02 - Strings, Input-Output, Branching

Programas de ramificación

Los tipos de cálculo que hemos visto hasta ahora se denominan programas en línea recta. Ejecutan una sentencia tras otra en el orden en que aparecen y se detienen cuando se quedan sin sentencias. Los tipos de cálculos que podemos describir con programas en línea recta no son muy interesantes. De hecho, son francamente aburridos.

Los programas ramificados son más interesantes. La *sentencia de bifurcación más sencilla es una condicional*. Una sentencia condicional tiene tres partes:

- Una **prueba**, es decir, una expresión que se evalúa como Verdadero o Falso.
- Un **bloque de código** que se ejecuta si la prueba es Verdadera.
- Un **bloque de código opcional** que se ejecuta si la prueba es igual a Falso.

Después de ejecutar una sentencia condicional, la ejecución se reanuda en
el código que sigue a la sentencia.

En Python, una sentencia condicional tiene la forma:

```
if Boolean expression:
    block of code
else:
    block of code
# O
if Boolean expression:
    block of code
```

`Boolean expression` indica que cualquier expresión que evalúe a Verdadero o Falso puede seguir a la palabra reservada `if`, y `block of code` indica que cualquier secuencia de sentencias Python puede seguir a `else:`.

Considere el siguiente programa que imprime `«Even»` si el valor de la variable `x` es par y `«Odd»` en caso contrario:

```
if x % 2 == 0:
    print('Even')
```

```
else:
    print('Odd')
    print('Done with conditional')
```

La expresión `x % 2 == 0` se evalúa como Verdadero cuando el resto de x dividido por 2 es 0, y Falso en caso contrario. Recuerda que `==` se utiliza para comparar, ya que `=` está reservado para asignar.

La **sangría tiene un significado semántico en Python**. Por ejemplo, si la última sentencia del código anterior estuviera sangrada, formaría parte del bloque de código asociado al `else`, en lugar del bloque de código que sigue a la sentencia condicional.

Python es inusual en el uso de la sangría de esta manera. *La mayoría de los otros lenguajes de programación usan símbolos de corchetes para delinear bloques de código*, por ejemplo, C encierra bloques entre llaves `{ }`. Una ventaja del enfoque de Python es que asegura que la estructura visual de un programa es una representación exacta de su estructura semántica. Dado que la *indentación es semánticamente importante*, la noción de línea también lo es. Una línea de código que es demasiado larga para leerla fácilmente puede dividirse en múltiples líneas en la pantalla terminando cada línea en la pantalla, excepto la última, con una barra invertida (`\`). Por ejemplo,

```
x = 11111111111111111111111111111111 + 22222222222333222222222 +\
333333333333333333333333333333
```

Las líneas largas también se pueden envolver utilizando la continuación de línea implícita de Python. Esto se hace con corchetes, es decir, paréntesis, corchetes y llaves. Por ejemplo,

```
x = 11111111111111111111111111111111 +
22222222222333222222222 +
333333333333333333333333333333
```

Se interpreta como dos líneas (y por lo tanto produce un error de sintaxis `"unexpected indent"` mientras que

```
x = (11111111111111111111111111111111 +
22222222222333222222222 +
333333333333333333333333333333)
```

Se interpreta como una sola línea debido a los paréntesis. Muchos programadores de Python prefieren usar continuaciones de línea implícitas a usar una barra invertida. Lo más común es que los

programadores rompan las líneas largas en las comas o en los operadores.

Volviendo a las condicionales, cuando el bloque verdadero o el bloque falso de una condicional contiene otra condicional, se dice que las *sentencias condicionales están anidadas*. El siguiente código contiene condicionales anidadas en ambas ramas de la sentencia `if` de nivel superior.

```
if x % 2 == 0:
    if x % 3 == 0:
        print('Divisible by 2 and 3')
    else:
        print('Divisible by 2 and not by 3')
elif x % 3 == 0:
    print('Divisible by 3 and not by 2')
```

`elif` en el código anterior significa `«else if»`.
A menudo es conveniente utilizar una expresión booleana compuesta en la prueba de una condicional, por ejemplo,

```
if x < y and x < z:
    print('x is least')
elif y < z:
    print('y is least')
else:
    print('z is least')
```

Ejercicio manual

Escribe un programa que examine tres variables –x, y y z– e imprima el mayor número impar entre ellas. Si ninguno de ellos es impar, debe imprimir el valor más pequeño de los tres.

Puedes atacar este ejercicio de varias maneras. Hay ocho casos distintos a considerar: todos son impares (un caso), exactamente dos de ellos son impares (tres casos), exactamente uno de ellos es impar (tres casos), o ninguno de ellos es impar (un caso). Por lo tanto, una solución simple implicaría una secuencia de ocho sentencias `if`, cada una con una única sentencia `print`:

```
if x % 2 != 0 and y % 2 != 0 and z % 2 != 0:
    print(max(x, y, z))
if x % 2 != 0 and y % 2 != 0 and z % 2 == 0:
    print(max(x, y))
if x % 2 != 0 and y % 2 == 0 and z % 2 != 0:
    print(max(x, z))
```

```
if x % 2 == 0 and y % 2 != 0 and z % 2 != 0:
    print(max(y, z))
if x % 2 != 0 and y % 2 == 0 and z % 2 == 0:
    print(x)
if x % 2 == 0 and y % 2 != 0 and z % 2 == 0:
    print(y)
if x % 2 == 0 and y % 2 == 0 and z % 2 != 0:
    print(z)
if x % 2 == 0 and y % 2 == 0 and z % 2 == 0:
    print(min(x, y, z))
```

Eso hace el trabajo, pero es bastante engorroso. No sólo son 16 líneas de código, sino que las variables se comprueban repetidamente para ver si son extrañas.
El siguiente código es más elegante y eficiente:

```
answer = min(x, y, z)
if x % 2 != 0:
    answer = x
if y % 2 != 0 and y > answer:
    answer = y
if z % 2 != 0 and z > answer:
    answer = z
print(answer)
```

El código se basa en un paradigma de programación común. Comienza asignando un valor provisional a una variable (respuesta), actualizándolo cuando procede, y luego imprimiendo el valor final de la variable. Observa que comprueba si cada variable es impar exactamente una vez, y contiene una única sentencia `print`. Este código es más o menos lo mejor que podemos hacer, ya que cualquier programa correcto debe comprobar si cada variable es impar y comparar los valores de las variables impares para encontrar la mayor de ellas.

Python soporta **expresiones condicionales** así como **sentencias condicionales**. Las expresiones condicionales son de la forma:

```
expr1 if condition else expr2
```

Si la condición se evalúa como True, el valor de la expresión completa es `expr1`; en caso contrario, es `expr2`. Por ejemplo, la expresión:

```
x = y if y > z else z
```

Establece x como el máximo de `y` y `z`. Una expresión condicional puede aparecer en cualquier lugar donde pueda aparecer una expresión normal, incluso dentro de expresiones condicionales. Así, por ejemplo:

```
```

```
print((x if x > z else z) if x > y else (y if y > z else z))
```

imprime el máximo de `x`, `y` y `z`.

Una forma de pensar en la potencia de una clase de programas es en términos de cuánto tiempo pueden tardar en ejecutarse. Supongamos que cada línea de código tarda una unidad de tiempo en ejecutarse. Si un programa lineal tiene n líneas de código, tardará n unidades de tiempo en ejecutarse. ¿Y un programa ramificado con n líneas de código? Puede tardar menos de n unidades de tiempo en ejecutarse, pero no puede tardar más, ya que cada línea de código se ejecuta como máximo una vez.

Se dice que un programa cuyo tiempo máximo de ejecución está limitado por la longitud del programa se ejecuta en tiempo constante. Esto no significa que cada vez que se ejecuta el programa se ejecuta el mismo número de pasos. Significa que existe una constante, k, tal que se garantiza que el programa no tarda más de k pasos en ejecutarse. Esto implica que el tiempo de ejecución no crece con el tamaño de la entrada del programa.

Los programas de tiempo constante están limitados en lo que pueden hacer. Imaginemos un programa para contar los votos en unas elecciones. Sería realmente sorprendente si se pudiera escribir un programa que pudiera hacer esto en un tiempo que fuera independiente del número de votos emitidos. De hecho, es imposible hacerlo. El estudio de la dificultad intrínseca de los problemas es el tema de la *complejidad computacional*.

Cadenas de texto y entradas

Los objetos de tipo `str` se utilizan para representar caracteres. Los literales de tipo `str` pueden escribirse utilizando *comillas simples o dobles*, por ejemplo, `'abc'` o `"abc"`. El literal `'123'` denota una cadena de tres caracteres, no el número 123.

Intente escribir las siguientes expresiones en el intérprete de Python.

```
'a'
3 * 4
3 * 'a'
3 + 4
'a' + 'a'
```

Se dice que el operador `+` está sobrecargado porque tiene significados diferentes según los tipos de los objetos a los que se aplica. Por ejemplo, el operador `+` significa **suma** cuando se aplica a dos números y **concatenación** cuando se aplica a dos cadenas. El operador `*` también está sobrecargado. Significa lo que se espera que signifique cuando sus operandos son ambos números. Cuando se aplica a un `int` y a una cadena, es un operador de **repetición**: la expresión `n * s`, donde `n` es un `int` y `s` es una cadena, se evalúa como una cadena con *n repeticiones de s*. Por ejemplo, la expresión `2 * 'Juan'` tiene el valor `'JuanJuan'`. Esto tiene su lógica. Al igual que la expresión matemática `3 * 2` es equivalente a `2 + 2 + 2`, la expresión `3 * 'a'` es equivalente a `'a'+'a'+'a'`.

Ahora intenta:

```
new_id
'a' * 'a'
```

Cada una de estas líneas genera un mensaje de error. La primera línea produce el mensaje:

```
NameError: name 'new_id' is not defined
```

Dado que `new_id` no es un literal de ningún tipo, el intérprete lo trata como un nombre. Sin embargo, como ese nombre no está ligado a ningún objeto, al intentar utilizarlo se produce un error de ejecución. El código `'a' * 'a'` produce el mensaje de error:

```
TypeError: can't multiply sequence by non-int of type ‘str'
```

Que exista la comprobación de tipos es algo positivo. Convierte errores descuidados (y a veces sutiles) en errores que detienen la ejecución, en lugar de errores que llevan a los programas a comportarse de formas misteriosas. La comprobación de tipos en Python no es tan fuerte como en otros lenguajes de programación (por ejemplo, Java), pero es mejor en Python 3 que en Python 2.

Por ejemplo, está claro qué debe significar `<` cuando se usa para comparar dos cadenas o dos números. Pero, ¿Cuál debería ser el valor de `"4" < 3`? De forma bastante arbitraria, los diseñadores de Python 2 decidieron que debería ser False, porque todos los valores numéricos deberían ser menores que todos los valores de tipo `str`. Los diseñadores de Python 3, y de la mayoría de los lenguajes modernos, decidieron que como tales expresiones no tienen un significado obvio, deberían generar un *mensaje de error*.

Las cadenas son uno de los varios tipos de secuencias de Python. Comparten las siguientes operaciones con todos los tipos de secuencia.

- La **longitud** de una cadena puede calcularse con la función `len`. Por ejemplo, el valor de `len('abc')` es `3`.
- La **indexación** puede utilizarse para extraer caracteres individuales de una cadena. En Python, todos los índices se basan en cero. Por ejemplo, si se escribe `'abc'[0]` en el intérprete, éste mostrará la cadena `'a'`. Si se escribe `'abc'[3]`, aparecerá el mensaje de error `IndexError: índice de cadena fuera de rango`. Dado que Python utiliza *0 para indicar el primer elemento de una cadena*, se accede al último elemento de una cadena de longitud 3 utilizando el índice 2. Los números negativos se utilizan para indexar desde el final de una cadena. Por ejemplo, el valor de `'abc'[-1]` es `'c'`.
- El **slicing** (troceado) se utiliza para *extraer subcadenas de longitud arbitraria*. Si `s` es una cadena, la expresión `s[start:end]` denota la subcadena de `s` que comienza en el índice `start` y termina en el índice `end - 1`. Por ejemplo, `'abc'[1:3]` se evalúa como `'bc'`. ¿Por qué termina en el índice `end - 1` en lugar de `end`? Para que expresiones como `'abc'[0:len('abc')]` tengan el valor que cabría esperar. Si se omiten los dos puntos, el valor por defecto es 0. Si se omiten los dos puntos, el valor por defecto es la longitud de la cadena. En consecuencia, la expresión `'abc'[:]` es semánticamente equivalente a `'abc'[0:len('abc'))]`. También es posible proporcionar un tercer argumento para seleccionar una porción no contigua de una cadena. Por ejemplo, el valor de la expresión `'123456789'[0:8:2]` es la cadena `'1357'`.

A menudo es conveniente **convertir objetos de otros tipos en cadenas** utilizando la función `str()`. Consideremos, por ejemplo, el código:

```
num = 30000000
fraction = 1 / 2
print(num * fraction, 'is', fraction * 100, '%', 'of', num)
print(num * fraction, 'is', str(fraction * 100) + '%', 'of', num)
```

Que imprime:

```
15000000.0 is 50.0 % of 30000000
15000000.0 is 50.0 % of 30000000
```

La primera sentencia `print` inserta un espacio entre 50 y % porque Python inserta automáticamente un espacio entre los argumentos a imprimir. La segunda sentencia `print` produce una salida más apropiada combinando el 50 y el % en un único argumento de tipo `str`.

Las **conversiones de tipo** (también llamadas **type casts**) se utilizan a menudo en el código Python. Usamos el nombre de un tipo para convertir valores a ese tipo. Así, por ejemplo, el valor de `int('3') * 4` es 12. Cuando un flotante se convierte en un `int`, el número se **trunca** (no se redondea), por ejemplo, el valor de `int(3.9)` es el `int 3`.

Volviendo a la salida de nuestras sentencias `print`, puede que te estés preguntando por ese 0 al final del primer número impreso. Eso aparece porque `1 / 2` es un número de punto flotante, y el producto de un `int` y un `float` es un `float`. Puede evitarse convirtiendo `num * fracción` en un `int`. El código

```
print(int(num * fraction), 'is', str(fraction * 100) + '%', 'of', num)
```

imprime `15000000` es el `50.0%` de `30000000`.

Python 3.6 introdujo una forma alternativa, más compacta, de construir expresiones de cadena. Una **cadena f** consiste en el carácter `f` (o `F`) seguido de un **tipo especial de literal de cadena llamado literal de cadena formateado**. Los literales de cadena formateados contienen tanto secuencias de caracteres (como otros literales de cadena) como expresiones entre corchetes. *Estas expresiones se evalúan en tiempo de ejecución y se convierten automáticamente en cadenas*. El código

```
print(f'{int(num * fraction)} is {fraction * 100}% of {num}')
```

Produce la misma salida que la sentencia `print` anterior. Si desea incluir una llave en la cadena indicada por una cadena f, utilice dos llaves. Por ejemplo, `print(f'{{{3*5}}}')` imprime `{15}`.

La expresión dentro de una cadena f puede contener modificadores que controlan la apariencia de la cadena de salida. Estos modificadores se separan de la expresión que denota el valor a modificar mediante dos puntos. Por ejemplo, la cadena `f'{3.14159:.2f}'` se evalúa como la cadena `'3.14'` porque el modificador `.2f` indica a Python que trunque la representación de la cadena de un número de coma flotante a dos dígitos después del punto decimal. Y la sentencia

```
print(f'{num * fraction:,.0f} is {fraction * 100}% of {num:,}')
```

imprime `15,000,000 is 50.0% of 30,000,000` porque el modificador , indica a Python que use comas como separadores de miles.

Entradas

Python 3 tiene una función, `input`, que se puede utilizar para obtener información directamente de un usuario. La función `input` toma una cadena como argumento y la muestra como un prompt en el shell. La función espera a que el usuario escriba algo y pulse la tecla intro. *La línea tecleada por el usuario se trata como una cadena y se convierte en el valor devuelto por la función*.

Ejecutando el código `nombre = input('Escriba su nombre: ')` mostrará:

```
Enter your name:
```

en la ventana de la consola. Si a continuación escribe `George Washington` y pulsa intro, la cadena `'George Washington'` se asignará a la variable `nombre`. Si a continuación ejecutas `print('¿De verdad eres', nombre, '?')`, la línea:

```
¿De verdad eres George Washington ?
```

Observe que la sentencia `print` introduce un espacio antes del `"?"`. Esto se debe a que cuando `print` recibe múltiples argumentos, coloca un espacio entre los valores asociados a los argumentos. El espacio podría evitarse ejecutando `print('¿De verdad eres ' + nombre + '?')` o `print(f'¿De verdad eres {nombre}?')`, cada una de las cuales produce una única cadena y pasa esa cadena como único argumento a `print`.

Consideremos ahora el código:

```
n = input('Enter an int: ')
print(type(n))
```

Este código siempre imprimirá

```
<class 'str'>
```

porque input siempre devuelve un objeto de tipo `str`, incluso si el usuario ha introducido algo que parece un entero. Por ejemplo, si el usuario hubiera introducido 3, n estaría ligado a la cadena '3' y no al `int 3`. Así, el valor de la expresión `n * 4` sería `'3333'` en lugar de 12. La buena noticia es que siempre que una cadena sea un literal válido de algún tipo, se le puede aplicar una conversión de tipo.

Ejercicio manual

Escriba un código que pida al usuario que introduzca su fecha de nacimiento en el formato mm/dd/aaaa y, a continuación, imprima una

cadena del tipo «Usted nació en el año aaaa».

Digresión sobre la codificación de caracteres

Durante muchos años, la mayoría de los lenguajes de programación utilizaron un estándar llamado ASCII para la representación interna de caracteres. Este estándar incluía 128 caracteres, suficientes para representar el conjunto habitual de caracteres que aparecen en el texto en inglés, pero no suficientes para cubrir los caracteres y acentos que aparecen en todos los idiomas del mundo.

El estándar Unicode es un sistema de codificación de caracteres diseñado para facilitar el procesamiento digital y la visualización de textos escritos en todos los idiomas. El estándar *contiene más de 120.000 caracteres*, que abarcan 129 alfabetos modernos e históricos y múltiples conjuntos de símbolos.

El estándar Unicode puede implementarse utilizando diferentes codificaciones internas de caracteres. Puede indicar a Python qué codificación utilizar insertando un comentario de la forma:

```
# -*- coding: encoding name -*-
```

Como primera o segunda línea de su programa. Por ejemplo:

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

Indica a Python que utilice UTF-8, la codificación de caracteres más utilizada en las páginas web. Si no incluye este comentario en su programa, la mayoría de las implementaciones de Python utilizarán UTF-8 por defecto.

Cuando se usa UTF-8, puedes, si el editor de texto lo permite, introducir directamente código como:

```
print('Mluvíš anglicky?')
print('◌ा आप अंग्रेज़ी बोलते ह?')
```

Que imprimirá:

```
Mluvíš anglicky?
◌ा आप अंग्रेज़ी बोलते ह?
```

Te estarás preguntando cómo he conseguido escribir la cadena `'ा आप अैज़ी बोलते ह?'`. Como la mayor parte de la web utiliza UTF-8, pude copiar la cadena de una página web y pegarla directamente en mi programa. Hay formas de introducir directamente caracteres Unicode desde un teclado, pero a menos que tengas un teclado especial, todas son bastante engorrosas.

Ejercicio manual de la lectura

Asuma que se le da una variable llamada `number` (tiene un valor numérico). Escribe un fragmento de código Python que imprima una de las siguientes cadenas:

- `positive` si la variable `number` es positiva.
- `negative` si la variable `number` es negativa.
- `zero` si la variable `number` es cero.

Mi aproximación

```
if number > 0:
    print("positive")
elif number < 0:
    print("negative")
else:
    print("zero")
```

Utilizando los condicionales vistos en esta lectura puedo ramificar el código verificando si la variable `number` si esta es positiva, negativa o cero, e imprimiendo su correspondiente cadena.