suele crear, al comenzar a trabajar con un lenguaje de programación, consiste en escribir algo en pantalla. Con frecuencia, ese texto que escribe el programa es un **saludo** al mundo que le rodea, un «Hola, mundo». En otros más antiguos y simples, como BASIC, esto se podía conseguir así:

```
PRINT "Hola, mundo"
```

La mayoría de los lenguajes modernos añaden nuevas posibilidades, como el **acceso a Internet** o **la manipulación de textos** de gran tamaño y de imágenes, pero, a cambio, pueden complicar un programa simple. Por ejemplo, el equivalente al programa anterior, escrito en Java, sería:

```
class HolaMundo {
   public static void main( String args[] ) {
       System.out.print( "Hola, mundo" );
   }
}
```

Por el contrario, **Python** supone una vuelta a la **simplicidad.** En este lenguaje, un programa básico tendría esta apariencia:

```
print("Hola, mundo")
```

➡ Hola, mundo

Esta sencillez aparente no quiere decir que Python sea un lenguaje menos potente que Java, sino que se ha diseñado pensando en que su curva de aprendizaje sea menos pronunciada.

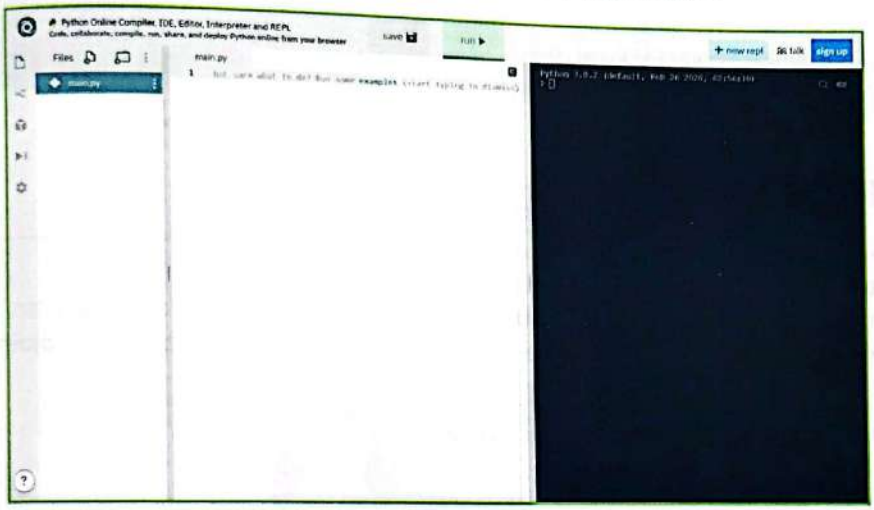
En **anayaeducacion.es** hay una práctica guiada q
las peculiaridades de la sintaxis de Python.

1.6. Probar un programa Python en un navegador web

Resultan, cada vez más frecuentes, los **entornos de programación *online*,** que permiten teclear y probar un programa **sin necesidad de instalar** nada en nuestro equipo. Además, son **independientes** de nuestro sistema operativo, por lo que se podrían emplear, incluso, desde una tableta o un teléfono inteligente.

Es habitual que estos estén divididos en **dos paneles,** uno al lado del otro. Normalmente, el panel **izquierdo** se deberá emplear para **teclear** el programa, mientras que el panel **derecho** mostrará los **resultados.**

Para **lanzar** el programa, existirá un botón como **Run** o **Execute**, o uno que recuerde al botón **Play** de los equipos de música y multimedia.



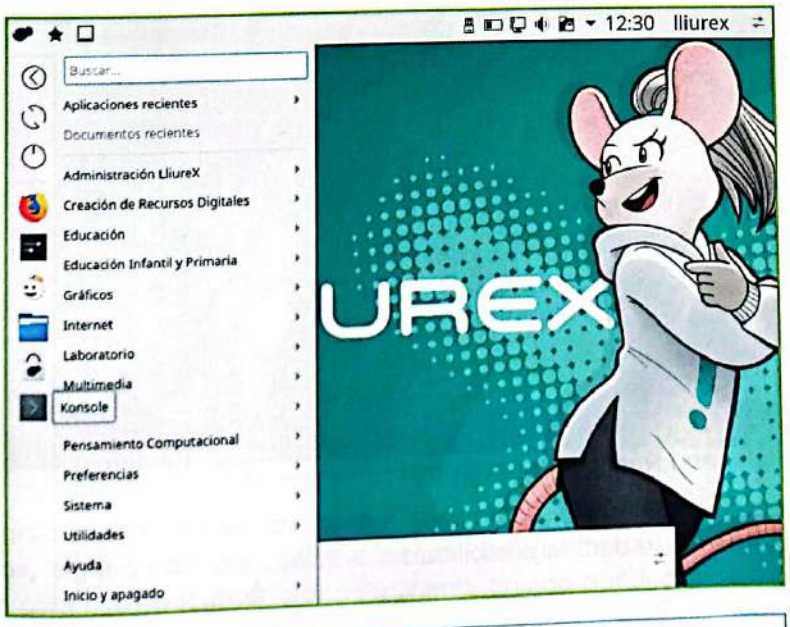
Entornos de programación *online*

Existen muchos entornos de programación *online*. Basta con teclear «Python 3 online compiler» en un buscador, como Google, para encontrar diferentes sitios web, por ejemplo, repl.it, onlinegdb, tutorialspoint o paiza.io.

1.7. Probar un programa en LliureX

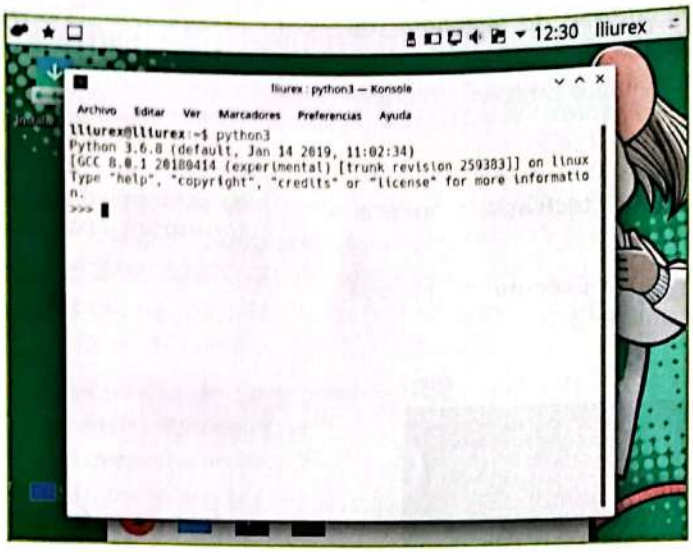
En LliureX, al igual que en la mayoría de distribuciones de Linux, es habitual que Python esté preinstalado y sea accesible a través del menú del sistema, en un apartado llamado **Desarrollo** o **Programación**.

En caso de no existir ese menú, Python se podría lanzar desde un terminal (como Konsole):



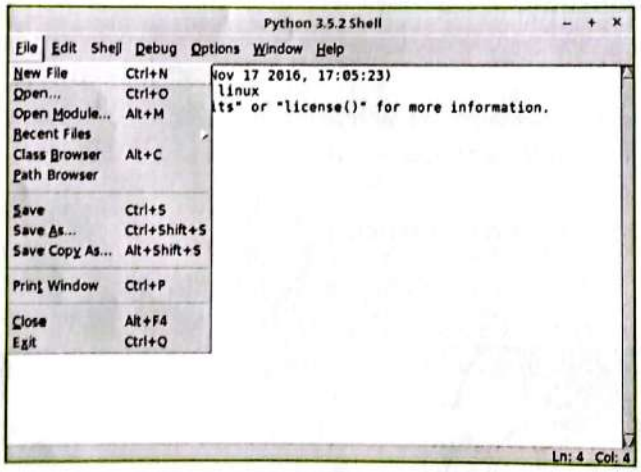
En **anayaeducacion.es** encontrarás un videotutorial sobre entornos *online* para programar con Python.

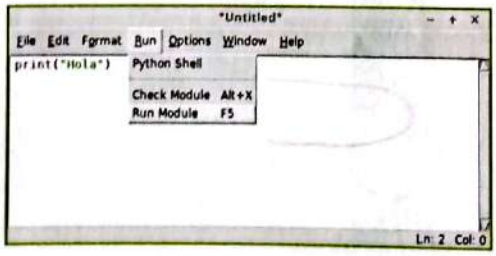
Así, desde la consola, bastaría con teclear «python3», para entrar en su entorno básico de programación.



Si existe un apartado para Python en el menú del sistema, es habitual que aparezca también la opción **Idle**, que es el **editor de texto** que viene preinstalado como parte de la distribución estándar de Python, el cual será más cómodo que el entorno básico y suficiente para programas cuya complejidad no sea muy grande.

Una vez dentro de **Idle**, para crear programas formados por varias órdenes, se deberá acceder al menú **File** y usar la opción **New File**.





Tras teclear el programa, será necesario acceder al menú **Run** y escoger la opción **Run Module**.

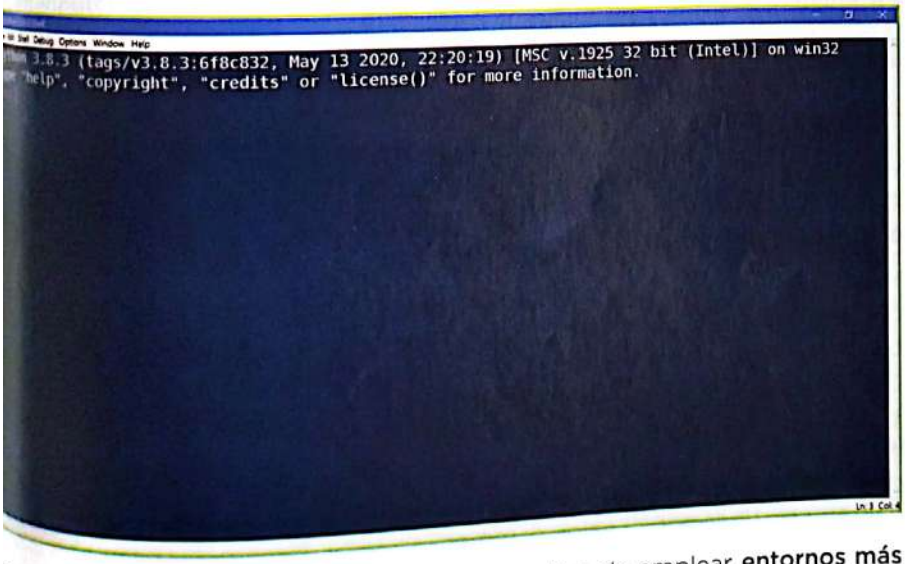
Si el programa no se hubiera guardado, en ese momento se dará la posibilidad para ello.

Probar un programa en Windows

ndows, al contrario que LliureX, no incluye Python preinstalado, pero se puede cargar e instalar, de forma sencilla, desde su página web, www.python.org.

es instalarlo, tanto el entorno básico como **Idle** estarán disponibles en el menú e **Inicio**, dentro de la carpeta **Python**.

manejo de ambos será idéntico al de la versión para Lliurex o cualquier otro stema operativo.



ra programas más complejos, existe la alternativa de emplear **entornos más anzados,** algunos de los cuales cuentan con versiones gratuitas para uso neral o para uso educativo, como PyCharm, creado por JetBrains.

Actividades

1. Crea un programa que escriba tu nombre en pantalla y pruébalo.
2. Diseña un programa que primero escriba «Hola» y después, en la siguiente línea, tu nombre.
3. Si no estás usando un entorno web, sino uno de escritorio (como Lliurex, Windows o macOS), comprueba el contenido de la carpeta en la que has dejado tu programa fuente.
   a) ¿Aparece en ella algún ejecutable (fichero con el mismo nombre que tu programa, pero terminado en una extensión como **.exe**)?
   b) A partir de la observación de esa carpeta, ¿opinas que Python es un lenguaje compilado o interpretado? Argumenta la respuesta en tu cuaderno.

2 UN PROGRAMA QUE CALCULA

2.1. Realizar operaciones prefijadas

Realizar operaciones matemáticas en Python es fácil: basta con indicar la operación **sin encerrarla entre comillas:**

```
print( 5 + 7 )
```

→ 12

Si se desea, es posible incluir **espacios** adicionales entre los paréntesis, o separar los números de los operadores, para conseguir una **mejor legibilidad.**

Las operaciones de **suma, resta, multiplicación** y **división** son sencillas. Sin embargo, una operación muy habitual en programación, pero que suele resultar más difícil de comprender, es el **resto** o **módulo:** el resultado de 21%5 es el resto de dividir 21 entre 5; es decir, 1:

```
print( 21 + 5 )
print( 21 - 5 )
print( 21 * 5 )
print( 21 / 5 )
print( 21 % 5 )
```

→
```
26
16
105
4.2
1
```

Otras dos operaciones frecuentes que permite realizar Python, y que no están disponibles en todos los lenguajes de programación, son la **potencia** (que se expresa con **dos asteriscos** seguidos) y la **división entera** (sin decimales, con **dos barras** de división seguidas):

```
print( 21 ** 2 )
print( 21 // 2 )
```

→
```
441
10
```

2.2. Escribir varios textos

Es posible escribir varios mensajes en la misma línea con una única orden **print**. Para ello, solo es necesario separarlos mediante **comas.** Cada coma se reflejará en pantalla como un espacio en blanco, separando los distintos mensajes. Por supuesto, se pueden incluir tanto **textos prefijados** (entre comillas) como **operaciones** (sin comillas).

```
print( "21 + 5 =", 21 + 5 )
```

→ 21 + 5 = 26

Si se desea que una orden **print** no avance a la línea siguiente, porque otra posterior deba mostrar su resultado en la misma línea, la primera de ambas deberá terminar con **end=""**. **Esta alternativa no incluye espacios intermedios en blanco.**

```
print( "21 + 5 = ", end="" )
print( 21 + 5 )
```

→ 21 + 5 = 26

Desde Python 3.6, se permite usar **cadenas con formato** (en inglés, *f-strings*, abreviatura de *formatted string literals*). Estas comienzan con una letra **f** e indican, entre llaves, aquello que se deba analizar y sustituir por su valor, como en este ejemplo:

```
print( f"21 + 5 = {21+5}")
```

→ 21 + 5 = 26

Operaciones básicas en Python

Operador	Operación
+	Suma
-	Resta, negación
*	Multiplicación
**	Potencia
/	División
//	División entera
%	Resto

2.3. Comentarios

Las líneas que comiencen por un símbolo de **almohadilla** o *hash* (**#**) se consideran **comentarios.** Estos se incluyen para **ayudar** al programador o la programadora a recordar mejor qué se pretendía **con un fragmento** del programa o cuál es **su cometido.** Para el ordenador, se comportan como si no se hubiera escrito nada:

```
# prueba de la potencia y la división entera
print( 21 ** 2 )
print( 21 // 2 )
```

Uso de caracteres internacionales

En los comentarios y en los textos que se muestran en pantalla, se puede escribir texto con acentos, eñes y cualquier otro carácter internacional.

Por el contrario, en los nombres de las variables, es recomendable usar solo el alfabeto inglés, como buena costumbre aplicable a otros lenguajes de programación que sean menos tolerantes que Python.

En versiones antiguas de Python (anteriores a Python 3), quizá incluso los acentos en los comentarios o en los textos en pantalla sean problemáticos. En ese caso, será necesario añadir al principio del programa la siguiente línea:

```
# coding: utf-8
```

2.4. Pedir datos a la persona usuaria

Un programa en el que todos los datos están prefijados es poco útil; no tiene mucho sentido que sea preciso modificar el código fuente y relanzar el programa cada vez que se hace necesario probar otros nuevos. Por el contrario, lo habitual es que esos **datos** sean **introducidos** por el usuario o la usuaria —la alternativa más sencilla— o se **lean desde un fichero, una base de datos** o una **red de ordenadores** (Internet o cualquier otra).

Para **leer datos** en Python, se emplea la orden **input()**, y será necesario disponer de un lugar donde guardarlos, un **espacio de memoria** al que se le dará un nombre: es lo que se conoce como una ***variable.*** En otros lenguajes de programación, es necesario declarar aquellas que un programa va a emplear, detallando, además, el tipo de datos que se van a guardar (por ejemplo, si será un texto o un número entero). En Python, esto no es necesario, ya que el propio **sistema se encargará de deducir** qué **tipo de datos** debe guardar cada variable **y de almacenarlos** correctamente. Por ejemplo, el siguiente programa pregunta a la persona usuaria su nombre y luego la saluda:

```
print("Dime tu nombre:")
nombre = input()
print("Hola,", nombre)
```

```
Dime tu nombre:
Enrique
Hola, Enrique
```

Ese programa puede abreviarse un poco, porque, entre los paréntesis de la orden **input()**, se puede indicar un mensaje de aviso, que ayude al usuario o la usuaria a saber qué tipo de información debe introducir. En este caso, y al contrario que en el ejemplo anterior, el texto se introducirá en la misma línea, a continuación del aviso, y sin ningún espacio adicional de separación:

```
nombre = input("Dime tu nombre: ")
print("Hola,", nombre)
```

```
Dime tu nombre: Enrique
Hola, Enrique
```

En ambos ejemplos, se pide a la persona que introduzca su nombre, y el texto que teclee se almacena en un espacio de memoria, al que se podrá acceder usando la palabra «nombre». Esta palabra se ha elegido por legibilidad, pero también podría haber sido algo tan breve como «n», o algo más detallado, como «nombreDelUsuario». Los **nombres** de las variables deben **empezar por una letra, no deben contener espacios** intermedios, y **respetar las mayúsculas y minúsculas** que se hayan escogido, de modo que el siguiente programa daría un mensaje de error al llegar a la segunda línea, porque la palabra «nombre» empieza por mayúsculas en la orden **print**:

```
nombre = input("Dime tu nombre: ")
print("Hola,", Nombre)
```

```
Dime tu nombre: Enrique
Traceback (most recent call last):
  File "saludo.py", line 2, in <module>
    print("Hola,", Nombre)
NameError: name 'Nombre' is not defined
```

2.5. Operaciones con datos numéricos introducidos por el usuario o la usuaria

Si lo que se desea es que el usuario o la usuaria **introduzca un número,** será necesario indicar a Python que el resultado de **input()** debe ser tratado como un número. En programación, se suele distinguir entre **números enteros** (sin decimales) y **números reales** o **de coma flotante** (que sí permiten decimales, pero pueden tener cierta pérdida de precisión y, según el sistema concreto, ser algo más lentos de manipular que los números enteros).

Así, el siguiente programa pide un número entero a la persona usuaria y, cuando se introduce, calcula su triple:

```
# Triple de un número
n = int ( input("Dime un numero: ") )
print("El triple de tu numero es", n*3)
```

```
Dime un numero: 15
El triple de tu numero es 45
```

Tal como se puede observar, para que un dato introducido por la persona se tome como **número entero,** basta con encerrarlo dentro de **int()**.

Es habitual que un programa emplee **varias variables,** tanto si se van a solicitar varios datos al usuario o la usuaria como si se van a realizar cálculos intermedios. Por ejemplo, se podría calcular la suma de dos números reales así:

```
# Sumar dos datos introducidos por el usuario
n1 = float ( input("Dime un número: ") )
n2 = float ( input("Dime otro número: ") )
suma = n1 + n2
print("Su suma es", suma)
```

```
Dime un numero: 23
Dime otro numero: 58
Su suma es 81
```

2.6. Funciones matemáticas

Existen muchas funciones matemáticas incorporadas en Python, como estas:

- Raíz cuadrada: **math.sqrt(x)**
- x elevado a y: **math.pow(x,y)**
- Coseno: **math.cos(x)**
- Seno: **math.sin(x)**
- Tangente: **math.tan(x)**
- Exponencial de x (*e* elevado a x): **math.exp(x)**
- Logaritmo natural (o neperiano) en base *e:* **math.log(x)**
- Logaritmo en base 10: **math.log10(x)**

Estas funciones **no son parte del lenguaje base,** sino de la **biblioteca math**, por lo que, para utilizarlas, el programa deberá comenzar con la línea **import math**, como en el siguiente ejemplo:

```
# Raíz cuadrada de un número
import math
n = float ( input("Dime un número: ") )
raiz = math.sqrt(n)
print( "Su raíz cuadrada es", raiz )
```

```
Dime un numero: 8
Su raíz cuadrada es
2.8284271247461903
```

División de números [illegible]

Al contrario que en algunos lenguajes de programación (en general, C y los que derivan de él, como C++ o Java), la división de dos números enteros se calcula con decimales en Python, es decir **print(7/2)** escribirá 3.5, mientras que en otros lenguajes el resultado de la orden equivalente es un número sin decimales.

Actividades

4 Crea un programa que calcule la diferencia entre 32 y 37, y otro que calcule el producto de 84 y -23.

5 Realiza un programa que pida dos números enteros y calcule la división del primero entre el segundo.

Para las **funciones trigonométricas,** el ángulo se indica en **radianes** en vez de en grados. Así, si el dato original está en grados, habrá que convertirlo a radianes, teniendo en cuenta la equivalencia 180 grados = π radianes.

El número π, en Python, se representa como **math.pi**, tal como se puede ver en el siguiente ejemplo:

```
# Funciones matemáticas: coseno

import math

anguloGrados = 45
anguloRadianes = anguloGrados * math.pi / 180
print( "El coseno de", anguloGrados, "es", math.cos(anguloRadianes) )
```

→ El coseno de 45 es 0.7071067811865476

2.7. Partir líneas largas

A pesar de que, por lo general, Python es más compacto que otros lenguajes de programación, en ocasiones una orden ocupará más que el ancho visible de la pantalla, lo que puede disminuir la legibilidad del programa, al tener que hacer *scroll* a un lado y a otro, para revisarlo.

En la práctica, se recomienda que una **línea no esté formada por más de 80 caracteres,** y que si una orden concreta va a ser más larga, **se descomponga en varias líneas.**

Existen dos formas de **partir líneas: dentro de un paréntesis,** se podrá partir la línea (lo deseable, tras una coma o un operador matemático) y **tabular** la continuación **a la derecha** (habitualmente, cuatro espacios); o bien, **en líneas que no contengan paréntesis,** se puede emplear la **barra invertida** (\ ), para indicar que una determinada **orden no está completa,** sino que continúa en la línea siguiente:

→ Dentro de un paréntesis

```
# Coseno y orden que abarca dos líneas (1)

import math

anguloGrados = 45
anguloRadianes = anguloGrados * math.pi / 180
print( "El coseno de", anguloGrados, "es",
    math.cos(anguloRadianes) )
```

→ En líneas que no contengan paréntesis

```
# Coseno y orden que abarca dos líneas (2)

import math

anguloGrados = 45
anguloRadianes = anguloGrados * \
    math.pi / 180
print( "El coseno de", anguloGrados, "es",
    math.cos(anguloRadianes) )
```

Actividades

6 Crea un programa que calcule la raíz cuadrada del número que elija el usuario o la usuaria.

7 Diseña un programa que calcule la raíz cúbica del número introducido por la persona usuaria (pista: deberás elevar ese número a 1/3).

8 Elabora un programa que pida, al usuario o la usuaria, una cantidad de millas terrestres y calcule su equivalente en kilómetros (1 milla = 1,609 km).

9 Realiza un programa que pida cinco números reales a la persona usuaria y calcule su media.

10 Diseña un programa que calcule el seno de ángulos con los siguientes grados: 30, 45, 60 y 90°.

11 Si se ingresan *E* euros en un banco, con un interés *R*, durante *N* periodos de tiempo, el dinero que se obtendrá finalmente vendrá dado por el interés compuesto $(E \cdot (1+R)^N)$. A partir de estos datos, compón un programa que calcule el interés para los datos introducidos la persona usuaria.

3 TOMA DE DECISIONES

3.1. if

Para **comprobar** si se cumple una **determinada condición** e indicar qué pasos se deben dar en dicho caso, se emplea la orden **if** («si» condicional). Una vez escrita, tras la palabra **if**, se señalará la condición que se desea evaluar, seguida de dos puntos (:).

La orden u órdenes que se deban realizar, en caso de que se cumpla esa condición, se detallará en la línea o líneas siguientes, tabulando un poco más a la derecha (de nuevo, cuatro espacios es el estándar más aceptado por regla general):

```
# Condiciones con if: comprobar si un número es positivo

numero = int ( input("Escribe un numero: ") )
if numero > 0 :
    print("El numero es positivo.")
```

```
Escribe un numero: 5
El numero es positivo.
```

3.2. Operadores relacionales: <, <=, >, >=, ==, !=

Igual que se observa en el ejemplo del apartado anterior, el símbolo **mayor que** (**>**) se emplea para comprobar si un número es mayor que otro. Por su parte, el símbolo **menor que** (**<**) también es sencillo, pero los demás operadores de comparación son un poco menos evidentes, tal como se puede observar en la siguiente tabla:

Relación de operadores y sus significados

Operador	Operación
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual a
!=	No igual a (Distinto de)

Los espacios intermedios

Los espacios intermedios junto a los paréntesis y operadores no son necesarios, pero en ocasiones el programa puede resultar más legible con espacios adicionales.

A efectos prácticos, es lo mismo escribir

```
if x > y :
    print( "x es mayor" )
```

que esta alternativa más compacta

```
if x>y:
    print("x es mayor")
```

```
# Comprobar el valor de una variable

numero = int ( input("Escribe un numero: ") )
if numero == 0 :
    print("Has escrito cero.")
```

```
Escribe un numero: 0
Has escrito cero.
```

Es importante tener en cuenta que los operadores formados por **dos símbolos**, como >=, **no** deben tener un **espacio** entre ambos símbolos, o no se reconocerán correctamente, tal como muestra el siguiente ejemplo:

```
# Error al comprobar el valor de una variable

numero = int ( input("Escribe un numero: ") )
if numero > = 0 :
    print("Es positivo o cero.")
```

```
File "main.py", line 3
  if numero > = 0:
               ^
SyntaxError: invalid syntax
```

3.3. El caso contrario: else

También es habitual indicar lo que debe hacer el programa **si no se cumple una determinada condición,** para lo cual se añade la cláusula **else**. Al igual que ocurre con el bloque que sigue a **if**, las órdenes que siguen a **else** deben tabularse a la derecha:

```
# if y else

numero = int ( input("Escribe un numero: ") )
if numero == 0 :
    print("Has escrito cero.")
else :
    print("Has escrito un numero distinto de cero.")
```

Cláusula else

En Python y otros muchos lenguajes de programación, la palabra «else» no puede existir por sí sola, sino como parte opcional de una orden **if**, y según cada lenguaje concreto, quizá de alguna otra. Por eso, no se suele hablar de «la orden **else**», sino de «la cláusula **else**».

3.4. Bloques de órdenes

Tal como ya se ha anticipado, cuando sea necesario **ejecutar varias órdenes** al cumplirse una condición, estas deberán **tabularse** más **a la derecha:**

```
# Bloques de órdenes

numero = int ( input("Escribe un numero: ") )
if numero == 0 :
    print("Has escrito cero.")
    print("Tambien puedes escribir negativos.")
else :
    print("Has escrito un numero distinto de cero.")
```

```
Escribe un numero: 0
Has escrito cero.
Tambien puedes escribir negativos.
```

```
Escribe un numero: 2
Has escrito un numero distinto de cero.
```

Un ejemplo adicional puede ser multiplicar dos números introducidos por la persona usuaria, pero pidiendo el segundo número solo en caso de que el primero no sea cero:

```
print("Vamos a multiplicar dos números...")
n1 = int ( input("Dime el primero: ") )
if n1 == 0 :
    print("Al multiplicar por cero obtendrás cero.")
else :
    n2 = int ( input("Dime el segundo: ") )
    print("Su producto es... ", end ="")
    print(n1*n2)
```

```
Vamos a multiplicar dos números...
Dime el primero: 0
Al multiplicar por cero obtendrás cero.
```

```
Vamos a multiplicar dos números...
Dime el primero: 2
Dime el segundo: 3
Su producto es... 6
```

Actividades

12 Crea un programa que pida un número entero al usuario o la usuaria e indique si es par o no. Para ello, deberá comprobar si el resto que obtiene al dividir dicho número entre dos es 0: **if numero % 2 == 0**.

13 Realiza un programa que pida a la persona usuaria dos números enteros y diga cuál es mayor.

14 Elabora un programa que pida dos números a la persona usuaria. Si el segundo no es 0, mostrará la división de ambos; en caso contrario, aparecerá un mensaje de error.

15 Crea un programa que pida dos números e indique si el primero es múltiplo del segundo.

3.5. Encadenar condiciones: and, or, not

Las condiciones se pueden **encadenar** con **and** (y), **or** (o), **not** (no), etc., igual que se indica en la tabla del margen. Así, se pueden escribir expresiones como:

Relación de operadores y sus significados

Operador	Significado
and	Y
or	O
not	No

```
# Condiciones múltiples

numero = int ( input("Escribe un numero: ") )
if not (numero == 0):
    print ("No es cero.")
if (numero == 2) or (numero == 3):
    print ("Es dos o tres.")
if (numero >= 2) and (numero <= 7):
    print ("Esta entre 2 y 7 (incluidos).")
```

```
Escribe un numero: 3
No es cero.
Es dos o tres.
Esta entre 2 y 7 (incluidos).
```

3.6. Expresiones condicionales

Existe otra forma de **asignar un valor a una variable,** en función de **si se cumple una condición** o no. Se trata de la ***expresión condicional*** u ***operador ternario,*** que equivale a otra forma de escribir la orden **if**, en la que actúa como operador.

A título de ejemplo, un planteamiento habitual para calcular el mayor de dos números sería:

```
# if "convencional"

a = 3
b = 5
if a>b:
    mayor = a
else:
    mayor = b
print ("El mayor es", mayor)
```

```
El mayor es 5
```

Pero Python permite también formularlo de la siguiente forma, más compacta pero (según el caso) también algo menos legible:

```
# if como expresión condicional

a = 3
b = 5
mayor = a if a>b else b
print ("El mayor es", mayor)
```

```
El mayor es 5
```

Es decir, su sintaxis es:

```
variable = valor1 if condición else valor2
```

Equivale a decir «si se cumple la condición, toma el primer valor; si no se cumple, toma el segundo valor».

Otro ejemplo sería una forma compacta de ver si un número es par:

```
dato = 6
esPar = "Si" if dato%2 == 0 else "No"
print ("Es par?", esPar)
```

```
Es par? Si
```

3.7. elif

Cuando se deben analizar varias **condiciones consecutivas,** el hecho de tener que tabular cada nuevo bloque más a la derecha puede hacer que el programa adquiera la apariencia de una escalera y resulte más **difícil de leer,** tal como ocurre con el siguiente programa, que escribe el nombre de los números del 1 al 5:

```
# Condiciones sin elif

numero = int ( input("Introduce un numero del 1 al 5: ") )
if numero == 1 :
    print("Uno")
else :
    if numero == 2 :
        print("Dos")
    else :
        if numero == 3 :
            print("Tres")
        else :
            if numero == 4 :
                print("Cuatro")
            else :
                if numero == 5 :
                    print("Cinco")
                else :
                    print("Valor incorrecto!")
```

La alternativa es usar la orden **elif**, que permite **enlazar** los **else : if** de una forma más compacta y evitar ese aumento de las tabulaciones:

```
# Condiciones con elif

numero = int ( input("Introduce un numero del 1 al 5: ") )
if numero == 1 :
    print("Uno")
elif numero == 2 :
    print("Dos")
elif numero == 3 :
    print("Tres")
elif numero == 4 :
    print("Cuatro")
elif numero == 5 :
    print("Cinco")
else :
    print("Valor incorrecto!")
```

En otros lenguajes, como los que proceden de C, existe una orden **switch**, para comprobar varios posibles valores de una variable. Esa orden no existe en Python, sino que se debe emplear bloques **if-elif**. Este planteamiento hace el programa un poco menos legible, pero también más versátil, porque, normalmente, la orden **switch** solo permite ver valores exactos, y no operadores como **>** o **<=**.

Actividades

16 Elabora un programa que pida, al usuario o la usuaria, dos números enteros y diga: «Uno de los números es positivo», «Los dos números son positivos» o bien «Ninguno de los números es positivo», según corresponda.

17 Crea un programa que pida tres números reales e indique el valor numérico del mayor de ellos (no que informe de cuál es su posición, ya que algún valor puede estar repetido).

18 Diseña, usando **if encadenados** en vez del operador **%**, un programa que pida un número del 1 al 10 y diga si es múltiplo de 3 o no.

19 Crea un programa, usando **elif** en vez de **%**, que pida un número del 1 al 10 y diga si es múltiplo de 3 o no.

20 Haz, usando **elif**, un programa que pida un número enteró del 1 al 10 y escriba el nombre de la nota correspondiente (por ejemplo, tanto para el 7 como para el 8 deberá escribir «Notable»).

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
```

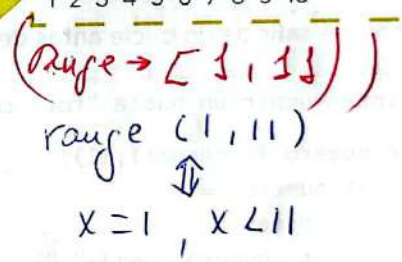
4.4. for

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,
```

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
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 3×3, 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).

7.1. 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.