Chapter 3

Ejecución condicional

3.1 Expresiones booleanas

Una *expresión booleana* es aquella que puede ser verdadera (`True`) o falsa (`False`). Los ejemplos siguientes usan el operador `==`, que compara dos operandos y devuelve `True` si son iguales y `False` en caso contrario:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` y `False` son valores especiales que pertenecen al tipo `bool (booleano)`; no son cadenas:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

El operador `==` es uno de los *operadores de comparación*; los demás son:

```
x != y               # x es distinto de y
x > y                # x es mayor que y
x < y                # x es menor que y
x >= y               # x es mayor o igual que y
x <= y               # x es menor o igual que y
x is y               # x es lo mismo que y
x is not y           # x no es lo mismo que y
```

A pesar de que estas operaciones probablemente te resulten familiares, los símbolos en Python son diferentes de los símbolos matemáticos que se usan para realizar las mismas operaciones. Un error muy común es usar sólo un símbolo igual (`=`) en vez del símbolo de doble igualdad (`==`). Recuerda que `=` es un operador de asignación, y `==` es un operador de comparación. No existe algo como `=<` o `=>`.

3.2 Operadores lógicos

Existen tres *operadores lógicos*: `and (y)`, `or (o)`, y `not (no)`. El significado semántico de estas operaciones es similar a su significado en inglés. Por ejemplo,

`x > 0 and x < 10`

es verdadero sólo cuando `x` es mayor que 0 *y* menor que 10.

`n%2 == 0 or n%3 == 0` es verdadero si *cualquiera* de las condiciones es verdadera, es decir, si el número es divisible por 2 *o* por 3.

Finalmente, el operador `not` niega una expresión booleana, de modo que `not (x > y)` es verdadero si `x > y` es falso; es decir, si `x` es menor o igual que `y`.

Estrictamente hablando, los operandos de los operadores lógicos deberían ser expresiones booleanas, pero Python no es muy estricto. Cualquier número distinto de cero se interpreta como "verdadero."

```
>>> 17 and True
True
```

Esta flexibilidad puede ser útil, pero existen ciertas sutilezas en ese tipo de uso que pueden resultar confusas. Es posible que prefieras evitar usarlo de este modo hasta que estés bien seguro de lo que estás haciendo.

3.3 Ejecución condicional

Para poder escribir programas útiles, casi siempre vamos a necesitar la capacidad de comprobar condiciones y cambiar el comportamiento del programa de acuerdo a ellas. Las `sentencias condicionales` nos proporcionan esa capacidad. La forma más sencilla es la sentencia `if`:

```
if x > 0 :
    print('x es positivo')
```

La expresión booleana después de la sentencia `if` recibe el nombre de *condición*. La sentencia `if` se finaliza con un carácter de dos-puntos (:) y la(s) línea(s) que van detrás de la sentencia if van indentadas[1] (es decir, llevan una tabulación o varios espacios en blanco al principio).

Si la condición lógica es verdadera, la sentencia indentada será ejecutada. Si la condición es falsa, la sentencia indentada será omitida.

La sentencia `if` tiene la misma estructura que la definición de funciones o los bucles `for`[2]. La sentencia consiste en una línea de encabezado que termina con el carácter dos-puntos (:) seguido por un bloque indentado. Las sentencias de este tipo reciben el nombre de *sentencias compuestas*, porque se extienden a lo largo de varias líneas.

[1] el término correcto en español sería "sangradas", pero en el mundillo de la programación se suele decir que las líneas van "indentadas" (Nota del trad.)

[2] Estudiaremos las funciones en el capítulo 4 y los bucles en el capítulo 5.

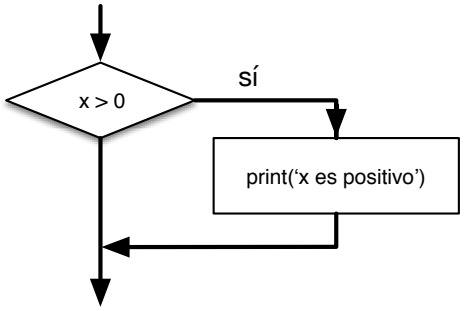

Figure 3.1: If Logic

No hay límite en el número de sentencias que pueden aparecer en el cuerpo, pero debe haber al menos una. Ocasionalmente, puede resultar útil tener un cuerpo sin sentencias (normalmente como emplazamiento reservado para código que no se ha escrito aún). En ese caso, se puede usar la sentencia `pass`, que no hace nada.

```
if x < 0 :
    pass          # ¡necesito gestionar los valores negativos!
```

Si introduces una sentencia `if` en el intérprete de Python, el prompt cambiará su aspecto habitual por puntos suspensivos, para indicar que estás en medio de un bloque de sentencias, como se muestra a continuación:

```
>>> x = 3
>>> if x < 10:
...     print('Pequeño')
...
Pequeño
>>>
```

Al usar el intérprete de Python, debe dejar una línea en blanco al final de un bloque, de lo contrario Python devolverá un error:

```
>>> x = 3
>>> if x < 10:
...     print('Pequeño')
... print('Hecho')
  File "<stdin>", line 3
    print('Hecho')
        ^
SyntaxError: invalid syntax
```

No es necesaria una línea en blanco al final de un bloque de instrucciones al escribir y ejecutar un script, pero puede mejorar la legibilidad de su código.

3.4 Ejecución alternativa

La segunda forma de la sentencia `if` es la *ejecución alternativa*, en la cual existen dos posibilidades y la condición determina cual de ellas será ejecutada. La sintaxis es similar a ésta:

```
if x%2 == 0 :
    print('x es par')
else :
    print('x es impar')
```

Si al dividir `x` por 2 obtenemos como resto 0, entonces sabemos que `x` es par, y el programa muestra un mensaje a tal efecto. Si esa condición es falsa, se ejecuta el segundo conjunto de sentencias.

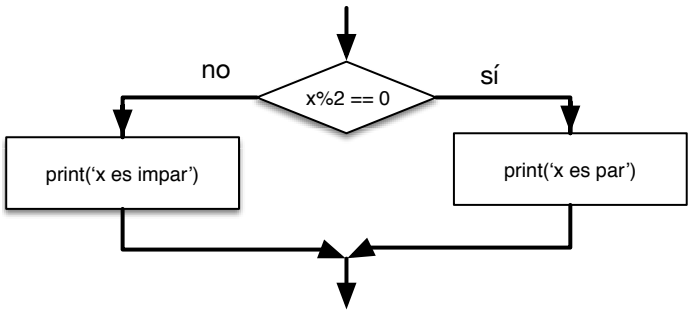

Figure 3.2: If-Then-Else Logic

Dado que la condición debe ser obligatoriamente verdadera o falsa, solamente una de las alternativas será ejecutada. Las alternativas reciben el nombre de *ramas*, dado que se trata de ramificaciones en el flujo de la ejecución.

3.5 Condicionales encadenados

Algunas veces hay más de dos posibilidades, de modo que necesitamos más de dos ramas. Una forma de expresar una operación como ésa es usar un *condicional encadenado*:

```
if x < y:
    print('x es menor que y')
elif x > y:
    print('x es mayor que y')
else:
    print('x e y son iguales')
```

`elif` es una abreviatura para "else if". En este caso también será ejecutada únicamente una de las ramas.

No hay un límite para el número de sentencias `elif`. Si hay una clausula `else`, debe ir al final, pero tampoco es obligatorio que ésta exista.

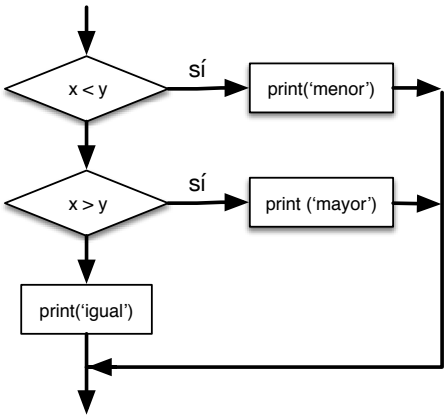

Figure 3.3: If-Then-ElseIf Logic

```
if choice == 'a':
    print('Respuesta incorrecta')
elif choice == 'b':
    print('Respuesta correcta')
elif choice == 'c':
    print('Casi, pero no es correcto')
```

Cada condición es comprobada en orden. Si la primera es falsa, se comprueba la siguiente y así con las demás. Si una de ellas es verdadera, se ejecuta la rama correspondiente, y la sentencia termina. Incluso si hay más de una condición que sea verdadera, sólo se ejecuta la primera que se encuentra.

3.6 Condicionales anidados

Un condicional puede también estar anidado dentro de otro. Podríamos haber escrito el ejemplo anterior de las tres ramas de este modo:

```
if x == y:
    print('x e y son iguales')
else:
    if x < y:
        print('x es menor que y')
    else:
        print('x es mayor que y')
```

El condicional exterior contiene dos ramas. La primera rama ejecuta una sentencia simple. La segunda contiene otra sentencia `if`, que tiene a su vez sus propias dos ramas. Esas dos ramas son ambas sentencias simples, pero podrían haber sido sentencias condicionales también.

A pesar de que el indentado de las sentencias hace que la estructura esté clara, los *condicionales anidados* pueden volverse difíciles de leer rápidamente. En general, es buena idea evitarlos si se puede.

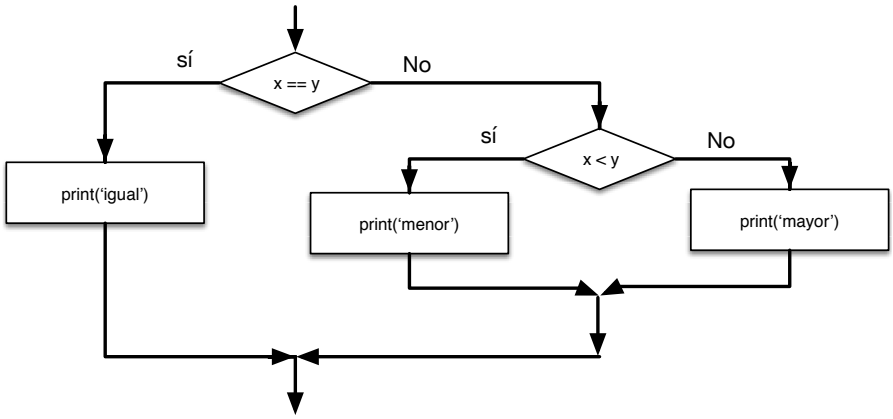

Figure 3.4: Nested If Statements

Los operadores lógicos a menudo proporcionan un modo de simplificar las sentencias condicionales anidadas. Por ejemplo, el código siguiente puede ser reescrito usando un único condicional:

```
if 0 < x:
    if x < 10:
        print('x es un número positivo con un sólo dígito.')
```

La sentencia `print` se ejecuta solamente si se cumplen las dos condiciones anteriores, así que en realidad podemos conseguir el mismo efecto con el operador `and`:

```
if 0 < x and x < 10:
    print('x es un número positivo con un sólo dígito.')
```

3.7 Captura de excepciones usando try y except

Anteriormente vimos un fragmento de código donde usábamos las funciones `input` e `int` para leer y analizar un número entero introducido por el usuario. También vimos lo poco seguro que podía llegar a resultar hacer algo así:

```
>>> velocidad = input(prompt)
¿Cual.... es la velocidad de vuelo de una golondrina sin carga?
¿Te refieres a una golondrina africana o a una europea?
>>> int(velocidad)
ValueError: invalid literal for int() with base 10:
>>>
```

Cuando estamos trabajando con el intérprete de Python, tras el error simplemente nos aparece de nuevo el prompt, así que pensamos "¡epa, me he equivocado!", y continuamos con la siguiente sentencia.

Sin embargo, si se escribe ese código en un script de Python y se produce el error, el script se detendrá inmediatamente, y mostrará un "traceback". No ejecutará la siguiente sentencia.

He aquí un programa de ejemplo para convertir una temperatura desde grados Fahrenheit a grados Celsius:

```
ent = input('Introduzca la Temperatura Fahrenheit:')
fahr = float(ent)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Código: https://es.py4e.com/code3/fahren.py
```

Si ejecutamos este código y le damos una entrada no válida, simplemente fallará con un mensaje de error bastante antipático:

```
python fahren.py
Introduzca la Temperatura Fahrenheit:72
22.2222222222

python fahren.py
Introduzca la Temperatura Fahrenheit:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(ent)
ValueError: invalid literal for float(): fred
```

Existen estructuras de ejecución condicional dentro de Python para manejar este tipo de errores esperados e inesperados, llamadas "try / except". La idea de `try` y `except` es que si se sabe que cierta secuencia de instrucciones puede generar un problema, sea posible añadir ciertas sentencias para que sean ejecutadas en caso de error. Estas sentencias extras (el bloque except) serán ignoradas si no se produce ningún error.

Puedes pensar en la característica `try` y `except` de Python como una "póliza de seguros" en una secuencia de sentencias.

Se puede reescribir nuestro conversor de temperaturas de esta forma:

```
ent = input('Introduzca la Temperatura Fahrenheit:')
try:
    fahr = float(ent)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Por favor, introduzca un número')

# Código: https://es.py4e.com/code3/fahren2.py
```

Python comienza ejecutando la secuencia de sentencias del bloque `try`. Si todo va bien, se saltará todo el bloque `except` y terminará. Si ocurre una excepción dentro del bloque `try`, Python saltará fuera de ese bloque y ejecutará la secuencia de sentencias del bloque `except`.

```
python fahren2.py
Introduzca la Temperatura Fahrenheit:72
22.2222222222

python fahren2.py
Introduzca la Temperatura Fahrenheit:fred
Por favor, introduzca un número
```

Gestionar una excepción con una sentencia `try` recibe el nombre de *capturar* una excepción. En este ejemplo, la clausula `except` muestra un mensaje de error. En general, capturar una excepción te da la oportunidad de corregir el problema, volverlo a intentar o, al menos, terminar el programa con elegancia.

3.8 Evaluación en cortocircuito de expresiones lógicas

Cuando Python está procesando una expresión lógica, como `x >= 2 and (x/y) > 2`, evalúa la expresión de izquierda a derecha. Debido a la definición de `and`, si `x` es menor de 2, la expresión `x >= 2` resulta ser `falsa`, de modo que la expresión completa ya va a resultar `falsa`, independientemente de si `(x/y) > 2` se evalúa como `verdadera` o `falsa`.

Cuando Python detecta que no se gana nada evaluando el resto de una expresión lógica, detiene su evaluación y no realiza el cálculo del resto de la expresión. Cuando la evaluación de una expresión lógica se detiene debido a que ya se conoce el valor final, eso es conocido como *cortocircuitar* la evaluación.

A pesar de que esto pueda parecer hilar demasiado fino, el funcionamiento en cortocircuito nos descubre una ingeniosa técnica conocida como *patrón guardián*. Examina la siguiente secuencia de código en el intérprete de Python:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

La tercera operación ha fallado porque Python intentó evaluar `(x/y)` e `y` era cero, lo cual provoca un runtime error (error en tiempo de ejecución). Pero el segundo

ejemplo *no* falló, porque la primera parte de la expresión `x >= 2` fue evaluada como `falsa`, así que `(x/y)` no llegó a ejecutarse debido a la regla del *cortocircuito*, y no se produjo ningún error.

Es posible construir las expresiones lógicas colocando estratégicamente una evaluación como *guardián* justo antes de la evaluación que podría causar un error, como se muestra a continuación:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

En la primera expresión lógica, `x >= 2` es `falsa`, así que la evaluación se detiene en el `and`. En la segunda expresión lógica, `x >= 2` es `verdadera`, pero `y != 0` es `falsa`, de modo que nunca se alcanza `(x/y)`.

En la tercera expresión lógica, el `y != 0` va *después* del cálculo de `(x/y)`, de modo que la expresión falla con un error.

En la segunda expresión, se dice que `y != 0` actúa como *guardián* para garantizar que sólo se ejecute `(x/y)` en el caso de que `y` no sea cero.

3.9 Depuración

Los "traceback" que Python muestra cuando se produce un error contienen un montón de información, pero pueden resultar abrumadores. Las partes más útiles normalmente son:

- Qué tipo de error se ha producido, y
- Dónde ha ocurrido.

Los errores de sintaxis (syntax errors), normalmente son fáciles de localizar, pero a veces tienen trampa. Los errores debido a espacios en blanco pueden ser complicados, ya que los espacios y las tabulaciones son invisibles, y solemos ignorarlos.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

En este ejemplo, el problema es que la segunda línea está indentada por un espacio. Pero el mensaje de error apunta a `y`, lo cual resulta engañoso. En general, los mensajes de error indican dónde se ha descubierto el problema, pero el error real podría estar en el código previo, a veces en alguna línea anterior.

Ocurre lo mismo con los errores en tiempo de ejecución (runtime errors). Supón que estás tratando de calcular una relación señal-ruido en decibelios. La fórmula es $SNR_{db} = 10 \log_{10}(P_{senal}/P_{ruido})$. En Python, podrías escribir algo como esto:

```
import math
int_senal = 9
int_ruido = 10
relacion = int_senal / int_ruido
decibelios = 10 * math.log10(relacion)
print(decibelios)

# Código: https://es.py4e.com/code3/snr.py
```

Pero cuando lo haces funcionar, obtienes un mensaje de error[3]:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibelios = 10 * math.log10(relacion)
OverflowError: math range error
```

El mensaje de error apunta a la línea 5, pero no hay nada incorrecto en ese línea. Para encontrar el error real, puede resultar útil mostrar en pantalla el valor de `relacion`, que resulta ser 0. El problema está en la línea 4, ya que al dividir dos enteros se realiza una división entera. La solución es representar la intensidad de la señal y la intensidad del ruido con valores en punto flotante.

En general, los mensajes de error te dicen dónde se ha descubierto el problema, pero a menudo no es ahí exactamente donde se ha producido.

3.10 Glosario

condición La expresión booleana en una sentencia condicional que determina qué rama será ejecutada.

condicional anidado Una sentencia condicional que aparece en una de las ramas de otra sentencia condicional.

condicional encadenado Una sentencia condicional con una serie de ramas alternativas.

[3] En Python 3.0, ya no se produce el mensaje de error; el operador de división realiza división en punto flotante incluso con operandos enteros.

cortocircuito Cuando Python va evaluando una expresión lógica por tramos y detiene el proceso de evaluación debido a que ya conoce el valor final que va a tener el resultado sin necesidad de evaluar el resto de la expresión.

cuerpo La secuencia de sentencias en el interior de una sentencia compuesta.

expresión booleana Un expresión cuyo valor puede ser o bien `Verdadero` o bien `Falso`.

operadores de comparación Uno de los operadores que se utiliza para comparar dos operandos: `==`, `!=`, `>`, `<`, `>=`, y `<=`.

operador lógico Uno de los operadores que se combinan en las expresiones booleanas: `and`, `or`, y `not`.

patrón guardián Cuando construimos una expresión lógica con comparaciones adicionales para aprovecharnos del funcionamiento en cortocircuito.

rama Una de las secuencias alternativas de sentencias en una sentencia condicional.

sentencia compuesta Una sentencia que consiste en un encabezado y un cuerpo. El encabezado termina con dos-puntos (:). El cuerpo está indentado con relación al encabezado.

sentencia condicional Una sentencia que controla el flujo de ejecución, dependiendo de cierta condición.

traceback Una lista de las funciones que se están ejecutando, que se muestra en pantalla cuando se produce una excepción.

3.11 Ejercicios

Ejercicio 1: Reescribe el programa del cálculo del salario para darle al empleado 1.5 veces la tarifa horaria para todas las horas trabajadas que excedan de 40.

```
Introduzca las Horas: 45
Introduzca la Tarifa por hora: 10
Salario: 475.0
```

Ejercicio 2: Reescribe el programa del salario usando `try` y `except`, de modo que el programa sea capaz de gestionar entradas no numéricas con elegancia, mostrando un mensaje y saliendo del programa. A continuación se muestran dos ejecuciones del programa:

```
Introduzca las Horas: 20
Introduzca la Tarifa por hora: nueve
Error, por favor introduzca un número

Introduzca las Horas: cuarenta
Error, por favor introduzca un número
```

Ejercicio 3: Escribe un programa que solicite una puntuación entre 0.0 y 1.0. Si la puntuación está fuera de ese rango, muestra un mensaje de error. Si la puntuación está entre 0.0 y 1.0, muestra la calificación usando la tabla siguiente:

```
Puntuación Calificación
>= 0.9     Sobresaliente
>= 0.8     Notable
>= 0.7     Bien
>= 0.6     Suficiente
< 0.6      Insuficiente

Introduzca puntuación: 0.95
Sobresaliente

Introduzca puntuación: perfecto
Puntuación incorrecta

Introduzca puntuación: 10.0
Puntuación incorrecta

Introduzca puntuación: 0.75
Bien

Introduzca puntuación: 0.5
Insuficiente
```

Ejecuta el programa repetidamente, como se muestra arriba, para probar con varios valores de entrada diferentes.