logica-condicional-control-de-flujo

January 28, 2020

1 Logica Condicional y Control de Flujo

Hasta el momento, todo el codigo que hemos visto ha sido incondicional. El codigo no tuvo que tomar decision alguna. Cada linea de codigo es ejecutada en el orden que es escrita o en el orden en que las funciones han sido llamadas, con posibles repeticiones adentro del `loop`.

En esta seccion aprenderemos como escribir programas que realizan distintas acciones a base de distintas condiciones, utilizando **logica condicional**.

Vamos a: 1. Comparar valores con dos o mas variables 2. Escribir declaraciones `if` para controlar el flujo de nuestro programa 3. Gestionar errores con `try` y `except` 4. Aplicar logica condicional para crear simulaciones simples

1.1 Comparacion de valores

Logica condicional radica en la realizacion condicionada de distintas acciones. Condicionada porque la realizacion de las mismas depende si una expresion, llamada condicion, es veridica o no.

Las condiciones en muchas ocasiones radican en comparar dos valores, e.g. si un valor es mas alto que el otro o si los dos valores son iguales. Un estandar de simbolos utilizado para este fin son los comparadores booleanos.

Comparador	Ejemplo	Significado
>	a > b	a es mayor que b
<	a < b	a es menor que b
>=	a >= b	a es mayor o igual a b
<=	a <= b	a es menor o igual a b
!=	a != b	a no es igual a b
==	a == b	a es igual a b

[ ]:
```
# mayor
2 > 1
```

[ ]:
```
# menor
1 < 2
```

[ ]:
```
# mayor o igual
1 >= 1
```

[ ]:
```
# menor o igual
1 <= 1
```

[ ]:
```
# anti-equivalencia
1 != 1
```

[ ]:
```
# equivalencia
1 == 1
```

El termino **boolean** es derivado del apellido del Matematico ingles **George Boole**, cuyas obras sentaron las bases de la computacion moderna. En honor a Boole, la logica condicional es conocida como **logica booleana** y las condicionales son referidas como **expresiones booleanas**.

Existe tambien un tipo de dato llamado **boolean** o formalmente `bool`, que consiste en dos valores: `True` y `False`

[ ]:
```
type(True)
```

[ ]:
```
type(False)
```

[ ]:
```
# evaluando una condicional siempre retorna un valor booleano
3 > 5
```

[ ]:
```
# tambien podemos comparar caracteres
'a' == 'a'
```

[ ]:
```
'a' == 'b'
```

[ ]:
```
# sorprendido?
'a' < 'b'  # orden lexicografico
```

[ ]:
```
# la letra a no viene despues de la b
'a' > 'b'
```

[ ]:
```
# el orden lexicografico se extiende a strings con dos o mas caracteres
'arte' < 'arbol'
```

1.1.1 Ejercicios

Adivinemos el resultado de las siguientes expresiones:

```
1 < = 1
1 != 1
1 != 2
'bien' != 'mal'
'bien' !- 'Bien'
123 == '123'
```

Adivine el comparador booleano indicado para que la expresion evalue a True

```
3 _ 4
10 _ 5
'jorge' _ 'jose'
42 _ '42'
```

1.2 Algo de logica

En adicion a los comparadores booleanos, Python tiene palabras especiales llamadas **operadores logicos** que puede ser utilizados para combinar expresioens booleanas. Existen tres operadores logicos: 1. `and` 2. `or` 3. `not`

1.2.1 La palabra `and`

Considera la proxima declaracion: 1. Gatos tienen cuatro patas 2. Gatos tienen colas

En general, ambas declaraciones son veridicas. Si unimos ambas declaraciones con la palabra `and`, la expresion booleada sigue siendo cierta porque ambas cosas son ciertas. Esta palabra retorna `True` si ambos lados de la palabra evaluan a `True`, de lo contrario retorna `False`

[ ]:
```
1 < 2 and 3 < 4  # ambos ciertos
```

[ ]:
```
1 < 2 and 3 > 4  # un cierto un falso
```

[ ]:
```
2 < 1 and 4 < 3  # ambos falsos
```

[ ]:
```
True and True
```

[ ]:
```
True and False
```

[ ]:
```
False and True
```

[ ]:
```
False and False
```

1.2.2 La palabra `or`

[ ]:
```
1 < 2 or 3 < 4  # True or True
```

[ ]:
```
1 < 2 or 2 < 1  # True or False
```

[ ]:
```
2 < 1 or 4 < 3  # False or False
```

[ ]:
```
2 < 1 or 3 < 4  # False or True
```

1.2.3 La palabra `not`

Esta palabra revierte la veracidad de una expresion

[ ]:
```
not True
```

[ ]:
```
not False
```

[ ]:
```
not True == False
```

[ ]:
```
False == not True  # error por regla de precedencia de operador
```

El orden de precedencia para operadores logicos y booleanos, de lo mas alto a lo mas bajo es: 1. < <= == >= > 2. not 3. and 4. or

[ ]:
```
# el problema anteriormente es que por precedencia, se evaluaba a False == not
False == (not True)  # lo arreglamos con parentesis
```

1.2.4 Construyendo Expresiones Complejas

Podemos combinar `and`, `or` y `not` con `True` y `False` para crear expresiones mas complejas.

[ ]:
```
# un ejemplo de una expresion mas compleja
True and not (1 != 1)
```

[ ]:
```
# la anterior expresion se interpreta asi:
True and not (False)
```

[ ]:
```
# luego asi
True and True
```

[ ]:
```
("A" != "A") or not (2 >= 3)
```

[ ]:
```
# se evalua asi:
(False) or not (False)
```

[ ]:
```
# luego asi:
False or True
```

Finalmente, toda vez que cualquier expresion compuesta con `or` es `True` si una de las expresiones a la izquierda o derecha de `or` es `True`, podemos concluir que el resultado es `True`

Agrupando expresiones on una declaracion condicional compuesta que tiene parentesis mejora la la lectura, pero a veces el parentesis es requerido para producir el resultado esperado.

[ ]:
```
# por ejemplo, puede ser que esperamos que esto retorne True
True and False == True and False
```

La razon es que el operador `==` tiene precedencia ante `and`, entonces Python interpreta la expresion como:

```
True and (False == True) and False
```

Lo que se convierte en:

```
True and False and False
```

Lo que equivale a `False`

[ ]:
```
# si agregamos el parentesis
(True and False) == (True and False)
```

1.2.5 Ejercicios

Cual es el resultado de las siguientes expresiones?

```
(1 <= 1) and (1 != 1)
not (1 != 2)
("good" != "bad") or False
("good" != "Good") and not (1 == 1)
```

Agrega parentesis donde sea necesario para que cada una de las siguientes expresiones evalue a `True`:

```
False == not True
True and False == True and False
not True and "A" == "B"
```

1.3 Control de flujo de un programa

Ahora que podemos comparar valores con comparadores booleanos y podemos construir declaraciones complejas con operadores logicos, vamos a agregar algo de logica a nuestro codigo para que realice diferentes acciones para diferentes condiciones.

1.3.1 La declaracion `if`

Una declaracion `if` le dice a Python que ejecute de forma exclusiva una porcion de codigo si una condicion se cumple.

[ ]:
```
# por ejemplo
if 2 + 2 == 4:
    print('2 + 2 = 4')
```

Como el `while` loop, el `if` tiene tres partes: 1. La palabra `if` 2. La condicion de prueba, seguida por un colon 3. Un bloque indentado que es ejecutado si una condicion es `True`

[ ]:
```
# cuando la condicion es falsa
if 2 + 2 == 5:
    print('Esto no existe!')
print('Esto si')
```

[ ]:
```
nota = 4.5
if nota >= 3.5:
    print('Pasaste la clase!')
print('Gracias por asistir.')
```

[ ]:
```
nota = 3.0

if nota >= 3.5:
    print('Pasaste!')

if nota < 3.5:
    print('Te quedaste, lo siento!')

print('Gracias por asistir.')
```

1.3.2 La palabra `else`

`else` es utilizado despues de un **`if`**, a fin de ejecutar una porcion de codigo solo si la condicion del **`if`** no es veridica (no llega a ejecutarse).

[ ]:
```
nota = 3.0

if nota >= 3.5:
    print('Pasaste!')

else:
    print('Te quedaste, lo siento.')

print('Gracias por asistir.')
```

Las palabras **`if`** y **`else`** funcionan bien cuando hay que revisar exactamente dos (2) estados / condiciones. Sin embargo, hay veces que necesitamos revisar tres (3) o mas condiciones. Para esto, utilizamos la palabra **`elif`**.

1.3.3 La palabra `elif`

La palabra `elif` es corto por `else if` y se puede utilizar para agregar condiciones adicionales despues del `if`. Como los `if`, los `elif` tienen tres (3) partes: 1. La palabra `elif` 2. La condicion de prueba, seguida por un colon 3. Un bloque de codigo indentado que se ejecuta si la condicion evalua a `True`

[ ]:
```
nota = 4.0  # 1

if nota >= 4.5:  # 2
    print('Pasaste con cuadro de honor!')

elif nota >= 4.0:  # 3
    print('Pasastes con buena nota, felicidades!')

elif nota >= 3.5:  # 4
    print('Pasastes, pero tienes que esforzarte mas')

else:  # 5
```

```
    print('Lo siento, no pasastes, nos vemos el proximo año misma clase misma␣
 ↪hora!')

print('Gracias por asistir.')  #6
```

1.4 Declaraciones `if` anidadas

Al igual que los `for` y `while` loops, tambien podemos anidar una declaracion `if` adentro de otra para crear estructuras de decisiones un poco mas complejas.

Considera el proximo escenario. Dos personas juegan uno contra uno. Debemos decidir quien gana dependiendo de los puntajes y el deporte que estan jugando: 1. Si los dos jugadores estan jugando baloncesto, el jugador con el puntaje mas alto gana. 2. Si estan jugando golf, el jugador con el puntaje mas bajo gana. 3. En cualquier de los dos deportes, si el puntaje es el mismo, es un empate.

[ ]:
```
deporte = input('Ingresa un deporte: ')
puntaje_1 = input('Ingresa el puntaje del jugador 1: ')
puntaje_2 = input('Ingresa el puntaje del jugador 2: ')

# 1
if deporte.lower() == 'baloncesto':
    if puntaje_1 == puntaje_2:
        print('Empate!')
    elif puntaje_1 > puntaje_2:
        print('Ganador: Jugador 1!')
    else:
        print('Ganador: Jugador 2!')

# 2
elif deporte.lower() == 'golf':
    if puntaje_1 == puntaje_2:
        print('Empate!')
    elif puntaje_1 > puntaje_2:
        print('Ganador: Jugador 2!')
    else:
        print('Ganador: Jugador 1!')

else:
    print('Deporte desconocido.')

# en total hay 7 posibles resultados
```

[ ]:
```
# simplifiquemoslo un poco mas

# 1
if puntaje_1 == puntaje_2:
    print('Empate!')
```

```
elif deporte.lower() == 'baloncesto':
    elif puntaje_1 > puntaje_2:  # 2
        print('Ganador: Jugador 2!')
    else:  # 3
        print('Ganador: Jugador 1!')

elif deporte.lower() == 'golf':
    elif puntaje_1 > puntaje_2:  # 4
        print('Ganador: Jugador 2!')
    else:  # 5
        print('Ganador: Jugador 1!')

else:  # 6
    print('Deporte desconocido.')


# 6 distintas posibilidades
```

1.4.1 Lo simplificamos aun mas?

A ver: * El jugador 1 gana si el deporte es baloncesto y su puntaje es el mas alto * El jugador 1 gana si el deporte es golf y su puntaje es el mas bajo

[ ]:
```
# lo anterior lo podemos representar de la siguiente manera:
jugador_1_gana_baloncesto = deporte == 'baloncesto' and puntaje_1 > puntaje_2
jugador_1_gana_golf = deporte == 'golf' and puntaje_1 < puntaje_2
jugador_1_gana = jugador_1_gana_baloncesto or jugador_1_gana_golf
```

[ ]:
```
jugador_1_gana_baloncesto
```

[ ]:
```
jugador_1_gana_golf
```

[ ]:
```
jugador_1_gana
```

[ ]:
```
if puntaje_1 == puntaje_2:
    print('Empate!')  # 1

elif deporte.lower() == 'baloncesto' or deporte.lower() == 'golf':
    jugador_1_gana_baloncesto = deporte == 'baloncesto' and puntaje_1 >␣
 ↪puntaje_2
    jugador_1_gana_golf = deporte == 'golf' and puntaje_1 < puntaje_2
    jugador_1_gana = jugador_1_gana_baloncesto or jugador_1_gana_golf

    if jugador_1_gana:
        print('Ganador: Jugador 1!')  # 2
    else:
        print('Ganador: Jugador 2!')  # 3
```

```
else:
    print('Deporte desconocido')  # 4

# solo hay 4 maneras en el cual el programa puede ejecutar
```

1.4.2 Ejercicios

Escriba un script que le solicita al usuario ingresar una palabra. Guarde la palabra en una variable, y luego imprima si la longitud del string es: 1. menos de 5 caracteres 2. mas de 5 caracteres 3. igual a 5 caracteres

Utilize las declaraciones `if`, `elif` y `else`.

1.5 Reto: Encuentra los factores de un numero

El factor de un numero positivo entero `n` es cualquier numero positivo entero que es: * igual o menor a `n` * divide `n` enteramente

Por ejemplo, `3` es un factor de `12` porque `12` dividido entre `3` es `4`, sin ningun monto restante. Sin embargo, `5` no es un factor de `12` porque `5` por `2` es `10` y queda un monto restante de `2`.

Escribe un script `factores.py` que le solicita al usuario un numero positivo entero y luego imprime los factores de ese numero.

Un ejemplo:

Ingresa un numero positivo entero: 12 * 1 es factor de 12 * 2 es factor de 12 * 3 es factor de 12 * 4 es factor de 12 * 6 es factor de 12 * 12 es factor de 12

Pista: el operador `%`

1.6 Rompiendo el patron

Sabemos como repetir un bloque de codigo varias veces utilizando un `for` o `while` loop. Loops son utiles para repetir operaciones y procesar varios inputs.

Combinando declaraciones `if` con `for` loops abre una serie de tecnicas para controlar como se ejecuta un codigo.

En esta seccion aprenderemos a escribir declaraciones `if` que estan anidadas en `for` loops y aprenderemos de two (2) palabras - `break` y `continue` - que nos permiten controlar aun mas el flujo de ejecucion a atraves de un ciclo.

1.6.1 `if` y `for` loops

Un bloque de codigo en un `for` loop es como cualquier otro bloque de codigo. Es decir, podemos anidar un `if` en un `for` loop, tal como lo hariamos en cualquier otra parte del codigo.

[ ]:
```
suma_de_pares = 0

for n in range(1, 100):
```

```
    if n % 2 == 0:  # si es par (divisible por 2)
        suma_de_pares = suma_de_pares + n

print(suma_de_pares)
```

Primero, la `suma_de_pares` inicia en `0`. Luego el programa cicla sobre los numeros `1` a `99`, suma el par a la suma asignada a la `asuma_de_pares`, y luego hace lo mismo hasta llegar al numero 99. El valor final de `suma_de_pares` es `2450`.

1.7 `break`

La palabra `break` le dice a Python que literalmente rompa el ciclo, que salga del ciclo. Es decir, el ciclo se detiene completamente y el codigo despues del loop es ejecutado.

[ ]:
```
for n in range(0, 4):
    if n == 2:  # si el numero es igual a 2
        break  # rompemos el ciclo, la siguiente linea no se ejecuta
    print(n)

print(f'Terminamos con n = {n}')
```

1.8 `continue`

El palabra `continue` es utilizada para saltarse cualquier porcion restante de codigo en un ciclo y continuar a la proxima iteracion.

[ ]:
```
for i in range(0, 4):
    if i == 2:  # este no se imprime
        continue
    print(i)

print(f'Terminamos con n = {n}')
```

1.8.1 `for...else`

Loops pueden tener su propio `else` en Python, aunque esta estructura no es utilizada frecuentemente

[ ]:
```
frase = 'marca el lugar'

for caracter in frase:
    if caracter == 'X':
        break
else:  # ejecuta solo si el for loop termina sin que el break se ejecuta
    print('Nunca hubo una X')
```

[ ]:
```
frase = 'X marca el lugar'
```

```
for caracter in frase:
    if caracter == 'X':
        break
else:  # ejecuta solo si el for loop termina sin que el break se ejecuta
    print('Nunca hubo una X')
```

[ ]:
```
# ejemplo practico

for n in range(3):
    contraseña = input('Ingrese su contraseña: ')
    if contraseña == 'I<3Panama':
        break
    print('Incorrecto')
else:
    print('Actividad sospechosa, las autoridades han sido alertadas.')
```

1.9 Ejercicios

1. Utilizando `break`, escribe un programa que solicite al usuario dato de entrada, rompiendo el ciclo cuando el usuario ingrese `q` o `Q`.
2. Utilizando `continue`, escribe un programa que cicle sobre los numeros `1` a `50` e imprime todos los numeros que no son multiples de `3`.

1.10 Recuperacion de Errores

A fin de crear programas robustos, necesitamos gestionar errores causados por datos de entrada del usuario o cualquier otra fuente impredecible. En este seccion aprenderemos como hacerlo.

1.10.1 Un zoologico de excepciones

Cuando nos encontramos con una error, que tambien se le conoce como excepcion, es util saber que fue lo que ocurrió. Python tiene un numero de tipos de excepciones incorporadas que describen una variedad de errores. Ya hemos visto algunos de esos errores anteriormente, no obstante, visitaremos los mas importantes.

ValueError Este error ocurre cuando una operacion se encuentra con un valor invalido.

[ ]:
```
int('esto no es un numero')
```

TypeError Occure cuando una operacion es realizada en un valor del tipo incorrecto.

[ ]:
```
'1' + 2
```

Name Error Ocurre ucuando se intenta usar un nombre de variable que no ha sido definido todavia

[ ]:
```
print(no_existe)
```

ZeroDivisionError Ocurre cuando el divisor de una division es `0`

[ ]:
```
1 / 0
```

OverflowError Ocurre cuando el resultado de una operacion aritmetica es muy grande.

[ ]:
```
pow(2.0, 1_000_000)
```

1.10.2 try y except

En ocasiones debemos predecir que una excepcion o error pudiese ocurrir. En vez de dejar que el programa se detenga erroneamente, podemos agarrar el error si ocurre y podemos realizar alguna operacion como consecuencia del mismo.

Por ejemplo, si un usuario ingresa un dato de entrada no deseado, debemos informarle que han ingresado un valor invalido. No obstante, para prevenir que el programa se detenga, debemos utilizar `try` y `except`, que se traduce a *intento* y *excepto*.

[ ]:
```
try:
    numero = int(input('Ingresa un numero: '))  # si no es un numero
except ValueError:  # int() alza un error ValueError, aqui lo atrapamos
    print('Eso no fue un numero')
```

[ ]:
```
def divide(num1, num2):
    try:
        print(num1 / num2)
    except (TypeError, ZeroDivisionError):
        print('encontramos un error')
```

[ ]:
```
divide(1,0)
```

[ ]:
```
divide(10, 5)
```

[ ]:
```
# en ocasiones es mejor agarrar cada error individualmente
def divide(num1, num2):
    try:
        print(num1 / num2)
    except TypeError:
        print('Ambos argumentos deben ser un numero')
    except ZeroDivisionError:
        print('num2 no debe ser 0')
```

[ ]:
```
divide(1, 0)
```

[ ]:
```
divide('a', 0)
```

[ ]:
```
divide(0, 1)
```

1.10.3 Solo el `except`

```
[ ]: # este patron no es buena practica porque es propenso a esconder todo error
try:
    # nuestro codigo
except:
    print("Algo pasó!")
```

1.10.4 Ejercicios

1. Escriba un script que repetidamente le solicita al usuario que ingrese un numero entero, mostrando un mensaje de "intene nuevamente" cuando se rescata el error `ValueError` que es alzado si el usuario no ingresa un numero entero. Si el usuario ingresa el numero, el programa debe mostrar el numero nuevamente.
2. Escriba un programa que le solicita el usuario ingresar un string y un numero `n`. Luego muestre el caracter ubicado en el indice `n` del string. Haga uso de herramientas para la gestion de errores a fin de asegurar que el programa no se interrumpa cuando el usuario ingrese no ingrese un numero o cuando el indice queda fuera del rango del string. El programa deberia mostrar un mensaje diferente dependiendo del error que ocurra.

1.11 Simulacion de Eventos y Calculo de Probabilidades

En esta seccion, vamos aplicar algunos conceptos que hemos aprendido sobre ciclos y logica condicional a un problema real: simulando evento y calculando probabilidades.

Vamos a ejecutar una simulacion conocida como el experimento **Monte Carlo**.

Cada experimento consiste de una prueba u ensayo, que consiste en un proceso repetible - como lanzar una moneda - que genera un resultado, e.g. cara o sello. La prueba es repetida muchas veces a fin de calcular la probabilidad que algun resultado ocurra.

Para lograrlo, necesitaremos agregar aleatoridad en nuestro codigo.

1.11.1 El modulo `random`

Python nos proporciona varias funciones para generar numeros aleatorios en el modulo **random**. Un **modulo** es una coleccion de codigo relacionado. La libreria estandar de Python es una coleccion organizada de modulos que podemos importar en nuestro codigo, a fin de resolver varios problemas.

```
[ ]: import random
```

```
[ ]: random.randint(1, 10)
```

`.randint()` es un metodo del modulo `random`, es por eso que escribimos `random` primero, luego un punto `.` y luego el metodo `randint()` con sus dos parametros.

```
[ ]: help(random.randint)
```

Este metodo `randint()` retorna un numero entero aleatorio en el rango `[a, b]`, incluyendo ambos extremos. Observamos que la firma de la funcion en la documentacion indica la presencia de dos

(2) argumentos a y b.

[ ]:
```
random.randint(1, 10)
```

[ ]:
```
random.randint(0, 1)  # 0 o 1
```

1.11.2 Lanzamiento de moneda

Como podemos simular un lanzamiento?

Una moneda tiene dos (2) lados: cara o sello. Si decimos que 0 es cara y 1 es sello, podemos utilizar `randint(0, 1)` para simular un lanzamiento.

[ ]:
```
random.randint(0, 1)
```

En este experimento, queremos saber, cual es la probabilidad de que salga cara o que salga sello.

Necesitamos simular el lanzamiento muchas veces, para tener certeza de la probabilidad, y darle seguimiento a las veces que salga cara y de que salga sello.

Cada prueba tiene dos etapas: 1. Lanzamiento de moneda 2. Determinar si salio cara, en cuyo caso sumar +1 a la variable que representa el numero de veces que sale cara, y en caso contrario, sumar +1 a la variable que representa el numero de veces que sale sello.

Vamos a repetir esta prueba unas 10,000 veces (podemos utilizar `range(10_000)`).

Empezemos escribiendo una funcion llamada `lanzamiento()` que aleatoriamente retorna el string `cara` o `sello`.

[ ]:
```
import random
def lanzamiento():
    """Aleatoriamente retorna cara o sello"""
    if random.randint(0, 1) == 0:
        return "cara"
    else:
        return "sello"
```

[ ]:
```
# ahora podemos escribir un ciclo que haga el lanzamiento 10,000 veces

cara = 0
sello = 0

for prueba in range(100_000): # de 0 a 99,999
    if lanzamiento() == 'cara':
        cara = cara + 1
    else:
        sello = sello + 1
```

[ ]:
```
print(cara, sello)
proporcion = cara / sello
print(f'La probabilidad de cara a sello es de {proporcion:.2f}')
```

1.11.3 Ejercicios

1. Escribe una funcion `dados()` que utiliza `randint()` para simular un lanzado de dados (de 1 a 6).
2. Escribe un script que simula 10,000 lanzadas de dados y muestra el numero promedio lanzado.

1.12 Reto: Simulacion de un lanzamiento de moneda

En promedio, cuantos lanzamientos de moneda se necesitan para que la secuencia contenta cara y sello? En otras palabras, si lanazamos la moneda y sale cara, cuantas veces mas tenemos que lanzar para que salga sello? Por ejemplo, puede salir cara, cara y sello, lo que suman 3 lanzamientos.

Escriba una simulacion de 10,000 pruebas e imprima el numero promedio de lanzamientos requeridos para que la secuencia tenga cara y sello.

[ ]:
```
veces = []
cara, sello = 0, 0
for prueba in range(10_000): # de 0 a 99,999
    cara_o_sello = lanzamiento()
    if cara > 0 and sello > 0:
        veces.append(cara + sello)
        cara, sello = 0, 0
    elif cara_o_sello == 'cara':
        cara = cara + 1
    else:
        sello = sello + 1
```

[ ]:
```
import statistics
help(statistics)
```

[ ]:
```
statistics.mode(veces)  # el numero que salió mas veces
```

[ ]:
```
statistics.mean(veces)  # la suma de las veces entre las veces que viró
```

1.13 Resumen

En esta seccion hemos visto declaracion condicionales y logica condicional. Hemos visto como comparar valores utilizando operadores de comparaciones como <, >, <=, >=, != y ==. Tambien vimos como construir declaraciones complejas utilizando `and`, `or`, and `not`.

Vimos como controlar el flujo del programa utilizando `if`, y hemos aprendido como crear vertientes en el programa utilizando `if`, `else` o `if`, `elif` y `else`. Además, aprendimos como controlar la ejecucion del codigo adentro de un bloque `if` utilizando `break` y `continue`.

Aun mas, aprendimos sobre `try` y `except` para la gestion de errores de ejecucion. Este es un patron sumamente util toda vez que permite que el programa maneje errores esperados y mantenga el programa andando.

Finalmente, utilizamos el modulo `random` para simular el lanzamiento de una moneda.

> La logica es el comienzo de la sabiduria, no el fin.

Spock, *Star Trek*