TEMA 2: ESTRUCTURAS DE CONTROL

2.1.-Condicionales

2.1. Sentencias Condicionales

Las *sentencias condicionales* nos permiten comprobar si una o más condiciones son verdaderas o falsas, y ejecutar diferentes bloques de código en función de los resultados. En Python, las sentencias condicionales se realizan con las palabras clave `if`, `elif` y `else`.

Estas sentencias se repiten en muchos lenguajes de programación, pero la sintaxis puede variar ligeramente. A lo largo de esta unidad, veremos cómo funcionan las sentencias condicionales en Python y cómo podemos utilizarlas para controlar el flujo de ejecución de nuestros programas.

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 =>.

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 verdadero, 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
```

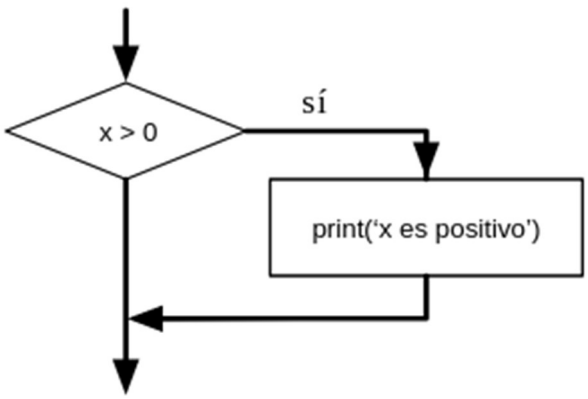
Esta flexibilidad puede ser útil, pero existen ciertas sutilezas en ese tipo de uso que pueden resultar confusas. Es posible que prefieras evitar usarlo de este modo hasta que estés bien seguro de lo que estás haciendo.

3. 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).



Lógica del IF

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.

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.

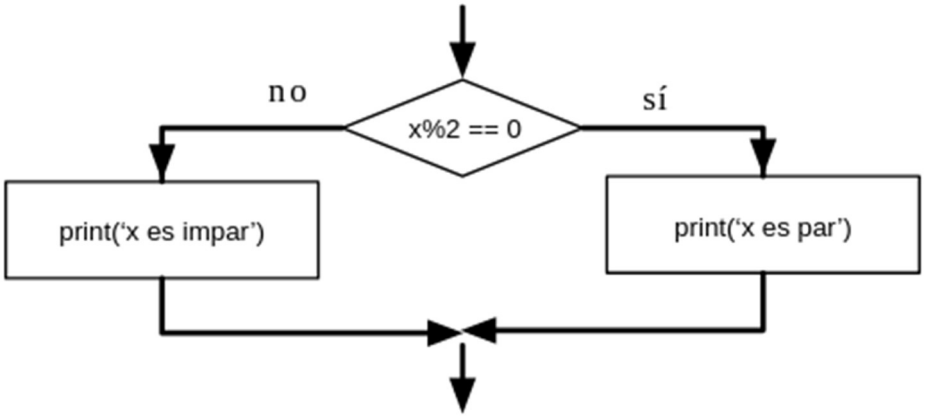
4. Ejecución alternativa

La segunda forma de la sentencia `if` es la *ejecución alternativa* , en la cual existen dos posibilidades y la condición determina cuál 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.



Lógica del IF-then-Else

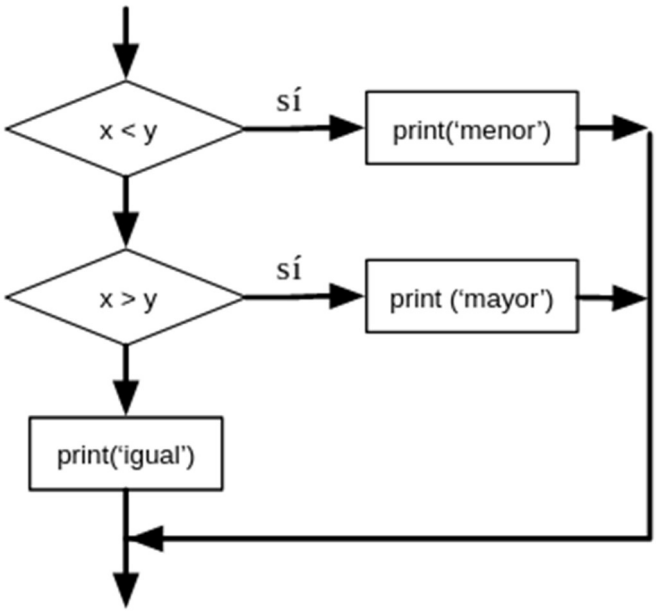
Dado que la condición debe ser obligatoriamente verdadera o falsa, solamente una de las alternativas será ejecutada. Las alternativas reciben el nombre de *ramas* , dado que se trata de ramificaciones en el flujo de la ejecución.

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.



Lógica del IF-then-Elif

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

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

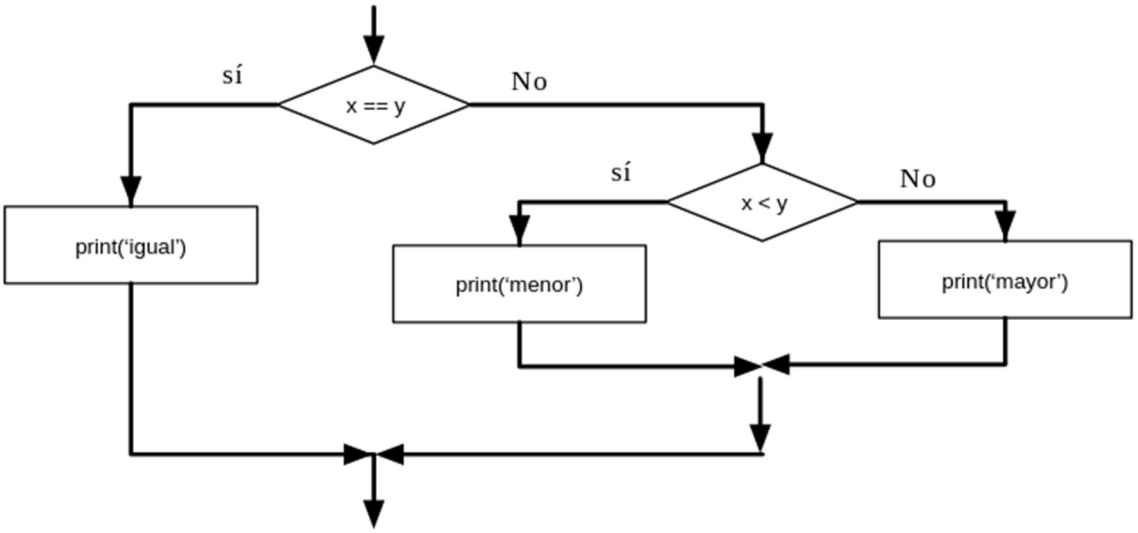
Cada condición es comprobada en orden. Si la primera es falsa, se comprueba la siguiente y así con las demás. Si una de ellas es verdadera, se ejecuta la rama correspondiente, y la sentencia termina. Incluso si hay más de una condición que sea verdadera, **sólo se ejecuta la primera que se encuentra**.

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.



Lógica del IF anidados

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.

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.')
```

7. 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.

8. 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 esa 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.

* Práctica 2.1: Sentencias condicionales

P2.1 - Ejercicios

Ejercicio 2.1.1

Escribir un programa que pregunte al usuario su edad y muestre por pantalla si es mayor de edad o no.

Ejercicio 2.1.2

Escribir un programa que almacene la cadena de caracteres `contraseña` en una variable, pregunte al usuario por la contraseña e imprima por pantalla si la contraseña introducida por el usuario coincide con la guardada en la variable sin tener en cuenta mayúsculas y minúsculas.

Ejercicio 2.1.3

Escribir un programa que pida al usuario dos números y muestre por pantalla su división. Si el divisor es cero el programa debe mostrar un error.

Ejercicio 2.1.4

Escribir un programa que pida al usuario un número entero y muestre por pantalla si es par o impar.

Ejercicio 2.1.5

Para tributar un determinado impuesto se debe ser mayor de 16 años y tener unos ingresos iguales o superiores a 1000 € mensuales. Escribir un programa que pregunte al usuario su edad y sus ingresos mensuales y muestre por pantalla si el usuario tiene que tributar o no.

Ejercicio 2.1.6

Los alumnos de un curso se han dividido en dos grupos A y B de acuerdo al sexo y el nombre. El grupo A esta formado por las mujeres con un nombre anterior a la M y los hombres con un nombre posterior a la N y el grupo B por el resto. Escribir un programa que pregunte al usuario su nombre y sexo, y muestre por pantalla el grupo que le corresponde.

Ejercicio 2.1.7

Los tramos impositivos para la declaración de la renta en un determinado país son los siguientes:

Renta	Tipo impositivo
Menos de 10000€	5%
Entre 10000€ y 20000€	15%
Entre 20000€ y 35000€	20%
Entre 35000€ y 60000€	30%
Más de 60000€	45%

Escribir un programa que pregunte al usuario su renta anual y muestre por pantalla el tipo impositivo que le corresponde.

Ejercicio 2.1.8

En una determinada empresa, sus empleados son evaluados al final de cada año. Los puntos que pueden obtener en la evaluación comienzan en 0.0 y pueden ir aumentando, traduciéndose en mejores beneficios. Los puntos que pueden conseguir los empleados pueden ser 0.0, 0.4, 0.6 o más, pero no valores intermedios entre las cifras mencionadas. A continuación, se muestra una tabla con los niveles correspondientes a cada puntuación. La cantidad de dinero conseguida en cada nivel es de 2.400€ multiplicada por la puntuación del nivel.

Nivel	Puntuación
Inaceptable	0.0
Aceptable	0.4
Meritorio	0.6 o más

Escribir un programa que lea la puntuación del usuario e indique su nivel de rendimiento, así como la cantidad de dinero que recibirá el usuario.

Ejercicio 2.1.9

Escribir un programa para una empresa que tiene salas de juegos para todas las edades y quiere calcular de forma automática el precio que debe cobrar a sus clientes por entrar. El programa debe preguntar al usuario la edad del cliente y mostrar el precio de la entrada. Si el cliente es menor de 4 años puede entrar gratis, si tiene entre 4 y 18 años debe pagar 5€ y si es mayor de 18 años, 10€.

Ejercicio 2.1.10

La pizzería Bella Napoli ofrece pizzas vegetarianas y no vegetarianas a sus clientes. Los ingredientes para cada tipo de pizza aparecen a continuación.

- Ingredientes vegetarianos: Pimiento y tofu.
- Ingredientes no vegetarianos: Pepperoni, Jamón y Salmón.

Escribir un programa que pregunte al usuario si quiere una pizza vegetariana o no, y en función de su respuesta le muestre un menú con los ingredientes disponibles para que elija. Solo se puede elegir un ingrediente además de la mozzarella y el tomate que están en todas las pizzas. Al final se debe mostrar por pantalla si la pizza elegida es vegetariana o no y todos los ingredientes que lleva.

2.2.-Iterativas y saltos

2.2. Sentencias Iterativas

Las sentencias iterativas o bucles permiten ejecutar un bloque de código varias veces. En Python, existen dos tipos de bucles: `for` y `while`. Estas sentencias nos las encontramos en la mayoría de los lenguajes de programación y son fundamentales para la programación. Por ejemplo, en otros lenguajes de programación, como C, C++, Java, Kotlin, etc., nos las encontramos, y además también se utilizan las sentencias `do-while`, pero en Python no existe esta sentencia.

A lo largo de esta unidad, veremos cómo se utilizan estas sentencias en Python, y cómo se pueden combinar con otras sentencias de control de flujo, como `break`, `continue`, `pass`, etc.

Antes de comenzar con las sentencias iterativas, vamos a recordar conceptos básicos de programación como son las sentencias de asignación de valores a las variables. Ya que, en muchos casos, los bucles se utilizan para realizar operaciones repetitivas sobre variables, y es importante la inicialización y actualización de las mismas.

1. Actualización de variables

Uno de los usos habituales de las sentencias de asignación consiste en realizar una actualización sobre una variable – en la cual el valor nuevo de esa variable depende del antiguo.

```
x = x + 1
```

Esto quiere decir “toma el valor actual de `x`, añádele 1, y luego actualiza `x` con el nuevo valor”.

Si intentas actualizar una variable que no existe, obtendrás un error, ya que Python evalúa el lado derecho antes de asignar el valor a `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Antes de que puedas actualizar una variable, debes *inicializarla*, normalmente mediante una simple asignación:

```
>>> x = 0
>>> x = x + 1
```

Actualizar una variable añadiéndole 1 se denomina *incrementar*; restarle 1 recibe el nombre de *decrementar* (o disminuir).

2. Bucles mediante la sentencia `while`

Los ordenadores, móviles, tables, etc se suelen utilizar a menudo para automatizar tareas repetitivas. Repetir tareas idénticas o muy similares sin cometer errores es algo que a las máquinas se les da bien y en cambio a las personas no. Como las iteraciones resultan tan habituales, Python y la mayoría de los lenguajes de programación proporcionan varias características para hacer más sencillas programar estas tareas..

Una forma de iteración en Python es la sentencia `while`. He aquí un programa sencillo que cuenta hacia atrás desde cinco y luego dice "¡Despegue!".

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('¡Despegue!')
```

Casi se puede leer la sentencia `while` como si estuviera escrita en inglés. Significa, "Mientras `n` sea mayor que 0, muestra el valor de `n` y luego reduce el valor de `n` en 1 unidad. Cuando llegues a 0, sal de la sentencia `while` y muestra la palabra `¡Despegue!`"

Éste es el flujo de ejecución de la sentencia `while`, explicado de un modo más formal:

1. Se evalúa la condición, obteniendo `Verdadero` or `Falso`.
2. Si la condición es falsa, se sale de la sentencia `while` y se continúa la ejecución en la siguiente sentencia.
3. Si la condición es verdadera, se ejecuta el cuerpo del `while` y luego se vuelve al paso 1.

Este tipo de flujo recibe el nombre de *bucle*, ya que el tercer paso enlaza de nuevo con el primero. Cada vez que se ejecuta el cuerpo del bucle se dice que realizamos una *iteración*. Para el bucle anterior, podríamos decir que "ha tenido cinco iteraciones", lo que significa que el cuerpo del bucle se ha ejecutado cinco veces.

El cuerpo del bucle debe cambiar el valor de una o más variables, de modo que la condición pueda en algún momento evaluarse como falsa y el bucle termine. La variable que cambia cada vez que el bucle se ejecuta y controla cuándo termina éste, recibe el nombre de *variable de iteración*. Si no hay variable de iteración, el bucle se repetirá para siempre, resultando así un *bucle infinito*.

2.1. Bucles infinitos

Una fuente de diversión sin fin para los programadores es la constatación de que las instrucciones del champú: "Enjabone, aclare, repita", son un bucle infinito, ya que no hay una *variable de iteración* que diga cuántas veces debe ejecutarse el proceso.

En el caso de una `cuenta atrás`, podemos verificar que el bucle termina, ya que sabemos que el valor de `n` es finito, y podemos ver que ese valor se va haciendo más pequeño cada vez que se repite el bucle, de modo que en algún momento llegará a 0. Otras veces un bucle es obviamente infinito, porque no tiene ninguna variable de iteración.

2.2.1. "Bucles infinitos" y `break`

A veces no se sabe si hay que terminar un bucle hasta que se ha recorrido la mitad del cuerpo del mismo. En ese caso se puede crear un bucle infinito a propósito y usar la sentencia `break` para salir fuera de él cuando se desee, aunque no es aconsejable puesto que podemos poner la condición de salida en la evaluación del `while`.

El bucle siguiente es, obviamente, un *bucle infinito*, porque la expresión lógica de la sentencia `while` es simplemente la constante lógica `True (verdadero)`;

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('¡Terminado!')
```

Si cometes el error de ejecutar este código, aprenderás rápidamente cómo detener un proceso de Python bloqueado en el sistema, o tendrás que localizar dónde se encuentra el botón de apagado de tu equipo. Este programa funcionará para siempre, o hasta que la batería del equipo se termine, ya que la expresión lógica al principio del bucle es siempre cierta, en virtud del hecho de que esa expresión es precisamente el valor constante `True`.

A pesar de que en este caso se trata de un bucle infinito inútil, se puede usar ese diseño para construir bucles útiles, siempre que se tenga la precaución de añadir código en el cuerpo del bucle para salir explícitamente, usando `break` cuando se haya alcanzado la condición de salida.

Por ejemplo, supón que quieres recoger entradas de texto del usuario hasta que éste escriba `fin`. Podrías escribir:

```
while True:
    linea = input('> ')
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Código: https://es.py4e.com/code3/copytildone1.py
```

La condición del bucle es `True`, lo cual es verdadero siempre, así que el bucle se repetirá hasta que se ejecute la sentencia break.

Cada vez que se entre en el bucle, se pedirá una entrada al usuario. Si el usuario escribe `fin`, la sentencia `break` hará que se salga del bucle. En cualquier otro caso, el programa repetirá cualquier cosa que el usuario escriba y volverá al principio del bucle. Éste es un ejemplo de su funcionamiento:

```
> hola a todos
hola a todos
> he terminado
he terminado
> fin
```

```
¡Terminado!
```

Es importante tener en cuenta que la sentencia `break` sólo afecta al bucle más interno en el que se encuentra. Si se utiliza en un bucle anidado, sólo se saldrá del bucle en el que se encuentra. El bucle exterior continuará ejecutándose.

En este ciclo vamos a evitar el uso del break, ya que no es una buena práctica, y vamos a utilizar una condición en la evaluación del `while` para salir del bucle. De esta forma nos acostumbramos a utilizar la estructura de control de flujo de forma correcta. En un futuro, cuando tu crezcas como programador, tendrás tu propio criterio para utilizar o no utilizar el `break`.

¿Cómo lo harías sin usar `while True:` y `break`?

2.2. Finalizar iteraciones con `continue`

Algunas veces, estando dentro de un bucle se necesita terminar con la iteración actual y saltar a la siguiente de forma inmediata. En ese caso se puede utilizar la sentencia `continue` para pasar a la siguiente iteración sin terminar la ejecución del cuerpo del bucle para la actual.

A continuación, se muestra un ejemplo de un bucle que repite lo que recibe como entrada hasta que el usuario escribe “fin”, pero trata las líneas que empiezan por el carácter almohadilla como líneas que no deben mostrarse en pantalla (algo parecido a lo que hace Python con los comentarios).

```
while True:
    linea = input('> ')
    if linea[0] == '#' :
        continue
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Código: https://es.py4e.com/code3/copytildone2.py
```

He aquí una ejecución de ejemplo de ese nuevo programa con la sentencia `continue` añadida.

```
> hola a todos
hola a todos
> # no imprimas esto
> ¡imprime esto!
¡imprime esto!
> fin
¡Terminado!
```

Todas las líneas se imprimen en pantalla, excepto la que comienza con el símbolo de almohadilla, ya que en ese caso se ejecuta `continue`, finaliza la iteración actual y salta de vuelta a la sentencia `while` para comenzar la siguiente iteración, de modo que que se omite la sentencia `print`.

Al igual que hemos dicho con el break, en este ciclo vamos a evitar el uso del `continue`, ya que no es una buena práctica, y vamos a utilizar una condición en la evaluación de que sentencias queremos ejecutar. De esta forma nos acostumbramos a utilizar la estructura de control de flujo de forma correcta. En un futuro, cuando tu crezcas como programador, tendrás tu propio criterio para utilizar o no utilizar el `continue`.

¿Cómo lo harías sin usar `while True:`, `break`, `continue`?

3. Bucles definidos usando `for`

A veces se desea repetir un bucle a través de un *conjunto* de cosas, como una lista de palabras, las líneas de un archivo, o una lista de números. Cuando se tiene una lista de cosas para recorrer, se puede construir un bucle *definido* usando una sentencia `for`. A la sentencia `while` se la llama un bucle *indefinido*, porque simplemente se repite hasta que cierta condición se hace `Falsa`, mientras que el bucle `for` se repite a través de un conjunto conocido de elementos, de modo que ejecuta tantas iteraciones como elementos hay en el conjunto.

La sintaxis de un bucle `for` es similar a la del bucle `while`, en ella hay una sentencia `for` y un cuerpo que se repite:

```
amigos = ['Joseph', 'Glenn', 'Sally']
for amigo in amigos:
    print('Feliz año nuevo:', amigo)
print('¡Terminado!')
```

En términos de Python, la variable `amigos` es una lista[1] de tres cadenas y el bucle `for` se mueve recorriendo la lista y ejecuta su cuerpo una vez para cada una de las tres cadenas en la lista, produciendo esta salida:

```
Feliz año nuevo: Joseph
Feliz año nuevo: Glenn
Feliz año nuevo: Sally
¡Terminado!
```

La traducción de este bucle `for` al español no es tan directa como en el caso del `while`, pero si piensas en los amigos como un *conjunto*, sería algo así como: “Ejecuta las sentencias en el cuerpo del bucle una vez *para (for)* cada amigo que esté *en (in)* el conjunto llamado amigos.”

Revisando el bucle `for`, *for* e *in* son palabras reservadas de Python, mientras que `amigo` y `amigos` son variables.

```
for amigo in amigos:
    print('Feliz año nuevo::', amigo)
```

En concreto, `amigo` es la *variable de iteración* para el bucle `for`. La variable `amigo` cambia para cada iteración del bucle y controla cuándo se termina el bucle `for`. La *variable de iteracion* se desplaza sucesivamente a través de las tres cadenas almacenadas en la variable `amigos`.

Otra forma de usar los bucles `for` es haciendo uso de la instrucción `range`:

- `range(fin)` : Genera una secuencia de números enteros desde 0 hasta `fin-1`.
- `range(inicio, fin, salto)` : Genera una secuencia de números enteros desde `inicio` hasta `fin-1` con un incremento de `salto`.

```
>>> for i in range(1, 10, 2):
...     print(i, end=", ")
...
1, 3, 5, 7, 9, >>>
```

4. Diseños de bucles

A menudo se usa un bucle `for` o `while` para movernos a través de una lista de elementos o el contenido de un archivo y se busca algo, como el valor más grande o el más pequeño de los datos que estamos revisando.

Los bucles generalmente se construyen así:

- Se inicializan una o más variables antes de que el bucle comience
- Se realiza alguna operación con cada elemento en el cuerpo del bucle, posiblemente cambiando las variables dentro de ese cuerpo.
- Se revisan las variables resultantes cuando el bucle se completa

Usaremos ahora una lista de números para demostrar los conceptos y construcción de estos diseños de bucles.

4.1. Bucles de recuento y suma

Por ejemplo, para contar el número de elementos en una lista, podemos escribir el siguiente bucle `for`:

```
contador = 0
for valor in [3, 41, 12, 9, 74, 15]:
    contador = contador + 1
print('Num. elementos: ', contador)
```

Ajustamos la variable `contador` a cero antes de que el bucle comience, después escribimos un bucle `for` para movernos a través de la lista de números. Nuestra *variable de iteración* se llama `valor`, y dado que no usamos `valor` dentro del bucle, lo único que hace es controlar el bucle y hacer que el cuerpo del mismo sea ejecutado una vez para cada uno de los valores de la lista.

En el cuerpo del bucle, añadimos 1 al valor actual de `contador` para cada uno de los valores de la lista. Mientras el bucle se está ejecutando, el valor de `contador` es la cantidad de valores que se hayan visto "hasta ese momento".

Una vez el bucle se completa, el valor de `contador` es el número total de elementos. El número total "cae en nuestro poder" al final del bucle. Se construye el bucle de modo que obtengamos lo que queremos cuando éste termina.

Otro bucle similar, que calcula el total de un conjunto de números, se muestra a continuación:

```
total = 0
for valor in [3, 41, 12, 9, 74, 15]:
    total = total + valor
print('Total: ', total)
```

En este bucle, *sí* utilizamos la *variable de iteración*. En vez de añadir simplemente uno a `contador` como en el bucle previo, ahora durante cada iteración del bucle añadimos el número actual (3, 41, 12, etc.) al total en ese momento. Si piensas en la variable `total`, ésta contiene la "suma parcial de valores hasta ese momento". Así que antes de que el bucle comience, `total` es cero, porque aún no se ha examinado ningún valor. Durante el bucle, `total` es la suma parcial, y al final del bucle, `total` es la suma total definitiva de todos los valores de la lista.

Cuando el bucle se ejecuta, `total` acumula la suma de los elementos; una variable que se usa de este modo recibe a veces el nombre de *acumulador*.

Ni el bucle que cuenta los elementos ni el que los suma resultan particularmente útiles en la práctica, dado que existen las funciones internas `len()` y `sum()` que cuentan el número de elementos de una lista y el total de elementos en la misma respectivamente.

4.2. Bucles de máximos y mínimos

Para encontrar el valor mayor de una lista o secuencia, construimos el bucle siguiente:

```
mayor = None
print('Antes:', mayor)
for valor in [3, 41, 12, 9, 74, 15]:
    if mayor is None or valor > mayor :
        mayor = valor
    print('Bucle:', valor, mayor)
print('Mayor:', mayor)
```

Cuando se ejecuta el programa, se obtiene la siguiente salida:

```
Antes: None
Bucle: 3 3
Bucle: 41 41
Bucle: 12 41
Bucle: 9 41
Bucle: 74 74
Bucle: 15 74
Mayor: 74
```

Debemos pensar en la variable `mayor` como el "mayor valor visto hasta ese momento". Antes del bucle, asignamos a `mayor` el valor `None`. `None` es un valor constante especial que se puede almacenar en una variable para indicar que la variable está "vacía".

Antes de que el bucle comience, el mayor valor visto hasta entonces es `None`, dado que no se ha visto aún ningún valor. Durante la ejecución del bucle, si `mayor` es `None`, entonces tomamos el primer valor que tenemos como el mayor hasta entonces. Se puede

ver en la primera iteración, cuando el valor de `valor` es 3, mientras que `mayor` es `None`, inmediatamente hacemos que `mayor` pase a ser 3.

Tras la primera iteración, `mayor` ya no es `None`, así que la segunda parte de la expresión lógica compuesta que comprueba si `valor > mayor` se activará sólo cuando encontremos un valor que sea mayor que el "mayor hasta ese momento". Cuando encontramos un nuevo valor "mayor aún", tomamos ese nuevo valor para `mayor`. Se puede ver en la salida del programa que `mayor` pasa desde 3 a 41 y luego a 74.

Al final del bucle, se habrán revisado todos los valores y la variable `mayor` contendrá entonces el mayor valor de la lista.

Para calcular el número más pequeño, el código es muy similar con un pequeño cambio:

```
print('Antes:', menor)
for valor in [3, 41, 12, 9, 74, 15]:
    if menor is None or valor < menor:
        menor = valor
    print('Bucle:', valor, menor)
print('Menor:', menor)
```

De nuevo, `menor` es el "menor hasta ese momento" antes, durante y después de que el bucle se ejecute. Cuando el bucle se ha completado, `menor` contendrá el valor mínimo de la lista

También como en el caso del número de elementos y de la suma, las funciones internas `max()` y `min()` convierten la escritura de este tipo de bucles en innecesaria.

Lo siguiente es una versión simple de la función interna de Python `min()`:

```
def min(valores):
    menor = None
    for valor in valores:
        if menor is None or valor < menor:
            menor = valor
    return menor
```

En esta versión de la función para calcular el mínimo, hemos eliminado las sentencias `print`, de modo que sea equivalente a la función `min`, que ya está incorporada dentro de Python.

5. Depuración

A medida que vayas escribiendo programas más grandes, puede que notes que vas necesitando emplear cada vez más tiempo en depurarlos. Más código significa más oportunidades de cometer un error y más lugares donde los **bugs** pueden esconderse.

Un método para acortar el tiempo de depuración es "depurar por bisección". Por ejemplo, si hay 100 líneas en tu programa y las compruebas de una en una, te llevará 100 pasos.

En lugar de eso, intenta partir el problema por la mitad. Busca en medio del programa, o cerca de ahí, un valor intermedio que puedas comprobar. Añade una sentencia `print` (o alguna otra cosa que tenga un efecto verificable), y haz funcionar el programa.

Si en el punto medio la verificación es incorrecta, el problema debería estar en la primera mitad del programa. Si ésta es correcta, el problema estará en la segunda mitad.

Cada vez que realices una comprobación como esta, reduces a la mitad el número de líneas en las que buscar. Después de seis pasos (que son muchos menos de 100), lo habrás reducido a una o dos líneas de código, al menos en teoría.

En la práctica no siempre está claro qué es “en medio del programa”, y no siempre es posible colocar ahí una verificación. No tiene sentido contar las líneas y encontrar el punto medio exacto. En lugar de eso, piensa en lugares del programa en los cuales pueda haber errores y en lugares donde resulte fácil colocar una comprobación. Luego elige un sitio donde estimes que las oportunidades de que el bug esté por delante y las de que esté por detrás de esa comprobación son más o menos las mismas.

Actividades

Actividad 1: Escribe un programa que lea repetidamente números hasta que el usuario introduzca "fin". Una vez se haya introducido "fin", muestra por pantalla el total, la cantidad de números y la media de esos números. Si el usuario introduce cualquier otra cosa que no sea un número, (mas adelante veremos como detectar los fallos usando `try` y `except`)

```
Introduzca un número: 4
Introduzca un número: 5
Introduzca un número: dato erróneo
Entrada inválida
Introduzca un número: 7
Introduzca un número: fin
16 3 5.33333333333
```

Actividad 2: Escribe otro programa que pida una lista de números como la anterior y al final muestre por pantalla el máximo y mínimo de los números, en vez de la media.

Fuente

- Pagina de Juan Jose Lozano Gomez sobre Python
- Aprende con Alf
- Python para todos

* Práctica 2.2: Sentencias iterativas y saltos

Durante la realización de estos ejercicios, no debes usar ninguna función (método) de las clases para ayudarte a realizarlo. Es decir, evita hacer uso de las funciones `len`, `count` de `str`, etc. La realización de estas funciones forman parte de la realización del ejercicio.

P2.2 - Ejercicios

Ejercicio 2.2.1

Escribir un programa que pida al usuario una palabra y la muestre por pantalla 10 veces.

Ejercicio 2.2.2

Escribir un programa que pregunte al usuario su edad y muestre por pantalla todos los años que ha cumplido (desde 1 hasta su edad).

Ejercicio 2.2.3

Escribir un programa que pida al usuario un número entero positivo y muestre por pantalla todos los números impares desde 1 hasta ese número separados por comas.

Ejercicio 2.2.4

Escribir un programa que pida al usuario un número entero positivo y muestre por pantalla la cuenta atrás desde ese número hasta cero separados por comas.

Ejercicio 2.2.5

Escribir un programa que pregunte al usuario una cantidad a invertir, el interés anual y el número de años, y muestre por pantalla el capital obtenido en la inversión cada año que dura la inversión.

```
# Formula para calcular El capital tras un año.
amount *= 1 + interest / 100
# En donde:
# - amount: Cantidad a invertir
# - interest: Interes porcentual anual
```

Ejercicio 2.2.6

Escribir un programa que pida al usuario un número entero y muestre por pantalla un triángulo rectángulo como el de más abajo, de altura el número introducido.

```
*
**
***
****
```

*****

Ejercicio 2.2.7

Escribir un programa que muestre por pantalla la tabla de multiplicar del 1 al 10.

Ejercicio 2.2.8

Escribir un programa que pida al usuario un número entero y muestre por pantalla un triángulo rectángulo como el de más abajo.

```
1
3 1
5 3 1
7 5 3 1
9 7 5 3 1
```

Ejercicio 2.2.9

Escribir un programa que almacene la cadena de caracteres contraseña en una variable, pregunte al usuario por la contraseña hasta que introduzca la contraseña correcta.

Ejercicio 2.2.10

Escribir un programa que pida al usuario un número entero y muestre por pantalla si es un número primo o no.

Ejercicio 2.2.11

Escribir un programa que pida al usuario una palabra y luego muestre por pantalla una a una las letras de la palabra introducida empezando por la última.

Ejercicio 2.2.12

Escribir un programa en el que se pregunte al usuario por una frase y una letra, y muestre por pantalla el número de veces que aparece la letra en la frase.

Ejercicio 2.2.13

Escribir un programa que muestre el eco de todo lo que el usuario introduzca hasta que el usuario escriba “salir” que terminará.

Ejercicio 2.2.14

Leer números enteros de teclado, hasta que el usuario ingrese el 0. Finalmente, mostrar la sumatoria de todos los números ingresados.

Ejercicio 2.2.15

Leer números enteros de teclado, hasta que el usuario ingrese el 0. Finalmente, mostrar la sumatoria de todos los números positivos ingresados.

Ejercicio 2.2.16

Leer números enteros positivos de teclado, hasta que el usuario ingrese el 0. Informar cuál fue el mayor número ingresado.

Ejercicio 2.2.17

Leer un número entero positivo desde teclado e imprimir la suma de los dígitos que lo componen.

Ejercicio 2.2.18

Solicitar al usuario que ingrese números enteros positivos y, por cada uno, imprimir la suma de los dígitos que lo componen. La condición de corte es que se ingrese el número -1. Al finalizar, mostrar cuántos de los números ingresados por el usuario fueron números pares.

Ejercicio 2.2.19

Mostrar un menú con tres opciones: 1- comenzar programa, 2- imprimir listado, 3- finalizar programa. A continuación, el usuario debe poder seleccionar una opción (1, 2 ó 3). Si elige una opción incorrecta, informarle del error. El menú se debe volver a mostrar luego de ejecutada cada opción, permitiendo volver a elegir. Si elige las opciones 1 ó 2 se imprimirá un texto. Si elige la opción 3, se interrumpirá la impresión del menú y el programa finalizará.

Ejercicio 2.2.20

Solicitar al usuario el ingreso de una frase y de una letra (que puede o no estar en la frase). Recorrer la frase, carácter a carácter, comparando con la letra buscada. Si el carácter no coincide, indicar que no hay coincidencia en esa posición (imprimiendo la posición) y continuar. Si se encuentra una coincidencia, indicar en qué posición se encontró y finalizar la ejecución.

Ejercicio 2.2.21

Crear un programa que permita al usuario ingresar los montos de las compras de un cliente (se desconoce la cantidad de datos que cargará, la cual puede cambiar en cada ejecución), cortando el ingreso de datos cuando el usuario ingrese el monto 0. Si ingresa un monto negativo, no se debe procesar y se debe pedir que ingrese un nuevo monto. Al finalizar, informar el total a pagar teniendo que cuenta que, si las ventas superan el total de $1000, se le debe aplicar un 10% de descuento.

Ejercicio 2.2.22

Crear un programa que solicite el ingreso de números enteros positivos, hasta que el usuario ingrese el 0. Por cada número, informar cuántos dígitos pares y cuántos impares tiene. Al finalizar, informar la cantidad de dígitos pares y de dígitos impares leídos en total.

Ejercicio 2.2.23

Crear un programa que permita al usuario ingresar títulos de libros por teclado, finalizando el ingreso al leerse el string "*" (asterisco). Cada vez que el usuario ingrese un string de longitud 1 que contenga sólo una barra ("/") se considera que termina una línea. Por cada línea completa, informar cuántos dígitos numéricos (del 0 al 9) aparecieron en total (en todos los títulos de libros que componen en esa línea). Finalmente, informar cuántas líneas completas se ingresaron.

```
Ejemplo de ejecución:
Libro: Los 3 mosqueteros
Libro: Historia de 2 ciudades
Libro: /
Línea completa. Aparecen 2 dígitos numéricos.
Libro: 20000 leguas de viaje submarino
Libro: El señor de los anillos
Libro: /
Línea completa. Aparecen 5 dígitos numéricos.
Libro: 20 años después
Libro: *
Fin. Se leyeron 2 líneas completas.
```

Ejercicio 2.2.24

Escribir un programa que solicite el ingreso de una cantidad indeterminada de números mayores que 1, finalizando cuando se reciba un cero. Imprimir la cantidad de números primos ingresados.

Ejercicio 2.2.25

Solicitar al usuario que ingrese una frase y luego informar cuál fue la palabra más larga (en caso de haber más de una, mostrar la primera) y cuántas palabras había.
Precondición: se tomará como separador de palabras al carácter " " (espacio), ya sea uno o más.