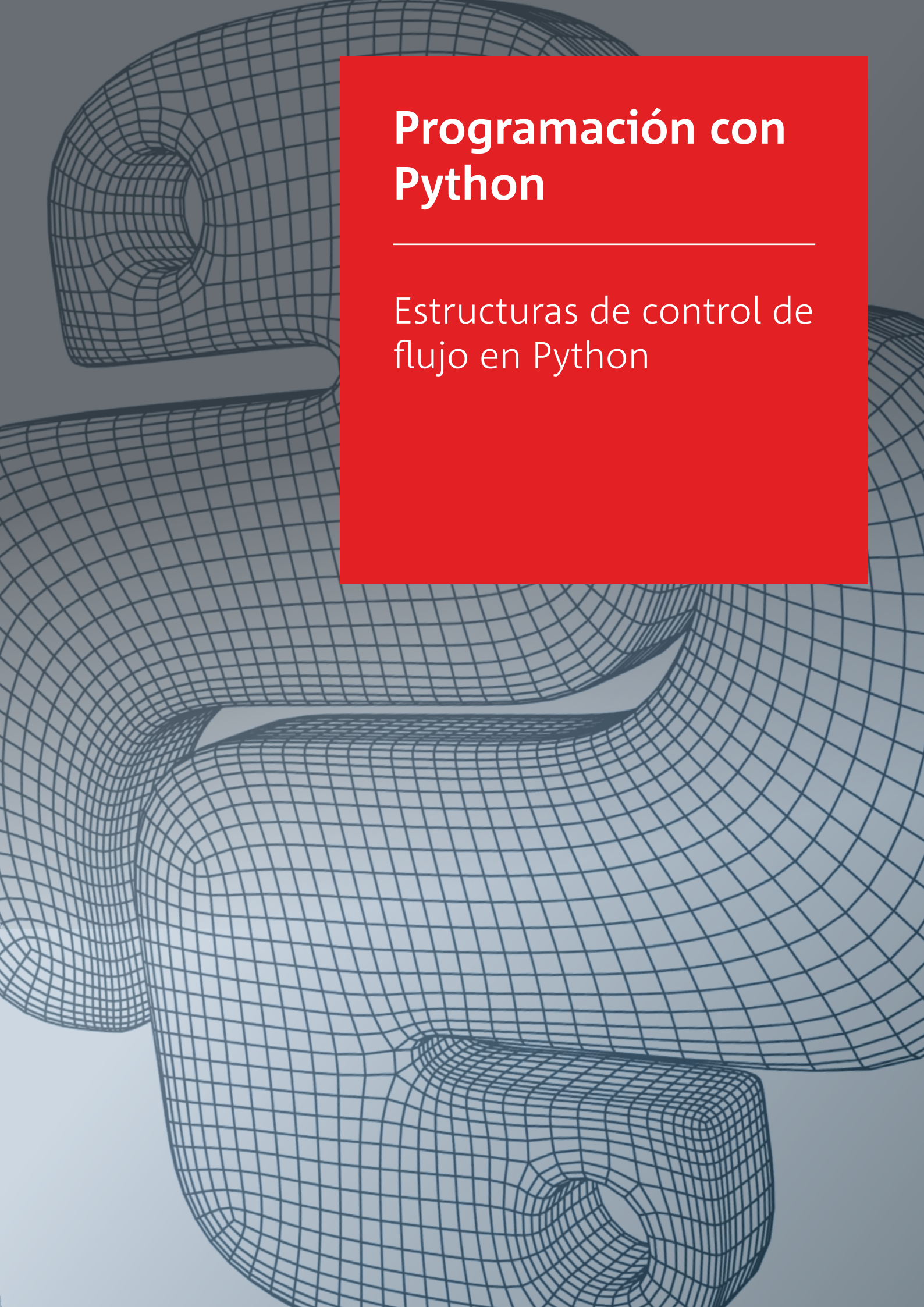

Programación con Python

Estructuras de control de flujo en Python

INICIATIVA Y COORDINACIÓN
DEUSTO FORMACIÓN

COLABORADORES
Realización:
E-Mafe E-Learning Solutions S.L.

Elaboración de contenidos:
Claudio García Martorell
Licenciado IT Telecomunicaciones especialidad Telemática.
Postgrado en Sistemas de Comunicación y Certificación en Business Intelligence TargIT University.
Concejal de Innovación y Tecnología.
Ponente y docente en distintas universidades y eventos.
Josep Estarlich Pau
Técnico de Ingeniería Informática.
Director Área de Software de la empresa Dismuntel.
Participante en proyectos con Python, C#, R y PHP orientados a *Machine Learning* y a la Inteligencia Artificial.

Supervisión técnica y pedagógica:
Grufium educación y excelencia

Coordinación editorial:
Grufium educación y excelencia

Barcelona (España), 2021

Primera edición: septiembre 2021

ISBN: 978-84-1300-688-8 (Obra completa)
ISBN: 978-84-1300-692-5 (Estructuras de control de flujo en Python)

Depósito Legal: B 11038-2021

Impreso por:
SERVINFORM
Avenida de los Premios Nobel, 37
Polígono Casablanca
28850 Torrejón de Ardoz (Madrid)

Printed in Spain
Impreso en España

Esquema de contenido

1. Sangrado, ejecución condicional y control o acceso de variables

- 1.1. Sangrado
- 1.2. Ejecución condicional
- 1.3. Control o acceso de variables: variables y su ámbito

2. Iteraciones: iteratividad y recursividad

- 2.1. Iteratividad
- 2.2. Recursividad

3. Instrucciones *break-continue*: usos y depuración de errores

- 3.1. Instrucción *break*
- 3.2. Instrucción *continue*
- 3.3. Instrucción *pass*
- 3.4. Conclusión sobre *break, continue* y *pass*

4. Instrucciones *raise* y *try-except-finally*: usos y depuración de errores

- 4.1. Control o manejo de excepciones
- 4.2. Propagar la excepción
- 4.3. Información de la excepción (tipo,valor, etc.)
- 4.4. Excepciones definidas por el programador
- 4.5. Algoritmos de búsqueda y ordenación
- 4.6. Depuración de errores

Introducción

En este módulo vamos a aprender a agrupar instrucciones en Python; con ello, estaremos listos para realizar comparaciones y evaluaciones de variables mediante las palabras reservadas *if*, *elif* y *else*.

Refrescaremos la utilización del sangrado, ya que su uso incorrecto afectará a la funcionalidad del código: no se ejecutará el código esperado, o se ejecutará código que solo debería haberse ejecutado de confirmarse una condición como verdadera.

Estudiaremos las iteraciones, que dividiremos en iterativas y recursivas. Veremos en detalle los bucles *while* y *for*, y su uso en Python, así como el uso de autollamadas de la misma función para realizar recorridos o iteraciones recursivas sobre el código fuente.

1. Sangrado, ejecución condicional y control o acceso de variables

En este punto recordaremos la importancia del sangrado o indentación, y estudiaremos la estructura condicional para empezar nuestros algoritmos, así como sus operadores para utilizar combinaciones de validaciones condicionales.

También revisaremos las variables y el ámbito en el que se encuentran para realizar la operación deseada.

1.1. Sangrado

Para empezar a crear nuestros algoritmos y funciones en Python, debemos tener claro el concepto de sangrado o indentación, necesario para poder continuar con las estructuras de control.

El sangrado en Python es obligatorio; se trata de una sangría en los códigos con el tabulador o con 4 espacios, dentro de funciones o bucles. De esta sangría dependerá la estructura de nuestro código fuente, tal como en Java o en C# utilizamos las claves (“{ }”) para abrir y cerrar nuestra estructura de código.

Además de ser una obligación, va a conseguir que nuestro código fuente sea mucho más legible y que cualquier otro desarrollador que intente leer nuestro código pueda comprenderlo más rápidamente; en este sentido, Python fue diseñado para ser leído con facilidad.

Recomendamos no mezclar en un mismo código espacios y tabulaciones, ya que en algunos momentos nos puede llevar a confusión en el código fuente y a errores de ejecución. Aquí podemos comprobar el error que nos indica el intérprete por no respetar el sangrado.

```
def Suma(a,b):
c = a + b
return c

print(Suma(1,2))
```

Certificación

Es importante interiorizar el concepto de sangrado pues en numerosas preguntas de la certificación se hace referencia a este.

Salida:

```
File “<ipython-input-1-023614362498>”,
line 2 c = a + b
       ^
IndentationError: expected an indented block
```

Y aquí, el mismo código, con un uso correcto de los sangrados:

```
def Suma(a,b):
    c= a + b
    return c

print(Suma(1,2))
```

Salida:

```
3
```

Por otro lado, cada instrucción debe situarse en una sola línea del código; pero si, por cuestiones de espacio o con la premisa de mejorar la legibilidad, decidimos escribir el código en más de una línea, podemos dividir la instrucción incluyendo al final de dicha línea una barra invertida (“\”); así indicaremos que la instrucción continúa en la siguiente línea.

```
lista=[‘valor 1’,’valor 2’ \
        ,’valor 3’]
  cadena=’Esta es una cadena ‘ \
         ‘bastante larga’
```

Certificación

Es importante interiorizar el concepto de ejecución condicional pues en numerosas preguntas de la certificación se hace referencia a este.

1.2. Ejecución condicional

Las estructuras de control condicionales son las que van a permitirnos evaluar en nuestro código fuente si una o varias condiciones son verdaderas (“True”) o falsas (“False”), y decidir qué función o código vamos a ejecutar en relación con el resultado de la condición.

Empezamos por la funcionalidad *if* (“condición”); de esta función solo vamos a obtener un resultado, que podrá ser “True” si la condición es verdadera o “False” si la condición no se cumple (Figura 1.1).

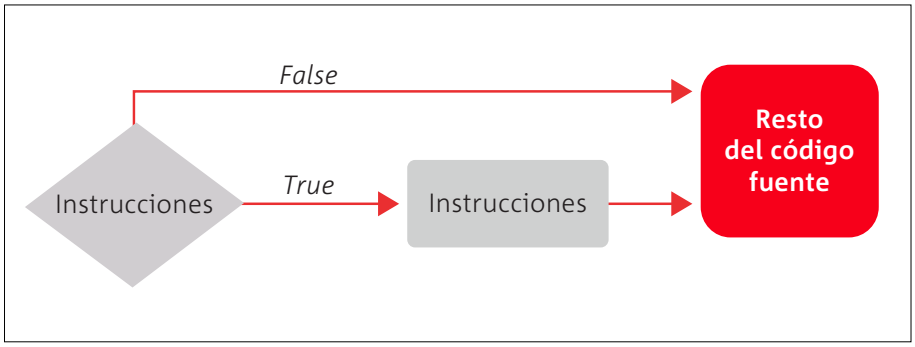

Figura 1.1

Flujo de una condición *if*.

También debe ir acompañada de los dos puntos finales (":") para indicar que empieza la definición del código que se ejecutará en caso de que la condición sea "True". Para evitar el error de sangrado, la siguiente línea del *if* debe contener la tabulación o los espacios correspondientes; así, el intérprete nos permitirá ejecutar nuestro código. Si incluimos más de una línea dentro del *if*, debemos mantener este sangrado/tabulado para que se ejecute si la condición es "True". Si se encuentra sin el tabulado, se ejecutará siempre, independientemente de que la evaluación del bucle devuelva "True" o "False".

```
def Suma(a,b):
    if(a == 1):
        a = 3
    print('Estoy fuera del if')
    c= a + b
    return c

print(Suma(2,2))
```

Salida:

```
Estoy fuera del if
4
```

```
def Suma(a,b):
    if(a == 1):
        a = 3
        print('Estoy dentro del if')
    c= a + b
    return c

print(Suma(2,2))
```

Salida:

```
4
```

Una vez reforzado esto, nos hacemos la siguiente pregunta: ¿Qué operadores utilizamos para validar la condición de un *if*?

Símbolo	Significado	Ejemplo	Resultado
==	Igual que	2 == 7	False
!=	Distinto de		
“rojo” != “azul”	True		
<	Menor que	2 < 7	True
>	Mayor que	25 > 1	True
<=	Mayor o igual que	15 <= 15	True
>=	Menor o igual que	11 >= 17	False

Si necesitamos validar más de una condición en un mismo if, podemos utilizar los operadores lógicos *and*, *or* y x*or*:

Operador	Ejemplo	Resultado individual	Resultado
and	12 < 25 and 12 == 13	True and False	False
or	12 < 25 or 12 == 13	True or False	True
xor	12 < 25 xor 12 == 13	True xor False	True

Las estructuras de control de flujo condicional se van a basar en tres palabras reservadas en Python: *if* (“si”), *elif* (“en cambio” + “si”) y *else* (“en cambio”). Por ejemplo:

```
edad = int(input(“¿Cuántos años tienes? “))
if(edad >= 18):
    print(“Eres mayor de edad”)
else:
    print(“Eres menor de edad”)
```

Ejercicios resueltos

Practica el **ejercicio 1** con los conocimientos adquiridos hasta esta unidad.

Salida:

```
¿Cuántos años tienes? 17
Eres menor de edad
```

En el primer *if* comprobamos si los años tecleados son mayores o iguales que 18; si se cumple esta condición, mostramos por pantalla "Eres mayor de edad"; en cambio, si la edad es menor que 18, entramos en la parte del *else* y mostramos por pantalla "Eres menor de edad".

Por otro lado, podemos tener la necesidad de realizar una comprobación adicional. Para ello, en Python utilizaremos *elif*, que nos permite evaluar una comprobación adicional si la primera condición del *if* no se cumple; en ese caso, se evaluará la que está definida dentro del *elif* declarado en el código fuente. Por ejemplo:

```
edad = int(input("¿Cuántos años tienes? "))
if(edad >= 18):
    print("Eres mayor de edad")
elif (edad > 16):
    print("Tienes más de 16 años")
else:
    print("Eres menor de edad")
```

Salida:

```
¿Cuántos años tienes? 17
Tienes más de 16 años
```

Comprobaremos que el uso excesivo de *if*, *elif* o *else* en el código fuente puede hacer que este resulte ilegible, por lo que puede ser conveniente la utilización de la instrucción *switch*. Aunque varios lenguajes como Java o C# sí que incluyen esta instrucción, en Python no existe como tal; por ello vamos a crear nuestras alternativas mediante diccionarios.

Esta alternativa será mucho más eficiente en tiempo de ejecución que realizar todas las comprobaciones mediante *if*, *elif* o *else*; y, por otro lado, la lectura del código fuente será mucho más fluida. Por ejemplo:

```
diccionario = {
    1: "Concierto Rock",
    2: "Concierto Heavy Metal",
    3: "Concierto Pop Rock"
}

print(diccionario)
valor = int(input("Selecciona un concierto con un número"))
print(diccionario.get(valor,"El concierto no existe"))
```

Ejercicios resueltos

Practica el **ejercicio 2** con los conocimientos adquiridos hasta esta unidad.

Salida:

```
{1: 'Concierto Rock', 2: 'Concierto Heavy Metal', 3: 'Concierto Pop Rock'}
Selecciona un concierto con un número 4
El concierto no existe
```

Ejercicios resueltos

Practica el **ejercicio 3** con los conocimientos adquiridos hasta esta unidad.

En el anterior ejemplo, se busca la clave en el diccionario; si se encuentra una coincidencia dentro de él, se imprimirá el valor de la *key* introducida por el teclado. En caso de que no se encuentre, se nos mostrará el mensaje "El concierto no existe".

Para entrar en más detalle y poder aplicar esta alternativa utilizando funciones, vamos a cambiar los *string* del ejemplo anterior por funciones; podremos ejecutar una mayor cantidad de código fuente, como si de un *switch* real se tratara.

```
def suma(a,b):
    valor = a + b
    return valor

def resta(a,b):
    valor = a - b
    return valor

def switch(operacion, a, b):
        diccionario = {
        "+": suma( a, b),
        "-": resta( a, b),
        }
return diccionario.get(operacion, lambda: """No se encuentra la operación""")

a = int(input("Introduce el valor de a: "))
b = int(input("Introduce el valor de b: "))
operacion = input("Selecciona una operación ( +, - )")
print(switch(operacion, a, b))
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
Introduce el valor de a: 2
Introduce el valor de b: 3
Selecciona una operación ( +, - )+
5
```

En este ejemplo podemos comprobar que hemos creado dos funciones que se encargarán de realizar las operaciones deseadas y otra función que

decidirá qué función ejecutar dependiendo de la operación seleccionada y realizando la búsqueda en el diccionario mediante la función “get()”.

1.3. Control o acceso de variables: variables y su ámbito

Como hemos aprendido anteriormente, las variables son registros donde almacenaremos resultados, valores en nuestro programa, etc.

En cualquier lenguaje de programación de alto nivel, estas variables declaradas estarán definidas dentro de un ámbito donde se van a poder utilizar.

Las variables que se declaran dentro de una función tendrán un ámbito local; no podrán ser utilizadas fuera de esta, ya que no serán reconocidas.

Al hablar del ámbito de las variables, debemos referirnos al ciclo de vida de estas, ya que determinará el tiempo en que permanecen en memoria. Una variable de una función estará en memoria solo durante su ejecución; una vez está terminada la variable, deja de estar en memoria y no se puede volver a referenciar.

```
def suma( a, b ):
    valor = a + 0
    return valor

print( suma(1,2))
print(valor)
```

Salida:

```
1
---------------------------------------------
NameError                                 Traceback (most recent
call last)
<ipython-input-7-df68d8bbe5c2> in <module>()
      4
      5 print( suma(1,2))
----> 6 print(valor)
NameError: name ‘valor’ is not defined
```

En el ejemplo podemos observar que la variable usada dentro de la función no puede utilizarse externamente, ya que para el intérprete no existe, y se nos mostrará un error.

Recuerda

Cuando hablamos de ámbito, nos referiremos a la zona del código fuente en la que se define una variable local o global.

De las variables definidas fuera de las funciones se dice que utilizan un ámbito global y se denominan *globales*; el valor de estas variables se podrá consultar tanto dentro como fuera de las funciones.

```
x = 31
y = 12
def funcion():
    x = 10
    print(f'x vale {x}')
    print(f'y vale {y}')

funcion()
print(x)
```

Salida:

```
x vale 10
y vale 12
31
```

En el siguiente ejemplo observamos que tenemos dos variables nombradas de igual forma; sin embargo, como no están en el mismo ámbito, cada una mantiene su valor. Por otro lado, podemos observar que la variable *y* sí permite realizar la lectura sobre dicha variable dentro de la función.

Si queremos modificar el valor de una variable de ámbito global dentro de una función, debemos utilizar la palabra reservada *global* para declarar la variable dentro de la función y poder modificarla con una asignación a dicha variable.

```
x = 31
y = 12
def funcion():
    global x
    x = 10
    print(f'x vale {x}')
    print(f'y vale {y}')

funcion()
print(x)
```

Salida:

Para saber más

Cuando en una función se hace referencia a una variable, esta se busca primero en el ámbito local, después en el global y finalmente en el espacio reservado de Python. Si hay una función dentro de otra, se anida un nuevo ámbito dentro del ámbito local.

```
x vale 10
y vale 12
10
```

En este ejemplo podemos observar que, utilizando la palabra *global*, la variable x deja de contener el valor 31 y pasa a tomar el valor 10 que se declara dentro de la función.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Resumen

- El sangrado (o indentación) es importante en Python, ya que es una de las validaciones sintácticas que realiza a la hora de interpretar el código.
- Una estructura de control condicional se basa en la comparación o validación de una expresión para ejecutar o no otra parte del código.
- Se pueden concatenar varias validaciones dentro de una misma expresión para que la ejecución del código dependa de diversos factores.
- Es posible usar las variables dentro o fuera de su ámbito, y también llevar a cabo un acceso global a ellas.

2. Iteraciones: iteratividad y recursividad

Iteración significa repetir más de una vez un proceso para alcanzar una meta deseada; cada repetición también es denominada *iteración*, y los resultados obtenidos en cada una de ellas son utilizados como punto de partida en la siguiente.

En este punto vamos a hablar de iteraciones, y lo haremos separándolas en dos categorías: las iteraciones iterativas y las iteraciones recursivas.

2.1. Iteratividad

Para poder realizar iteraciones iterativas en este lenguaje que estamos aprendiendo, es necesario conocer las palabras reservadas *while* ("mientras") y *for* ("para").

While se encarga de ejecutar una acción repetidamente mientras que una condición siga siendo verdadera.

La sintaxis para este bucle es:

Certificación

Es importante interiorizar el concepto de iteración pues en numerosas preguntas de la certificación se hace referencia a este.

```
#Código antes del bucle (inicializar variables, #operaciones previas...).
while expresion_para_validar_si_continua_el_bucle:
#Código que se ejecutará tantas veces como #la condición se evalúe como "True".
#Código fuera del bucle que se ejecutará una vez #que la condición se evalúe como "False", y
solo #se ejecutará una vez.
```

Un error bastante común en este tipo de bucles es dejar una ejecución eterna del código que está incluido dentro de este; esto se denomina bucle infinito, y la única forma de pararlo es deteniendo la ejecución del programa.

```
c = 0
while c <= 5:
    print("Valor de c: ", c)
```

En este ejemplo, el error es que no estamos incrementando la variable *c*; por tanto, esta siempre tiene el valor 0 y nunca va a poder finalizar la

comprobación de *c* <= 5, ya que siempre devuelve "True", y el bucle *while* solo finaliza cuando el resultado de esta operación nos devuelve "False".

```
c = 0
while c <= 5:
    c+=1
    print("Valor de c: ", c)
```

Vamos a añadir un ejemplo en Colab, en el cual introduciremos un "print" que nos indicará si en la siguiente iteración volverá a ejecutar el código interno del *while* o dará por terminado dicho bucle marcando el valor de la condición como "False".

```
c = 0
while c <= 5:
    c+=1
    print("Valor de c: ", c)
    print(c <= 5)
```

Salida:

```
Valor de c:  1
True
Valor de c:  2
True
Valor de c:  3
True
Valor de c:  4
True
Valor de c:  5
True
Valor de c:  6
False
```

Continuando con el bucle *while*, Python nos permite la utilización del comando *else* para que sea ejecutado el código del interior de este una vez que el bucle haya terminado sus iteraciones.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

```
c = 0
while c <= 5:
    c+=1
    print("Valor de c: ", c)
    print(c <= 5)
else:
  print("El bucle ha finalizado.")
```

Salida:

```
Valor de c:  1
True
Valor de c:  2
True
Valor de c:  3
True
Valor de c:  4
True
Valor de c:  5
True
Valor de c:  6
False
El bucle ha finalizado.
```

Podemos utilizar el bucle *while* para recorrer listas, como en este ejemplo:

```
lenguajes = [“Python”, “C#”, “Java”]
i = 0

while i < len(lenguajes):
    print(lenguajes[i],”id --> “,i)
    i += 1
```

Salida:

```
Python id -->  0
C# id -->  1
Java id -->  2
```

También podemos utilizar “while(True)” para un menú que queramos que siempre esté en ejecución, en bucle infinito, y que, una vez llegado al final del *script*, vuelva a ejecutar el código desde el inicio.

```
print(“Este es nuestro menú infinito”)
while(True):
    print(“””Selecciona una opción:
    1) Saludar
    2) Sumar dos números
    3) Salir”””)
    opcion = input()
    if opcion == ‘1’:
        print(“Hola, estamos aprendiendo Python”)
    elif opcion == ‘2’:
```

```
        x = float(input("Introduce el primer número: "))
        y = float(input("Introduce el segundo número: "))
        print("El resultado de la suma es: ",x + y)
    elif opcion =='3':
        print("Cerramos, nos vemos ;) bye")
        break
    else:
        print("Comando desconocido, vuelve a intentarlo")
```

Salida:

```
Este es nuestro menú infinito
Selecciona una opción:
    1) Saludar
    2) Sumar dos números
    3) Salir
1
Hola, estamos aprendiendo Python
Selecciona una opción:
    1) Saludar
    2) Sumar dos números
    3) Salir
2
Introduce el primer número: 4
Introduce el segundo número: 2
El resultado de la suma es: 6.0
Selecciona una opción:
    1) Saludar
    2) Sumar dos números
    3) Salir
4
Comando desconocido, vuelve a intentarlo
Selecciona una opción:
    1) Saludar
    2) Sumar dos números
    3) Salir
3
Cerramos, nos vemos ;) bye
```

Ejercicios resueltos

Practica el **ejercicio 4** con los conocimientos adquiridos hasta esta unidad.

En Python también disponemos del bucle *for*. Es el más utilizado para iterar, ya que dispone de una sintaxis simple y nos permite realizar un recorrido sobre una lista de una forma más sencilla que con el bucle *while*.

```
#Código antes del bucle (inicializar variables, #operaciones previas...).
for <elemento> in <iterable>:
#Código que se ejecutará tantas veces como #la condición se evalúe como "True".
#Código fuera del bucle, se ejecutará solo una #vez cuando la condición se evalúe como "False".
```

La variable “elemento” es la que toma el valor del elemento dentro del iterador en cada iteración del *for*, e “iterable” es cualquier elemento iterable.

El bucle finaliza su ejecución cuando todos los elementos del “iterable” han sido recorridos.

Se utiliza frecuentemente para recorrer elementos de un objeto iterable (lista, tupla, conjunto, diccionario...), y en cada paso de la iteración se selecciona un único elemento del objeto iterable, el cual utilizaremos para realizar operaciones.

```
lenguajes = [“Python”, “C#”, “Java”]

for lenguaje in lenguajes:  # Para [variable] en [lista]
    print(lenguaje)
```

Salida:

```
Python
C#
Java
```

En este ejemplo hemos realizado la repetición del último bucle en el *while* y lo sobrescribimos para utilizar el *for*, que nos simplificará un poco el código al evitar fallos de bucles infinitos; además, hará más legible nuestro código fuente.

2.1.1. Objetos iterables

Un objeto iterable es un tipo de objeto que puede ser recorrido, es decir, que nos permite navegar por sus elementos uno a uno. Podemos utilizar la función “iter()” y pasar nuestro objeto iterable como parámetro; si es iterable, nos devolverá un iterador basado en el mismo objeto; en caso contrario, se producirá un error y se indicará que no es un objeto iterable.

```
numero = 7
print(iter(numero))
```

Salida:

```
TypeError          Traceback (most recent call last)
<ipython-input-9-4ad5ec321075> in <module>()
      1 numero = 7
----> 2 print(iter(numero))
      3

TypeError: 'int' object is not iterable
```

Tras utilizar la función "iter()", podremos recorrer todos los elementos con la función "next()". Por ejemplo:

```
numeros = [1,2,3,4,5,6]
numero = 7
it = iter(numeros)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

Salida:

```
1
2
3
4
5
6
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-8-86e7e806afe7> in <module>()
      8 print(next(it))
      9 print(next(it))
---> 10 print(next(it))
```

Deberemos tener cuidado, ya que solo podremos realizar llamadas a la función "next()" hasta que lleguemos al último elemento de la lista.

Con el bucle *for*, podemos recorrer los elementos de un diccionario, dado que se trata de un tipo de dato iterable. Como un diccionario está compuesto por una clave y un valor en cada elemento iterable del diccionario, tenemos varias formas de recorrerlos.

Para recorrer las claves del diccionario, utilizaremos directamente el diccionario en el *for* como si de una lista se tratara.

```
valores = {1: 'E', 2: 'I', 3: 'D', 4 : 'E', 5: 'N'}
for x in valores:
    print(x)
```

Salida:

```
1
2
3
4
5
```

Para recorrer los valores del diccionario, añadiremos ".values()" a continuación del nombre de nuestro diccionario, como si de una lista se tratara.

```
valores = {1: 'E', 2: 'I', 3: 'D', 4 : 'E', 5: 'N'}
for x in valores.values():
    print(x)
```

Salida:

```
E
I
D
E
N
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Por último, para recorrer tanto la clave como el valor de cada elemento del objeto iterable, utilizaremos ".items()"; y, en la parte de declaración del elemento, definiremos dos variables: *x* e *y*.

```
valores = {1: 'E', 2: 'D', 3: 'R', 4 : 'I', 5: 'C'}
for x,y in valores.items():
    print("Clave : ",x,"Valor : ",y)
```

Salida:

```
Clave :  1 Valor :  E
```

```
Clave :  2 Valor :  D
Clave :  3 Valor :  R
Clave :  4 Valor :  I
Clave :  5 Valor :  C
```

También podemos basar el bucle *for* en una secuencia numérica; para estos casos, Python nos propone la clase *range* (anteriormente, en Python 2, era una función). El constructor de esta clase "range(n)" nos devuelve un objeto iterable, y los valores están comprendidos entre 0 y $n-1$.

```
n=5
for i in range(n):
    print(i)
```

Salida:

```
0
1
2
3
4
```

La clase *range* nos permite añadir una variable adicional en el constructor para definir el inicio del objeto iterable "range(x,y)". Nos devolverá el objeto, y los valores estarán comprendidos entre x e $y-1$.

```
inicio = 2
n=5
for i in range(inicio,n):
    print(i)
```

Salida:

```
2
3
4
```

Esta clase también nos permite definir el incremento entre números del objeto devuelto por la función "range(min,max,suma)". Nos devolverá el objeto iterable, en el que los valores estarán comprendidos entre *min* y *max* − 1; y, adicionalmente, cada elemento aplicará una suma al anterior (*min*, *min* + suma, *min* + *suma* + *suma*...) hasta llegar a *max* − 1.

```
inicio = 2
n=10
suma=2
for i in range(inicio,n,suma):
    print(i)
```

Salida:

```
2
4
6
8
```

Adicionalmente, el bucle *for* también incorpora la funcionalidad con la palabra reservada *else*, igual que el bucle *while*.

El bloque *else* siempre será ejecutado una vez que el bucle *for* se haya terminado, siempre que no se haya interrumpido el *for* de cualquier otra forma (más adelante veremos otras maneras de finalizar o acelerar un bucle *for*).

```
inicio = 2
n=10
suma=2
for i in range(inicio,n,suma):
    print(i)
else:
    print("Se ha recorrido completamente")
```

Salida:

```
2
4
6
8
Se ha recorrido completamente
```

Siguiendo con los bucles *for*, podemos seleccionar solo aquellos elementos de una lista que cumplan una condición booleana ("True" o "False") añadiendo una condición mediante la palabra reservada *if*, y añadiremos corchetes ("[ ]") para encerrar dicho bucle.

Como ejemplo:

```
lista_original = range(10)
lista_modificada = [x for x in lista_original if x < 3]
print(lista_original)
print(lista_modificada)
```

Salida:

```
range(0, 10)
[0, 1, 2]
```

En el ejemplo anterior, hemos puesto delante de la palabra *for* la declaración de la variable *x*, que indica la modificación que hay que ejecutar por cada elemento de la lista. También podemos aplicar a los elementos de nuestra lista una función cuyo resultado será generar una lista; los elementos de esta serán el resultado de ejecutar la función en cada elemento de la lista inicial.

Por ejemplo, vamos a utilizar una función que sumará los parámetros de la lista con un número pasado por la función como parámetro:

```
def suma(a,b):
    return a+b

lista_original = range(10)
lista_modificada = [suma(2,x) for x in lista_original if x < 3]
print(lista_original)
print(lista_modificada)
```

Salida:

```
range(0, 10)
[2, 3, 4]
```

Y podemos transformar una lista de datos de un tipo en concreto en otra lista con cualquier otro tipo.

En este ejemplo, vamos a modificar los elementos de la lista original cambiando por el tipo booleano; si el número es par, pondrá “True”, y en caso contrario, “False”, indicando así que el número es impar.

```
def es_par(a):
    if a % 2 == 0:
        return True
```

```
    else:
        return False

lista_original = range(10)
lista_modificada = [es_par(x) for x in lista_original]
print(lista_original)
print(lista_modificada)
```

Salida:

```
range(0, 10)
[True, False, True, False, True, False, True, False, True, False]
```

Se pueden combinar cualesquiera de las funcionalidades anteriores comentadas para optimizar nuestros algoritmos.

2.1.2. Uso de *for* y de *while*

Habitualmente, utilizaremos bucles *for* cuando el número de iteraciones está predefinido; por ejemplo, por una lista o un diccionario. Los bucles *while* los utilizaremos en los casos en los que conozcamos una condición para finalizar dicha iteración; por ejemplo, $n < 3$.

Muchas veces se podrán utilizar los dos bucles, aunque en el *for* habitualmente se reducirán las líneas de código fuente.

2.2. Recursividad

Denominaremos **llamadas recursivas** a aquellas funciones que hacen referencia en su código a sí mismas.

Estas funciones suelen ser muy útiles en momentos concretos, pero es muy habitual dejar nuestro código en bucles infinitos, por lo que deberemos tomar las medidas adecuadas para que esto no ocurra, y solo las utilizaremos cuando sean estrictamente necesarias y no dispongamos de otra forma de resolver el problema.

Dentro de nuestro código, debemos planificar cuándo dejará de autollamarse para no caer en el bucle infinito.

Campus online

Accede al Campus online para ver el videotutorial *Rendimiento y optimización de bucles.*

```
def cuenta_atras(numero):
     numero -= 1
     if numero > 0:
          print(numero)
          cuenta_atras(numero)
     else:
          print("GO!")
     print("Fin de la función", numero)

cuenta_atras(5)
```

Salida:

```
4
3
2
1
GO!
Fin de la función 0
Fin de la función 1
Fin de la función 2
Fin de la función 3
Fin de la función 4
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

En este ejemplo podemos comprobar que, una vez que se ejecuta la función, se ejecuta primero el acceso a cada función recursiva, y el resto del código de la función finaliza antes de la última llamada; por ello, en la salida vemos que finaliza primero la ejecutada con 0 hasta la 4.

Habitualmente, las versiones recursivas de los algoritmos consumen más memoria, ya que la pila del estado de las funciones requiere almacenarse en ella. Por otra parte, suelen ser más legibles y elegantes.

Anteriormente hemos probado una función recursiva que no dispone de retorno; si añadimos retorno a nuestra función, podremos realizar cálculos relacionados con las iteraciones recursivas y modificar el valor devuelto en cada iteración.

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
```

```
    numero = fibonacci(n-1) + fibonacci(n-2)
    return numero

print(fibonacci(5))
```

Salida:

```
55
```

En este ejemplo, a la función le pasamos un número entero y ella nos devolverá el número que corresponde con el índice en la serie de Fibonacci.

Podemos observar que se realizan dos llamadas recursivas dentro de la misma función, para después sumar todos los retornos de todas las funciones, restando el parámetro *n* en cada una de las llamadas internas de dicha función en −1 y en −2, respectivamente.

Ejemplo recursivo frente a ejemplo iterativo:

Antes de definir nuestra función recursiva, debemos decidir y especificar el caso base y el caso recursivo.

Para realizar potencias de números, vamos a tomar como base la asignación de $n=0$ y devolveremos 1; y, por otro lado, vamos a definir un condicional mediante *if/else* para comprobar si el número es impar o par.

Añadimos a nuestro código un contador de llamadas a la función para poder visualizar cuántas veces se accede a ella, e igualamos a 1 la variable para contar con la llamada inicial de llamada a la función.

```
llamadas = 0
def potencia(b,n):
    global llamadas
    #Cualquier número elevado a 0
#tendrá como resultado 1.
    if n <= 0:
        return 1

    #Número par
    if n % 2 == 0:
        llamadas +=1
        pot = potencia(b, n/2)
        return pot * pot
```

```
    #Número impar
    else:
        llamadas +=1
        pot = potencia(b, (n-1)/2)
        return pot * pot * b

print(potencia(3,6))
print(llamadas)

potencia(3,6)
```

Salida:

```
729
4
```

Para poder replicar, necesitamos almacenar los resultados del cálculo si es par o impar en una lista.

```
ciclos = 0
def potencia(b,n):
    global ciclos
    lista = []
    while n > 0:
        ciclos+=1
        if n % 2 == 0:
            lista.append(True)
            n /= 2
        else:
            lista.append(False)
            n = (n-1)/2

    pot = 1
    while lista:
        ciclos+=1
        es_par = lista.pop()
        if es_par:
            pot = pot * pot
        else:
            pot = pot * pot * b

    return pot

print(potencia(3,6))
print(ciclos)
```

Salida:

```
729
6
```

En este ejemplo tenemos un bucle adicional para recorrer la lista y comprobar los números que son pares o impares. Realizamos un mayor número de ciclos en las iteraciones y consumimos mayores recursos, pero por otro lado podemos comprobar que el código es más complejo para evaluar su funcionalidad y facilitar su lectura.

Resumen

- Una iteración es una "vuelta" al bucle. Se pueden realizar tantas iteraciones como marquen las variables iterativas.

- Las iteraciones pueden responder a un número definido (iterativas) o a un número indefinido (recursividad) de "vueltas". Para este último caso hay que establecer siempre una condición de salida.

- Las iterativas se realizan mediante llamadas a los bucles *while* y *for*, estableciendo una condición por la que se ejecutarán un número definido de veces.

- A las recursivas se les tiene que especificar una condición de escape o *break*; cuando la alcancen, saldrán del bucle. No dependen de número de "vueltas" o veces que se ejecutan.

3. Instrucciones *break-continue*: usos y depuración de errores

Como hemos visto en puntos anteriores, utilizar bucles *for* y *while* nos permite automatizar y repetir fragmentos de código de forma eficiente. Sin embargo, muchas veces necesitaremos parar el bucle por completo o modificar su ejecución omitiendo parte del algoritmo antes de continuar con la siguiente iteración, o incluso ignorar dicha condición de paro del bucle.

Podremos realizar estas acciones con las palabras reservadas para este fin, llamadas instrucciones: *break*, *continue* y *pass*.

3.1. Instrucción *break*

Mediante esta instrucción, podremos finalizar en Python un bucle después de su ejecución sin finalizar la actual iteración del bucle, es decir, "salirnos" del bucle de manera forzada.

Esta instrucción se puede usar en bucles *while* y *for*, y habitualmente la utilizaremos después de una condición *if* para finalizar el bucle, siempre que hayamos alcanzado la condición "True".

```
for num in range(10):
    if(num == 3):
        break
    print("Mi número es "+ str(num))

print("Estamos fuera del bucle")
```

Salida:

```
Mi número es 0
Mi número es 1
Mi número es 2
Estamos fuera del bucle
```

En este ejemplo, en el cual estamos realizando un bucle *for* de una lista de 10 números, podemos observar que en el condicional "if(num ==

Certificación

Es importante interiorizar el concepto de uso y depuración de errores pues en numerosas preguntas de la certificación se hace referencia a este.

3):" estamos comprobando que la variable *num* tiene como valor un 3; por tanto, cuando la condición se valida como "True", se ejecuta la instrucción *break* y no se permite dar por finalizado el bucle hasta el final de la lista (por lo que tendríamos un retraso no deseado).

Habitualmente, utilizaremos la instrucción *break* porque hemos encontrado un elemento de la lista o porque no necesitamos ni queremos que finalice el bucle.

Esta instrucción, como hemos dicho, podemos utilizarla en un bucle *while* infinito.

```
num = 0
while(True):
    if(num == 3):
        break
    num+=1
    print("Mi número es "+ str(num))

print("Estamos fuera del bucle")
```

Salida:

```
Mi número es 1
Mi número es 2
Mi número es 3
Estamos fuera del bucle
```

En este ejemplo, en el cual tenemos un bucle "while(True)" que no debería terminar nunca (porque siempre se evalúa como "True"), hemos añadido una condición "if(num == 3)" que, de la misma manera que en el bucle *for*, cuando la condición sea validada como "True", se ejecutará la instrucción *break* por situarse dentro de esta y hará que las iteraciones de dicho bucle finalicen sin terminar por completo la iteración.

3.2. Instrucción *continue*

Mediante esta instrucción, en Python tendremos la opción de omitir y no ejecutar la parte de un bucle, pero continuar con las siguientes iteraciones de este. Es decir, interrumpiremos la iteración actual, pero el programa seguirá ejecutándose en la parte superior del bucle.

Esta condición se utilizará dentro del bucle y, habitualmente, después de una condición *if*.

Vamos a utilizar los mismos ejemplos presentados anteriormente con la instrucción *break* para ver la diferencia entre estas instrucciones.

```
for num in range(10):
    if(num==3):
        continue
    print("Mi número es "+ str(num))

print("Estamos fuera del bucle")
```

Salida:

```
Mi número es 0
Mi número es 1
Mi número es 2
Mi número es 4
Mi número es 5
Mi número es 6
Mi número es 7
Mi número es 8
Mi número es 9
Estamos fuera del bucle
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

A diferencia de la ejecución con la instrucción *break*, podemos comprobar que el bucle se ha ejecutado con todas las iteraciones correspondientes; pero cuando la variable *num* ha obtenido el valor *num* = 3, ha hecho que la instrucción de imprimir por pantalla dicho valor no se ejecute y pase a la siguiente iteración hasta finalizar el recorrido de la lista.

```
num = 0
while(True):
    if(num == 3):
        continue
    num+=1
    print("Mi número es "+ str(num))

print("Estamos fuera del bucle")
```

Salida:

```
Mi número es 1
Mi número es 2
Mi número es 3
```

En este ejemplo debemos ser cuidadosos, ya que, al tratarse de un bucle infinito con "while(True)", podemos dejar nuestro algoritmo ejecutando este bucle infinitamente sin ningún resultado favorable; por tanto, en este bucle no tendría demasiado sentido utilizar la instrucción *continue*.

Cuando la condición *if* es evaluada como "True", se ejecuta la instrucción *continue* y, por tanto, la variable *num* ya no incrementa su valor; al disponer de un bucle "while(True)", este bucle se quedará ejecutado eternamente, sin imprimir nada por pantalla.

Podemos utilizar la instrucción *continue* para evitar anidar validaciones condicionales o para optimizar un bucle eliminando casos que no queremos ejecutar. Es decir, dicha instrucción permitirá a nuestro programa omitir factores que surjan dentro de un bucle, pero continuando con las siguientes iteraciones.

3.3. Instrucción *pass*

Mediante esta instrucción, Python permite controlar la condición sin que el bucle se pueda ver afectado. Todo el código continuará ejecutándose a menos que utilicemos la instrucción *break* o finalice su ejecución; es decir, que dicha función no tendría ninguna repercusión.

Esta instrucción se utilizará dentro de los bucles *for* y *while*; habitualmente, dentro de una condición *if*.

La utilizaremos para reservar una parte del código que escribiremos más adelante o sustituir un código que ya no sea necesario; es decir, nos ayuda a desarrollar la estructura de nuestro código sin desarrollar el código interno de cada condicional.

Vamos a utilizar los mismos ejemplos anteriores, pero con la instrucción *pass*.

```
for num in range(10):
    if(num==3):
        pass
    print("Mi número es "+ str(num))
```

```
print(“Estamos fuera del bucle”)
```

Salida:

```
Mi número es 0
Mi número es 1
Mi número es 2
Mi número es 3
Mi número es 4
Mi número es 5
Mi número es 6
Mi número es 7
Mi número es 8
Mi número es 9
Estamos fuera del bucle
```

En este ejemplo se ejecuta el bucle completamente hasta finalizar todas las iteraciones de la lista creada con “range(10)”.

```
num = 0
while(True):
    if(num == 3):
        pass
    num+=1
    print(“Mi número es “+ str(num))

print(“Estamos fuera del bucle”)
```

Salida:

```
Mi número es 1
Mi número es 2
Mi número es 3
Mi número es 4
Mi número es 5
Mi número es 6
Mi número es 7
Mi número es ...
```

En este ejemplo, al ejecutarse por completo y no disponer de una finalización previa por tratarse de un bucle infinito, estaremos siempre mostrando por pantalla el valor incrementado de la variable *num*.

Observamos que la ejecución de dicha instrucción es utilizada para eliminar código de una función y no modificar todo el algoritmo, o simplemente para reservar el espacio para un futuro código.

3.4. Conclusión sobre *break*, *continue* y *pass*

Las instrucciones *break*, *continue* y *pass* nos permiten utilizar los bucles de manera más eficiente y tener mayor control sobre las condiciones de paro o la continuación de las iteraciones.

Vamos a poner un ejemplo para detectar cuándo obtenemos un número mayor que 3 en una lista.

```
for num in range(10):
    if(num<3):
        print("Soy menor que 3")
        continue
    if(num==3):
        print("Soy igual que 3")
        pass
    if(num==4):
        print("Soy mayor que 3")
        break
    print("Mi número es "+ str(num))

print("Estamos fuera del bucle")
```

Salida:

```
Soy menor que 3
Soy menor que 3
Soy menor que 3
Soy igual que 3
Mi número es 3
Soy mayor que 3
Estamos fuera del bucle
```

En este ejemplo podemos comprobar que la ejecución de imprimir por pantalla "Mi número es..." solo se ha realizado una vez tras la instrucción *pass*. Por tanto, hemos reducido el tiempo de ejecución en nuestro código para realizar esta búsqueda del número 3.

Por ejemplo, si no utilizamos las instrucciones *continue*, *pass* y *break*, nuestro algoritmo debería ser así:

```
for num in range(10):
    if(num<3):
        print("Soy menor que 3")

    elif(num==3):
        print("Soy igual que 3")

    elif(num==4):
        print("Soy mayor que 3")

    print("Mi número es "+ str(num))

print("Estamos fuera del bucle")
```

Salida:

```
Soy menor que 3
Mi número es 0
Soy menor que 3
Mi número es 1
Soy menor que 3
Mi número es 2
Soy igual que 3
Mi número es 3
Soy mayor que 3
Mi número es 4
Mi número es 5
Mi número es 6
Mi número es 7
Mi número es 8
Mi número es 9
Estamos fuera del bucle
```

En este ejemplo obtenemos algo muy parecido, aunque ahora no eliminamos la ejecución hasta el final de todas las iteraciones. Por ello, se imprime todas las veces "Mi número es..." y obtenemos un menor rendimiento en nuestro algoritmo, cuyo fin es detectar cuándo el número es mayor que 3, ya que, aunque lo hemos detectado, seguimos con el bucle hasta finalizar la lista.

Para finalizar este bloque, vamos a programar un juego en el que existen dos personajes, un monstruo y un protagonista, los cuales tendrán su correspondiente vida; mediante la función *random*, simularemos una tirada de dados aleatoria. Si el "dado" saca un 0, le restaremos vida al monstruo; en cambio, si sale un 1, el protagonista perderá vida. El resultado será aleatorio.

```
for random import random as TirarDados
Monstruo = 10
Protagonista = 10
golpe = 5

while True:
  if Protagonista == 0:
    print(“Hemos perdido :(“)
    break
  elif Monstruo == 0:
    print(“Hemos ganado :)”)
    break
  else:
    pass
  dados=round(TirarDados())
  if ( dados == 0 ):
    Monstruo -= golpe
    print(“¡Monstruo golpeado! Le queda: “, Monstruo, “de vida.”)
  elif ( dados == 1):
   Protagonista -= golpe
   print(“¡Protagonista golpeado! Tenemos: “, Protagonista,
“de vida.”)
  else:
   print(“Error en los dados”)
```

Salida:

```
¡Protagonista golpeado! Tenemos: 5 de vida.
¡Monstruo golpeado! Le queda:  5 de vida.
¡Monstruo golpeado! Le queda:  0 de vida.
Hemos ganado :)
```

Resumen

- *Break* finaliza el bucle tras la ejecución de una instrucción.
- *Continue* interrumpe el código de la actual iteración y continúa el bucle con la siguiente.
- *Pass* no realiza ninguna acción; se utiliza para reservar espacio de código que se implementará más adelante.

4. Instrucciones *raise* y *try-except-finally*: usos y depuración de errores

A los errores de ejecución los llamaremos **excepciones**, que evaluaremos mediante las instrucciones *raise* y *try-except*.

Una vez que tengamos el código fuente sintácticamente correcto, se puede generar un error tras ejecutarlo. Los errores detectados u ocasionados en la ejecución se denominan *excepciones*. Vamos a aprender a gestionarlas en Python, puesto que, si no se controlan o gestionan, aparecerán como un mensaje en la consola y se detendrá la ejecución de nuestro código.

Si se provoca una excepción durante la ejecución de un programa y la función no la maneja, dicha excepción se propaga hacia la función que la invocó, y seguirá propagándose hasta llegar a la función inicial; y si tampoco aquí es controlada, se parará la ejecución. Por ello vamos a aprender a manejar las excepciones.

Certificación

Es importante interiorizar las excepciones pues en numerosas preguntas de la certificación se hace referencia a estas.

4.1. Control o manejo de excepciones

Para el control de excepciones vamos a utilizar las palabras reservadas para este fin en Python, que son *try*, *except* y *finally*. Nos van a permitir manejar las excepciones y evitar que el programa sea interrumpido de forma inesperada.

Dentro del bloque *try* vamos a colocar todo el código fuente que pueda llegar a producir una excepción.

A continuación, se utilizará la instrucción *except* para capturar dicha excepción y procesarla, relanzando el código por otro lado o simplemente mostrando al usuario un mensaje por pantalla. Por ejemplo, para capturar un error de una división entre 0:

Para saber más

Podemos afirmar que la detección y el manejo de las excepciones son las grandes diferencias entre un lenguaje moderno y uno antiguo.

Sin capturar dicha excepción:

```
print(5/0)
```

Salida:

```
ZeroDivisionError
Traceback (most recent call last)
<ipython-input-21-fad870a50e27> in <module>()
----> 1 print(5/0)
ZeroDivisionError: division by zero
```

Capturando la excepción y mostrando un error al usuario:

```
try:
  print(15/0)
except:
  print("No se permite la división por 0.")
```

Salida:

```
No se permite la división por 0.
```

Por otro lado, como dentro de un bloque *try* se pueden producir diversos errores, el bloque *except* nos permite seleccionar dichas excepciones de forma concreta especificando cuál o cuáles queremos capturar, separadas por coma al lado de la palabra *except*; incluso, si queremos ejecutar un código distinto para cada excepción, podemos crear varios bloques *except* de manera correlativa y solo uno de ellos se ejecutará en caso de excepción. Si no conocemos las posibles excepciones, dejaremos la palabra *except*: se capturarán todas las excepciones provocadas dentro del bloque *try*.

La estructura será esta:

```
try:
    # Código fuente
    print(5/0)
except IOError:
    # Se ejecuta si aparece una excepción IOError.
    print(1)
except ZeroDivisionError:
    # Se ejecuta si aparece una excepción ZeroDivisionError.
    print(2)
except:
    # Se ejecuta con cualquier excepción no gestionada previamente.
    print(3)
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
2
```

Finalmente, también se puede incorporar un bloque llamado *finally*, donde ubicaremos el código que queramos que sea ejecutado siempre, como acciones de limpieza, cierre de base de datos o cierre de ficheros.

Este bloque se ejecutará siempre, tanto si el código ha producido una excepción como si no. No es imprescindible su utilización, y también se puede usar solo con el bloque *try* (sin el bloque *except*).

En conclusión, Python está ejecutando el código fuente que se encuentra dentro del bloque *try*; si se produce una excepción, finalizará la ejecución de dicho código en el punto exacto donde surgió la excepción y pasará a ejecutar el bloque *except* correspondiente. Si Python detecta alguno cuyo tipo sea igual a la excepción, ejecutará el código de este bloque; en cambio, si no lo encuentra pero existe uno sin tipo predefinido, este bloque será ejecutado. Entonces, pasará a ejecutar el bloque finally siempre que este bloque esté definido.

Por otro lado, si no se produce ninguna excepción dentro del bloque *try*, pasará directamente a la ejecución del bloque *finally* siempre que dicho bloque este definido.

Vamos a ampliar el código que captura la excepción de división por 0 solicitando los parámetros por pantalla y capturando el error en el valor de los datos por tipo inesperado.

En este ejemplo vamos a añadir el bloque *finally* para comprobar que siempre se ejecuta después del bloque *try*, se haya producido una excepción o no, y finalizaremos con un *break* después de calcular el resultado de la división.

```
while(True):
  try:
    a = int(input("Introduce el numerador : "))
    b = int(input("Introduce el denominador : "))
    resultado = a/b
  except ZeroDivisionError:
    print("No se permite la división por 0")
  except ValueError:
    print("Solo se permiten números de tipo entero int()")
  except:
```

```
    print("Error desconocido")
    break
  else:
    print("El resultado es : ",resultado)
    break
  finally:
    print("Código Final")
```

Salida:

```
Introduce el numerador : 2
Introduce el denominador : 0
No se permite la división por 0
Código Final
Introduce el numerador : w
Solo se permiten números de tipo entero int()
Código Final
Introduce el numerador : 2.3
Solo se permiten números de tipo entero int()
Código Final
Introduce el numerador : 2
Introduce el denominador : 3
El resultado es :  0.6666666666666666
Código Final
```

En nuestros códigos utilizaremos habitualmente el bloque *finally* para liberar recursos; por ejemplo, para cerrar archivos o conexiones a base de datos sin tener en cuenta si el código lanzó una excepción, o evitar dejar un archivo abierto para siempre.

4.2. Propagar la excepción

Anteriormente hemos aprendido a capturar excepciones, y ahora vamos a ver qué es posible hacer con ellas. Podemos dejar registrada la excepción, propagarla e incluso realizar las dos cosas.

Para registrar una excepción, podemos escribir un archivo de *log* en nuestro algoritmo o simplemente mostrarlo por consola/pantalla con la función "print()". Para registrar la excepción, intentaremos almacenar siempre información necesaria, como la del contexto en que ocurrió la excepción; el tipo de excepción ocurrida, el momento en que ocurrió y cuáles fueron las llamadas ejecutadas antes de producirse la excepción. Esta información la utilizaremos a posteriori para mejorar nuestros programas o algoritmos y facilitarnos un diagnóstico de estos.

Por otra parte, podemos necesitar propagar la excepción a otras clases o funciones que han invocado la función actual, o propagar una excepción diferente, que dé más o menos información al usuario. Para ello utilizaremos la instrucción *raise*.

Si utilizamos la instrucción dentro de un bloque *except* sin pasarle parámetros, Python replicará la misma excepción antes producida.

También podremos necesitar propagar una excepción distinta de la que se ha producido, que sea más significativa para el usuario final. Para ello utilizaremos la instrucción *raise* y le indicaremos el tipo de excepción que tiene que lanzar pasándole los parámetros que queramos añadir como información adicional.

```
raise ZeroDivisionError("El divisor no puede ser cero")
```

Utilizaremos un único argumento con la instrucción *raise* para indicar la excepción que se generará; podrá ser una instancia de excepción o una clase que herede de *exception*. En este último caso, esta será instanciada llamando a su constructor sin argumentos.

```
try:
  try:
      print(15/0)
  except:
      raise ZeroDivisionError("El divisor no puede ser cero")
except:
  raise NameError('Prueba')
```

Salida:

```
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-14-7a045f1249fe> in <module>()
      2   try:
----> 3       print(15/0)
      4   except:

ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-14-7a045f1249fe> in <module>()
```

```
      4   except:
----> 5       raise ZeroDivisionError("El divisor no puede ser cero")
      6 except:

ZeroDivisionError: El divisor no puede ser cero

During handling of the above exception, another exception occurred:

NameError                                 Traceback (most recent call last)
<ipython-input-14-7a045f1249fe> in <module>()
      5       raise ZeroDivisionError("El divisor no puede ser cero")
      6 except:
----> 7   raise NameError('Prueba')

NameError: Prueba
```

En este ejemplo podemos comprobar que tenemos toda la información de todas las excepciones producidas. Puede darse el caso de que queramos abstraer al usuario de dichas excepciones; la instrucción *raise* nos permite añadir "from None" al final de la invocación de la instrucción para mostrar solo la última o últimas, dependiendo de si lo añadimos en el primer *except* o en el segundo del ejemplo anterior.

```
try:
  try:
      print(15/0)
  except:
      raise ZeroDivisionError("El divisor no puede ser cero")
except:
  raise NameError('Prueba') from None
```

Salida:

```
NameError                 Traceback (most recent call last)
<ipython-input-15-8a76edd84df1> in <module>()
      5       raise ZeroDivisionError("El divisor no puede ser cero")
      6 except:
----> 7   raise NameError('Prueba') from None

NameError: Prueba
```

4.3. Información de la excepción (tipo,valor, etc.)

Podemos tener la necesidad de acceder a la información de contexto dentro de un bloque *except*. Para esto tendremos dos alternativas:

- Utilizar la función *exc_info* del módulo “sys”; nos devolverá una tupla de información sobre la última excepción producida en un bloque *except*, y que contendrá el tipo de dicha excepción, su valor y las llamadas realizadas.

- Pasar desde la misma sentencia *except* un identificador para que nos guarde una referencia de la excepción producida, como en el siguiente ejemplo.

```
try:
      print(15/0)
except Exception as e:
      print(e)
```

Salida:

```
division by zero
```

Hay que tener en cuenta que, en otros lenguajes como Java o C#, si en cualquier situación se produce una excepción, esta debe formar parte siempre de la declaración de la función y estar capturada dentro de un bloque *try*.

En Python, al no tener esta restricción, debemos procurar capturar todas las excepciones posibles o utilizar *except* al final; si no, los programas terminarán de forma inesperada.

4.4. Excepciones definidas por el programador

Podemos programar nuestras propias excepciones creando una nueva clase.

- Deberán invocar la clase exception de forma directa o indirecta.

- Serán definidas de igual forma que cualquier otra clase.

- Deberemos mantenerlas lo más simples posible.

- Es importante incorporar un número de atributos con información relevante sobre el error de la excepción para facilitar la comprensión.

Cuando escribamos un programa que puede arrojar varios errores distintos, crearemos una clase base para excepciones definidas en este y la extenderemos para crear clases de excepciones específicas para los distintos errores.

```
class Error(Exception):
  pass

class InputError(Error):

    def __init__(self, expression, message):
      self.expression = expression
      self.message = message

class TransitionError(Error):

    def __init__(self,previous,next,message):
      self.previous = previous
      self.next = next
      self.message = message
```

Muchos módulos estándar definen sus excepciones para reportar excepciones producidas en funciones internas del mismo módulo y ofrecerle al usuario información más concreta sobre el error real.

Sin embargo, debemos tener en cuenta que una clase en un bloque *except* es compatible con una excepción si es de la misma clase o de una clase derivada de esta, pero no a la inversa. Por ejemplo:

```
class X(Exception):
    pass

class Y(X):
    pass

class Z(Y):
    pass

for f in [X, Y, Z]:
    try:
        raise f()
    except Z:
        print("Z")
    except Y:
        print("Y")
    except X:
        print("X")
```

```
class X(Exception):
    pass

class Y(X):
    pass

class Z(Y):
    pass

for f in [X, Y, Z]:
    try:
        raise f()
    except Z:
        print("X")
    except Y:
        print("Y")
    except X:
        print("Z")
```

Salida:

```
X
Y
Z
```

```
X
X
X
```

Podemos comprobar en el anterior ejemplo que la clase base siempre es detectada primero por el bloque *except* y, por tanto, el orden del *except* es importante, ya que en el caso de la derecha perderíamos la ejecución del *except Y* y *Z*.

Después de este punto, se nos puede generar una duda: ¿Debemos utilizar un *try-except* para detectar una excepción o será mejor un *if* para evitarla?

Respecto al *hardware* donde vamos a interpretar nuestro código fuente, habrá un conjunto de instrucciones que se ejecutarán; por ello, para un *if* implicará pocas instrucciones al traducir al lenguaje máquina. Sin embargo, para lanzar una excepción son necesarios muchos más pasos, como detener el hilo de la ejecución y reservar memoria para almacenar información, además del conjunto de instrucciones.

Por tanto, cuando se produce una excepción, un *if* implica pocas instrucciones y poco tiempo de ejecución, al contrario que *try-except*.

Teniendo esto en cuenta, utilizaremos *try-except* para controlar nuestras excepciones, pero intentaremos evitarlas con comprobaciones *if* para reducir el tiempo de ejecución y que nuestro código sea más eficiente.

Por ejemplo:

```
a = 2
b = 0
#parte 1
try:
      print(a/b)
except:
      print("Division por 0")

#parte 2
if(b != 0):
    print(a/b)
else:
    print("Division por 0")
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
Division por 0
Division por 0
```

El resultado es el mismo, pero en el *if* evitamos la excepción, que *a priori* utilizará más recursos máquina.

4.5. Algoritmos de búsqueda y ordenación

Una vez aprendidos los bucles, las condiciones y las excepciones, es importante conocer la forma de recorrer o buscar elementos de una forma eficiente dentro de una lista o un diccionario; por tanto, vamos a aprender algoritmos de búsqueda lineal o secuencial y binaria.

En el algoritmo de búsqueda lineal, se comienza a buscar el elemento desde el principio y se va desplazando hasta el final hasta que se encuentre el elemento buscado. Detiene la ejecución del programa cuando dicho elemento es encontrado. Siendo así, en los casos en los que el elemento se encuentre al final o no se encuentre, el algoritmo tendrá que recorrer todos los elementos.

No es necesario que la lista o el diccionario estén ordenados.

```
def buscar_elemento(lista, n, elemento):

  for i in range(n):
    if lista[i] == elemento:
      return True
  return False


lista = [1, 2, 8, 4, 5, 6, 7, 3, 9, 10]
n = 10
elemento = 6

if buscar_elemento(lista, n, elemento):
  print("Hemos encontrado ",elemento)
else:
  print("El elemento ",elemento," no existe en la lista")
```

Salida:

```
Hemos encontrado  6
```

Por otro lado, en el algoritmo de búsqueda binaria siempre se comprueba el elemento situado en el centro de la lista, y es condición indispensable que la lista esté ordenada para realizar esta búsqueda.

En este algoritmo se itera desde el centro y se va descartando la parte izquierda o derecha de la lista, según si el elemento buscado es mayor o menor, hasta localizar dicho elemento.

```
def buscar_elemento(lista, n, elemento):
  i = 0
  inicio = 0
  fin = n - 1
  while i < n:
    mitad = (inicio + fin) // 2
    if lista[mitad] == elemento:
      return True
    elif lista[mitad] < elemento:
      inicio = mitad + 1
    else:
      fin = mitad - 1
    i += 1
  return False

lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n = 10
elemento = 6

if buscar_elemento(lista, n, elemento):
  print("Hemos encontrado ",elemento)
else:
  print("El elemento ",elemento," no existe en la lista")
```

Salida:

```
Hemos encontrado 6
```

Como conclusión, el algoritmo de búsqueda binaria es mucho más eficiente que el lineal, pero tiene el inconveniente de que la lista o matriz debe estar ordenada previamente; por ello, vamos a aprender un algoritmo de ordenación por selección, aunque en Python dispongamos de la función "sorted" para ordenar listas.

Este algoritmo de ordenación se basa en seleccionar el primer elemento e ir verificando cuál es el mínimo valor comparándolo con el resto de la lista, actualizando su posición sucesivamente hasta finalizar todo el recorrido.

Por tanto, tendremos dos bucles:

- Uno para el recorrido de los elementos.
- Otro para realizar un recorrido por cada uno para poder comparar los elementos entre sí y realizar los cambios de posición oportunos.

```
def seleccion(lista):
    for i in range(len(lista)):
        actual = i
        for k in range(i+1, len(lista)):
            if lista[k] < lista[actual]:
                actual = k
        cambio(lista, actual, i)

def cambio(A, x, y):
    temp = A[x]
    A[x] = A[y]
    A[y] = temp

lista = [3.7,2.5,8.2,3.2,3.8,4.9,2.1,1.8]
seleccion(lista)
print(lista)
```

Salida:

```
[1.8, 2.1, 2.5, 3.2, 3.7, 3.8, 4.9, 8.2]
```

4.6. Depuración de errores

Cuando nuestros códigos sean más complejos no será tan sencillo encontrar el origen de nuestros errores; para ello podremos utilizar un depurador. Es una herramienta que nos permite trazar y encontrar nuestros errores. El depurador por defecto de Python, pdb, realiza un gran trabajo para ayudarnos a encontrar nuestros errores.

Primero importaremos el depurador en nuestro código con “import pdb”, y, para añadir puntos de control fácilmente, teclearemos “pdb.set_trace()”, que hará que se detenga el programa, se nos muestre por

pantalla el *prompt* del depurador y podamos llevar a cabo estas acciones:

- **Step (s)**

  Ejecuta la línea actual incluso entrando en una función, parando en la siguiente línea del código a la espera de nuevas instrucciones.

- **Next (n)**

  Continúa la ejecución hasta la siguiente línea o hasta que encuentre un *return*, pero no entraremos dentro de las funciones.

- **Continue (c)**

  Continúa con la ejecución y solo para si encuentra otro punto de interrupción.

- **Return (r)**

  Continúa hasta encontrar un *return*.

- **List (l)**

  Muestra 11 líneas de código desde la línea actual. Si añadimos argumentos, nos muestra las líneas indicadas en estos.

- **Quit (q)**

  Sale del depurador, finalizando la ejecución del programa.

Ejercicios resueltos

Practica **los ejercicios de autocomprobación** en Colab con los conocimientos adquiridos en este módulo.

Resumen

- Los errores detectados u ocasionados en la ejecución se denominan excepciones.
- En el bloque *try* estará todo el código susceptible de contener errores.
- En el bloque *except* realizaremos todas las acciones requeridas tras la detección de una excepción.
- En el bloque *finally* realizaremos las tareas de limpieza o cierre deseadas.
- Utilizaremos la instrucción *raise* para propagar excepciones definidas o no por el usuario.

Índice