Capítulo 4

Estructuras de control

> *—De ahí que estén dando vueltas continuamente, supongo —dijo Alicia.*
> *—Si, así es —dijo el Sombrerero—, conforme se van ensuciando las cosas.*
> *—Pero ¿qué ocurre cuando vuelven al principio de nuevo? —se atrevió a preguntar Alicia.*
>
> LEWIS CARROLL, *Alicia a través del espejo*.

Los programas que hemos aprendido a construir hasta el momento presentan siempre una misma secuencia de acciones:

1. Se piden datos al usuario (asignando a variables valores obtenidos con *raw_input*).
2. Se efectúan cálculos con los datos introducidos por el usuario, guardando el resultado en variables (mediante asignaciones).
3. Se muestran por pantalla los resultados almacenados en variables (mediante la sentencia **print**).

Estos programas se forman como una serie de líneas que se ejecutan una tras otra, desde la primera hasta la última y siguiendo el mismo orden con el que aparecen en el fichero: el *flujo de ejecución* del programa es estrictamente secuencial.

No obstante, es posible alterar el flujo de ejecución de los programas para hacer que:

- tomen decisiones a partir de los datos y/o resultados intermedios y, en función de éstas, ejecuten ciertas sentencias y otras no;
- tomen decisiones a partir de los datos y/o resultados intermedios y, en función de éstas, ejecuten ciertas sentencias más de una vez.

El primer tipo de alteración del flujo de control se efectúa con *sentencias condicionales* o *de selección* y el segundo tipo con *sentencias iterativas* o *de repetición*. Las sentencias que permiten alterar el flujo de ejecución se engloban en las denominadas *estructuras de control de flujo* (que abreviamos con el término «estructuras de control»).

Estudiaremos una forma adicional de alterar el flujo de control que permite señalar, detectar y tratar los errores que se producen al ejecutar un programa: las sentencias de emisión y captura de excepciones.

4.1. Sentencias condicionales

4.1.1. Un programa de ejemplo: resolución de ecuaciones de primer grado

Veamos un ejemplo. Diseñemos un programa para resolver cualquier ecuación de primer grado de la forma

$$ax + b = 0,$$

donde x es la incógnita.

Antes de empezar hemos de responder a dos preguntas:

1. ¿Cuáles son los datos del problema? (Generalmente, los datos del problema se pedirán al usuario con *raw_input*.)

   En nuestro problema, los coeficientes a y b son los datos del problema.

2. ¿Qué deseamos calcular? (Típicamente, lo que calculemos se mostrará al usuario mediante una sentencia **print**.)

   Obviamente, el valor de x.

Ahora que conocemos los *datos de entrada* y el resultado que hemos de calcular, es decir, los *datos de salida*, nos preguntamos: ¿cómo calculamos la salida a partir de la entrada? En nuestro ejemplo, despejando x de la ecuación llegamos a la conclusión de que x se obtiene calculando $-b/a$.

Siguiendo el esquema de los programas que sabemos hacer, procederemos así:

1. Pediremos el valor de a y el valor de b (que supondremos de tipo flotante).
2. Calcularemos el valor de x como $-b/a$.
3. Mostraremos por pantalla el valor de x.

Escribamos el siguiente programa en un fichero de texto llamado `primer_grado.py`:

primer_grado_7.py — primer_grado.py

```
a = float(raw_input('Valor de a: '))
b = float(raw_input('Valor de b: '))

x = -b / a

print 'Solución: ', x
```

Las líneas se ejecutan en el mismo orden con el que aparecen en el programa. Veámoslo funcionar:

```
Valor de a: 10
Valor de b: 2
Solución: -0.2
```

EJERCICIOS

▶ 53 Un programador propone el siguiente programa para resolver la ecuación de primer grado:

```
a = float(raw_input('Valor de a: '))
b = float(raw_input('Valor de b: '))

a * x + b = 0

print 'Solución: ', x
```

¿Es correcto este programa? Si no, explica qué está mal.

▶ 54 Otro programador propone este programa:

```
1 x = -b / a
2
3 a = float(raw_input('Valor de a: '))
4 b = float(raw_input('Valor de b: '))
5
6 print 'Solución: ', x
```

¿Es correcto? Si no lo es, explica qué está mal.

Nuestro programa presenta un punto débil: cuando *a* vale 0, se produce un error de división por cero:

```
Valor de a: 0
Valor de b: 3
Traceback (innermost last):
  File 'primer_grado.py', line 3, in ?
    x = -b / a
ZeroDivisionError: float division
```

En la medida de lo posible hemos de tratar de evitar los errores en tiempo de ejecución: detienen la ejecución del programa y muestran mensajes de error poco comprensibles para el usuario del programa. Si al escribir el programa hemos previsto una solución para todo posible error de ejecución, podemos (y debemos) tomar el control de la situación en todo momento.

> ***Errores de ejecución***
>
> Hemos dicho que conviene evitar los errores de programa que se producen en tiempo de ejecución y, ciertamente, la industria de desarrollo de software realiza un gran esfuerzo para que sus productos estén libres de errores de ejecución. No obstante, el gran tamaño de los programas y su complejidad (unidos a las prisas por sacar los productos al mercado) hacen que muchos de estos errores acaben haciendo acto de presencia. Todos hemos sufrido la experiencia de, ejecutando una aplicación, obtener un mensaje de error indicando que se ha abortado la ejecución del programa o, peor aún, el computador se ha quedado «colgado». Si la aplicación contenía datos de trabajo importantes y no los habíamos guardado en disco, éstos se habrán perdido irremisiblemente. Nada hay más irritante para el usuario que una aplicación poco estable, es decir, propensa a la comisión de errores en tiempo de ejecución.
>
> El sistema operativo es, también, software, y está sujeto a los mismos problemas de desarrollo de software que las aplicaciones convencionales. Sin embargo, los errores en el sistema operativo son, por regla general, más graves, pues suelen ser éstos los que dejan «colgado» al ordenador.
>
> El famoso «sal y vuelve a entrar en la aplicación» o «reinicia el computador» que suele proponerse como solución práctica a muchos problemas de estos es consecuencia de los bajos niveles de calidad de buena parte del software que se comercializa.

4.1.2. La sentencia condicional if

En nuestro programa de ejemplo nos gustaría *detectar* si *a* vale cero para, *en ese caso*, no ejecutar el cálculo de la cuarta línea de `primer_grado.py`, que es la que provoca el error. ¿Cómo hacer que cierta parte del programa se ejecute o deje de hacerlo en función de una condición determinada?

Los lenguajes de programación convencionales presentan una sentencia especial cuyo significado es:

«Al llegar a este punto, ejecuta esta(s) acción(es) *sólo si* esta condición *es cierta*.»

Este tipo de sentencia se denomina *condicional* o *de selección* y en Python es de la siguiente forma:

```
if condición:
    acción
    acción
    ...
    acción
```

(En inglés «if» significa «*si*».)

En nuestro caso, deseamos detectar la condición «a no vale 0» y, sólo en ese caso, ejecutar las últimas líneas del programa:

primer_grado_8.py — primer_grado.py

```
1 a = float(raw_input('Valor de a: '))
2 b = float(raw_input('Valor de b: '))
3
4 if a != 0:
5     x = -b/a
6     print 'Solución: ', x
```

Analicemos detenidamente las líneas 4, 5 y 6. En la línea 4 aparece la sentencia condicional **if** seguida de lo que, según hemos dicho, debe ser una condición. La condición se lee fácilmente si sabemos que **!=** significa «es distinto de». Así pues, la línea 4 se lee «si a es distinto de 0». La línea que empieza con **if** debe finalizar obligatoriamente con dos puntos (**:**). Fíjate en que las dos siguientes líneas se escriben más a la derecha. Para destacar esta característica, hemos dibujados dos líneas verticales que marcan el nivel al que apareció el **if**. Decimos que esta línea presentan mayor *indentación* o *sangrado* que la línea que empieza con **if**. Esta mayor indentación indica que la ejecución de estas dos líneas depende de que se satisfaga la condición a **!=** 0: sólo cuando ésta es cierta se ejecutan las líneas de mayor sangrado. Así pues, cuando a valga 0, esas líneas *no se ejecutarán*, evitando de este modo el error de división por cero.

Veamos qué ocurre ahora si volvemos a introducir los datos que antes provocaron el error:

```
Valor de a: 0
Valor de b: 3
```

Mmmm... no ocurre nada. No se produce un error, es cierto, pero el programa acaba sin proporcionar ninguna información. Analicemos la causa. Las dos primeras líneas del programa se han ejecutado (nos piden los valores de a y b); la tercera está en blanco; la cuarta línea también se ha ejecutado, pero dado que la condición no se ha cumplido (a vale 0), las líneas 5 y 6 se han ignorado y como no hay más líneas en el programa, la ejecución ha finalizado sin más. No se ha producido un error, ciertamente, pero acabar así la ejecución del programa puede resultar un tanto confuso para el usuario.

Veamos qué hace este otro programa:

primer_grado_9.py — primer_grado.py

```
1 a = float(raw_input('Valor de a: '))
2 b = float(raw_input('Valor de b: '))
3
4 if a != 0:
5     x = -b/a
6     print 'Solución: ', x
7 if a == 0:
8     print 'La ecuación no tiene solución.'
```

Las líneas 7 y 8 empiezan, nuevamente, con una sentencia condicional. En lugar de `!=`, el operador de comparación utilizado es `==`. La sentencia se lee «si a es igual a 0».

EJERCICIOS

▶ 55 Un estudiante ha tecleado el último programa y, al ejecutarlo, obtiene este mensaje de error.

```
  File "primer_grado4.py", line 7
    if a = 0:
         ^
SyntaxError: invalid syntax
```

Aquí tienes el contenido del fichero que él ha escrito:

primer_grado_10.py — primer_grado.py

```
1 a = float(raw_input('Valor de a: '))
2 b = float(raw_input('Valor de b: '))
3
4 if a != 0:
5     x = -b/a
6     print 'Solución: ', x
7 if a = 0:
8     print 'La ecuación no tiene solución.'
```

Por más que el estudiante lee el programa, no encuentra fallo alguno. Él dice que la línea 7, que es la marcada como errónea, se lee así: «si a es igual a cero...» ¿Está en lo cierto? ¿Por qué se detecta un error?

Ejecutando el programa con los mismos datos, tenemos ahora:

```
Valor de a: 0
Valor de b: 3
La ecuación no tiene solución.
```

Pero, ante datos tales que a es distinto de 0, el programa resuelve la ecuación:

```
Valor de a: 1
Valor de b: -1
Solución: 1
```

Estudiemos con detenimiento qué ha pasado en cada uno de los casos:

$a = 0$ y $b = 3$	$a = 1$ y $b = -1$
Las líneas 1 y 2 se ejecutan, con lo que se leen los valores de a y b.	Las líneas 1 y 2 se ejecutan, con lo que se leen los valores de a y b.
La línea 4 se ejecuta y el resultado de la comparación es *falso*.	La línea 4 se ejecuta y el resultado de la comparación es *cierto*.
Las líneas 5 y 6 se ignoran.	Se ejecutan las líneas 5 y 6, con lo que se muestra por pantalla el valor de la solución de la ecuación: `Solución: 1`.
La línea 7 se ejecuta y el resultado de la comparación es *cierto*.	La línea 7 se ejecuta y el resultado de la comparación es *falso*.
La línea 8 se ejecuta y se muestra por pantalla el mensaje «`La ecuación no tiene solución.`»	La línea 8 se ignora.

Este tipo de análisis, en el que seguimos el curso del programa línea a línea para una configuración dada de los datos de entrada, recibe el nombre de *traza* de ejecución. Las *trazas* de ejecución son de gran ayuda para comprender qué hace un programa y localizar así posibles errores.

EJERCICIOS

▶ 56 Un programador primerizo cree que la línea 7 de la última versión de `primer_grado.py` es innecesaria, así que propone esta otra versión como solución válida:

primer_grado_11.py — primer_grado.py

```
1 a = float(raw_input('Valor de a: '))
2 b = float(raw_input('Valor de b: '))
3
4 if a != 0:
5     x = -b/a
6     print 'Solución: ', x
7
8 print 'La ecuación no tiene solución.'
```

Haz una traza del programa para $a = 2$ y $b = 2$. ¿Son correctos todos los mensajes que muestra por pantalla el programa?