INGENIERÍA TÉCNICA EN DISEÑO INDUSTRIAL
509: INFORMÁTICA BÁSICA
Tema 6: Introducción a la programación. El lenguaje Python
Parte III (curso 06/07)

Índice

25. Sentencia de repetición `while` **55**
25.1. Traza de un programa mediante PythonG 56
25.2. Algunos programas sencillos 61
25.3. Mejorando la lectura de listas 66

26. Sentencia de repetición `for` **67**
26.1. La función interna `range` 70
26.2. Bucles equivalentes con `for` y `while` 72

27. Sentencias de repetición anidadas **76**

28. Sesión de problemas 5 **79**

Bibliografía

Libro de apuntes de la asignatura *Metodología y Tecnología de la Programación* (II04 e IG04). Temas 1, 2, 3, 4 y 5. Servicio de Reprografía de la E.S.T.C.E. y `http://marmota.act.uji.es/IG04/pdf/python.pdf`

Estos apuntes para la asignatura *Informática Básica* (509) se han basado en parte en los de *Metodología y Tecnología de la Programación*, realizados por los profesores Andrés Marzal e Isabel Gracia.

M. LUTZ Y D. ASCHER: *Learning Python*. O'Reilly & Associates, 1999.

25. Sentencia de repetición `while`

Muchas veces se requiere repetir una secuencia de acciones en un programa. Por ejemplo, la lectura de una lista de 4 números enteros hasta ahora la hemos podido realizar mediante una secuencia de instrucciones como la siguiente:

```
milista = [0]*4
milista[0] = int(raw_input("Dime el número 0: "))
milista[1] = int(raw_input("Dime el número 1: "))
milista[2] = int(raw_input("Dime el número 2: "))
milista[3] = int(raw_input("Dime el número 3: "))
```

Fíjate que con la primera instrucción se crea una lista de 4 elementos, cada uno de los cuales será `0`, y se asigna a la variable `milista`. Y a continuación, se realizan cuatro instrucciones "casi idénticas", en cada una de las cuales se lee un número entero del teclado y se asigna a una posición de la lista. La única diferencia entre estas cuatro instrucciones es la posición concreta a la que se asigna el número leído, como se ve por el distinto índice empleado para acceder a la correspondiente posición en cada una de ellas. En esencia, esta secuencia de cuatro instrucciones es la repetición 4 veces de una misma acción. Si deseasemos leer una lista de 100 elementos, ¿tendríamos que escribir explícitamente 100 instrucciones "casi idénticas" en el programa?

Las *sentencias de repetición* se utilizan para gestionar las condiciones bajo las cuales se repiten otras instrucciones. De esta manera, para repetir la ejecución de una secuencia de instrucciones no va a ser necesario escribir repetidamente las instrucciones en el programa, sino situar las instrucciones que deben repetirse "bajo el control" de una sentencia de repetición.

Vamos a estudiar las dos sentencias de repetición disponibles en Python: primero la sentencia `while`, que es más general, y después la sentencia `for`.

La sentencia `while` es una sentencia de repetición con condición inicial. Es decir, la decisión de repetir una vez más o detener la repetición de una secuencia de instrucciones dependerá de una condición que se sitúa al comienzo de la sentencia de repetición. La sentencia de repetición `while` de Python tiene la siguiente forma:

```
while condición:
    instrucciones
```

La *`condición`* será una expresión cuyo resultado será un valor lógico: *cierto* o *falso*. De esta manera, el funcionamiento general de la sentencia es:

1.- Cada vez que se ejecuta la línea `while` *`condición`*`:`, se evalúa la *`condición`* y se obtiene su resultado: *cierto* o *falso*.

2.- Si el resultado de la *`condición`* es *cierto*, *se ejecutarán* las *`instrucciones`* (por primera vez u otra vez más), y al finalizarlas *volverá a la línea* `while` *`condición`*`:` (o sea, se vuelve al paso 1).

Si el resultado de la *`condición`* es *falso*, *no se ejecutarán* las *`instrucciones`* (termina la repetición), y la ejecución proseguirá con la sentencia que se haya escrito después de esta sentencia de repetición.

En otras palabras, las *`instrucciones`* se ejecutan *mientras* se cumple la *`condición`* ("while" en inglés significa mientras), y dejan de ejecutarse cuando ya no se cumple.

A una sentencia de repetición también se le denomina *bucle*, y a las *`instrucciones`* que se repiten en ella, *cuerpo del bucle*. Una *iteración* denota una repetición (una de las posibles ejecuciones) del cuerpo del bucle.

Ejemplo: Vamos a escribir un programa en Python que utilice una sentencia `while` para calcular:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + 5 + \cdots + n-1 + n$$

leyendo el valor de n desde el teclado.

Fíjate que si el valor de n se lee del teclado (al ejecutar el programa) no podemos desarrollar un programa sólo con las sentencias que conocíamos hasta ahora, ya que el número de sumas a realizar dependerá del valor de n, y con las sentencias que conocíamos, deberíamos saber cuántas sumas se van a hacer al escribir el programa (antes de ejecutarlo).

Si nos planteásemos realizar a mano el sumatorio de todos los números naturales hasta uno dado n, podríamos comenzar estableciendo un valor de "suma parcial" inicialmente a 0, y sucesivamente

ir acumulando cada nuevo número a esta suma parcial. Así, primero acumularíamos el 1 a la suma parcial, después el 2, ..., hasta llegar a n. Es decir, iríamos obteniendo sumas parciales como si las sumas estuviesen parentizadas de la siguiente manera:

$$((\cdots(((((0+1)+2)+3)+4)+5)+\cdots+n-1)+n)$$

Vamos a aplicar esta misma idea en el programa. Empleamos una variable `suma_hasta_i` que inicialmente tendrá el valor 0, y en todo momento representará la suma parcial realizada con todos los números anteriores o iguales a uno dado, que vendrá indicado por la variable `i`.

De esta manera, dándole a `i` inicialmente el valor 1, mediante una sentencia de repetición se realiza la acumulación sucesiva, ejecutando dos acciones en cada iteración:

- acumular en `suma_hasta_i` el valor de `i` en esa iteración, e
- incrementar el valor de `i` para la siguiente iteración (si ésta se produce).

La acumulación sucesiva de valores de `i` debe acabar cuando se haya sumado el valor leído `n`, por eso estas dos acciones se repetirán mientras `i` sea menor o igual `n`. El programa que implementa la estrategia descrita es el siguiente:

```
n = int(raw_input("Dime el valor de n: "))
suma_hasta_i = 0
i = 1
while i <= n:
    suma_hasta_i = suma_hasta_i + i
    i = i + 1
print "Sumatorio =", suma_hasta_i
```

Fíjate en tres aspectos importantes de la sentencia `while` del ejemplo, que coinciden con los que destacamos para la sentencia `if`.

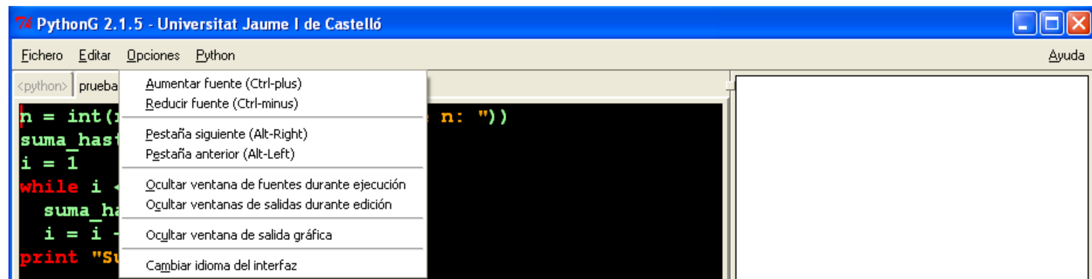
- A continuación de la palabra `while` se escribe la *`condición`* que permite decidir la continuación o finalización del bucle. En este ejemplo, hemos escrito la comparación `i <= n`, pero la condición de un `while` podrá ser cualquier expresión lógica.
- La línea acaba con dos puntos `:` escritos tras la condición. Al igual que en la sentencia `if`, siempre hay que finalizar la línea de `while` con dos puntos.
- Las dos líneas siguientes a `while` corresponden a lo que en el esquema hemos llamado *`instrucciones`* y ambas aparecen indentadas respecto a la línea de `while`. Recuerda que la indentación en Python determina la dependencia de unas líneas respecto a otras. Así, estas dos líneas dependen de la sentencia `while`, con lo que son las *`instrucciones`* que se repetirán (el cuerpo del bucle) mientras la *`condición`* se cumpla (mientras que el valor de `i` sea menor o igual que el valor de `n`).

25.1. Traza de un programa mediante PythonG

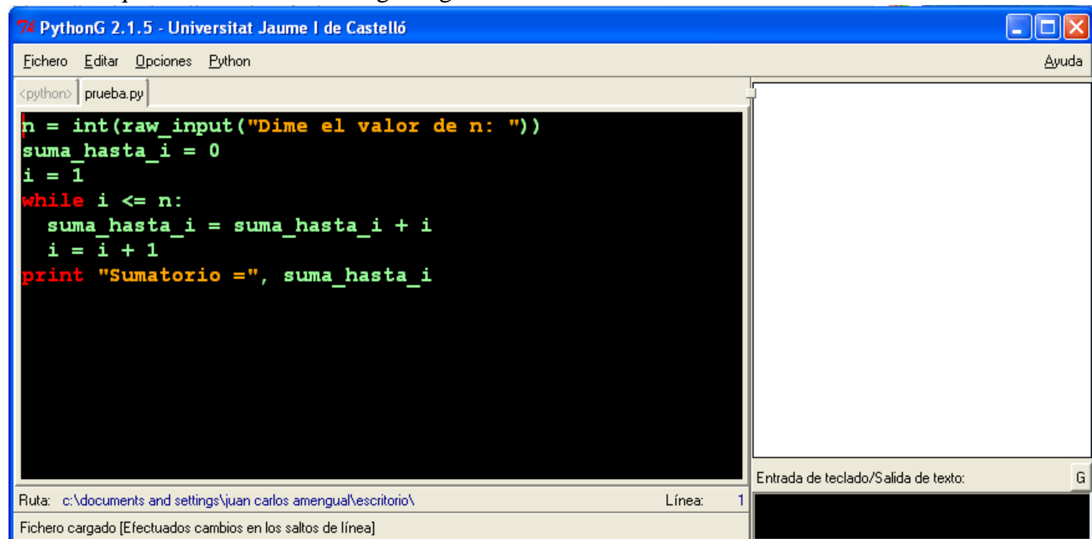
Vamos a realizar una traza de este programa para comprobar su funcionamiento. Una *traza* es un seguimiento detallado de las sucesivas acciones que va realizando el programa al ejecutarse con unos datos de entrada concretos. Pero en lugar de realizar nosotros la traza, vamos a dejar que sea PythonG quien la haga. Es decir, vamos a utilizar la herramienta de depuración de PythonG para observar lo que va realizando el intérprete de Python paso a paso al ejecutar el programa. En principio, el propósito de una herramienta de depuración es servir de ayuda para detectar errores en los programas, proporcionando medios para observar internamente su ejecución. No obstante, también podemos aprovechar estos medios con programas que no contienen errores, para simplemente conocer los detalles internos de su ejecución.

Mediante la opción **Python ▷ Activar modo depuración** ejecutaremos el programa que se muestra en la pestaña activa de edición en "*modo depuración*". Así, podremos ejecutar cada instrucción del programa paso a paso, observando los valores que van almacenando sucesivamente las variables que se crean/usan en él.

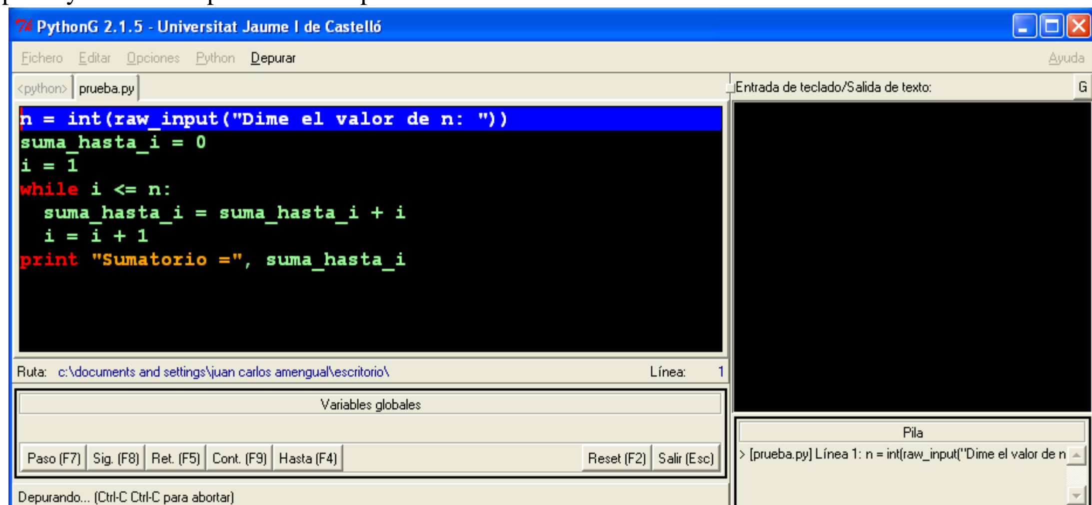
Pero antes, *resulta necesario que desmarquemos la opción* **Ocultar ventana de fuentes durante ejecución** que hay en el menú **Opciones**, tal como muestra la figura siguiente. Observa que en la ventana se muestra cómo debe aparecer el menú **Opciones** al desplegarlo:



Bien. Supón, pues, que tenemos el programa anterior cargado en la pestaña activa de edición. Nuestra situación de partida es la que se muestra en la imagen siguiente:



Ahora vamos a proceder a ejecutar el programa en "modo depuración". Para ello, seleccionamos **Python** ▷ **Activar modo depuración**. Entonces la ventana de PythonG ofrece el aspecto que se muestra a continuación. Importante: ten en cuenta que, puesto que este programa no utiliza gráficos (todavía no los hemos estudiado), en todas las imágenes que se muestran a continuación se ha marcado la opción **Ocultar ventana de salida gráfica** del menú **Opciones**. Así podemos ofrecer imágenes en tamaño más reducido con el consiguiente ahorro de espacio y tóner de impresora/fotocopiadora.



En la parte correspondiente a la pestaña activa de edición (donde se muestra el código del programa), observamos que la primera línea aparece resaltada en color azul. Este realce se utiliza para marcar la instrucción que actualmente vaya a ejecutar el intérprete de Python. También observamos que aparece un menú nuevo denominado **Depurar** en la barra de menús. Este menú permite acceder a las diversas opciones que presenta el "modo depuración", que son las que se muestran en los botones que se presentan en el párrafo siguiente.

Debajo, observamos una parte de la ventana que aparece en color gris. En la parte superior de la ventana vemos un rótulo que reza Variables globales. En esa parte es donde aparecerán sucesivamente los nombres de las variables usadas en el programa y cuál es el valor que actualmente almacenan (sea número, entero o de punto

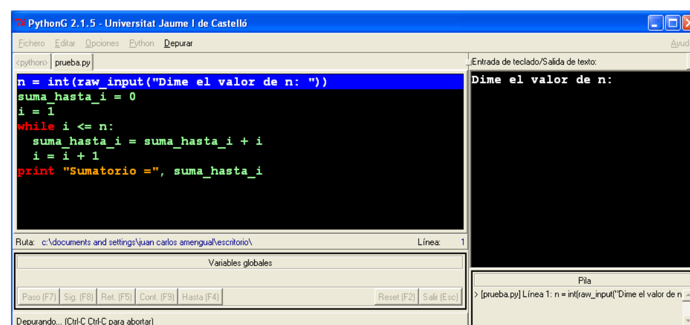
flotante, sea una cadena o una lista). Finalmente, observamos una serie de botones que indican cuál es su función:

- Paso (F7): bajo nuestro punto de vista, este botón (o pulsar la tecla F7 o elegir la correspondiente opción del menú **Depurar**) tiene la *misma funcionalidad* que el siguiente (Sig. (F8)). Realmente lo que hace es continuar con la siguiente instrucción, entrando a ejecutar las instrucciones de las funciones definidas por el usuario (a diferencia del siguiente botón, que no entra dentro del código de las funciones). Como no vamos a trabajar con funciones definidas por el usuario, ignoraremos este botón.
- Sig. (F8): si hacemos clic en este botón, pulsamos la tecla F8 o bien elegimos la correspondiente opción del menú **Depurar**, pasamos a ejecutar la siguiente instrucción del programa. Así, observaremos que se resalta (en azul) la siguiente instrucción y los datos que aparecen en las zonas grises abajo y a la derecha cambian.
- Ret. (F5): bajo nuestro punto de vista, este botón (o pulsar la tecla F5 o elegir la correspondiente opción del menú **Depurar**) tiene la *misma funcionalidad* que el siguiente (Cont. (F9)). Realmente lo que hace es terminar rápidamente la ejecución de la función cuyas instrucciones está ejecutando el intérprete. Como no vamos a trabajar con funciones definidas por el usuario, ignoraremos este botón.
- Cont. (F9): haciendo clic en este botón, pulsando la tecla F9 o bien eligiendo la correspondiente opción del menú **Depurar**, conseguimos que el programa termine su ejecución rápidamente y salga del "modo depuración". Opción útil si ya hemos contemplado paso a paso la ejecución de la parte del programa que nos interesaba y queremos terminar ya la ejecución del mismo saliendo del "modo depuración".
- Hasta (F4): si situamos el cursor en una línea concreta (instrucción determinada) del programa que muestra la pestaña activa de edición y hacemos clic en este botón (alternativamente, pulsamos la tecla F4 o bien elegimos la correspondiente opción del menú **Depurar**), entonces conseguimos que el programa se ejecute rápidamente en modo normal hasta llegar a la instrucción marcada con el cursor. En ese momento se detendrá la ejecución, volviendo al "modo depuración". Esta opción es útil si queremos pasar rápidamente a depurar un punto concreto de nuestro programa.
- Reset (F2): haciendo clic en este botón o pulsando la tecla F2, se interrumpe el proceso de ejecución del programa y se vuelve a iniciar la ejecución del mismo desde la primera instrucción *sin salir del "modo depuración"*.
- Salir (Esc): si hacemos clic en este botón, pulsamos la tecla Escape o bien elegimos la correspondiente opción del menú **Depurar**, saldremos del "modo depuración", abortando (deteniendo) la ejecución del programa.

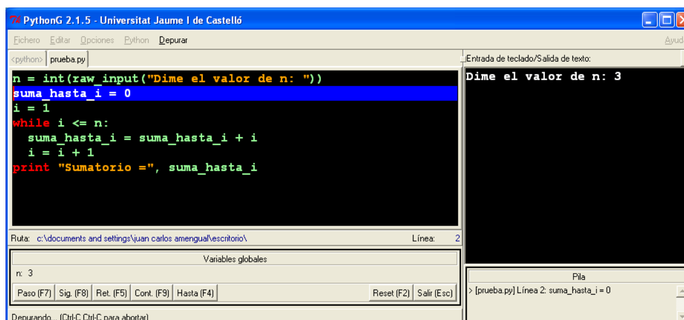
En la parte derecha aparece la ventana de Entrada de teclado/Salida de texto: y, justo debajo, aparece una zona de color gris encabezada con el rótulo Pila. A continuación, se muestra el nombre del fichero, la línea que está analizando el intérprete de Python y la instrucción que se ha escrito en dicha línea. En esta zona de color gris podemos seguir el historial de órdenes ejecutadas por el intérprete en la ejecución del programa.

Vamos a comenzar la ejecución de la traza, para la cual emplearemos el botón Sig. (F8), que permite ejecutar en cada paso una instrucción (línea) completa del programa, como acabamos de ver. Ya sabes que si quieres reiniciar la ejecución "en modo depuración" debes hacer clic en el botón Reset (F2), mientras que si quieres terminar la ejecución en "modo depuración" puedes optar por Salir (Esc) que, además aborta la ejecución del programa, o bien Cont. (F9) que concluye con la ejecución del programa y muestra los resultados (sólo sale del "modo depuración").

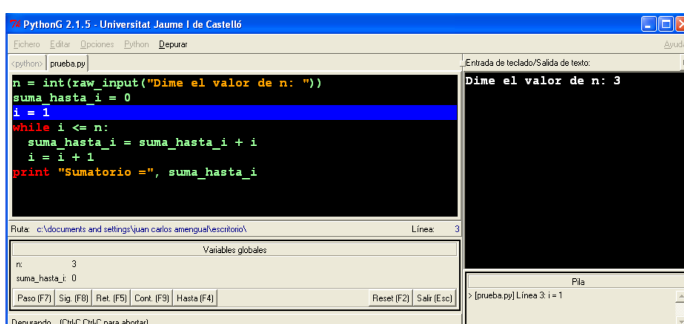
Paso 1. Pulsamos Sig. (F8). En la pestaña activa de edición se resalta la primera línea del programa al tiempo que en la ventana Entrada de teclado/Salida de texto: de la derecha aparece el mensaje correspondiente a la función `raw_input`. Ello indica que se está ejecutando la línea 1 del programa: la lectura de un número entero.



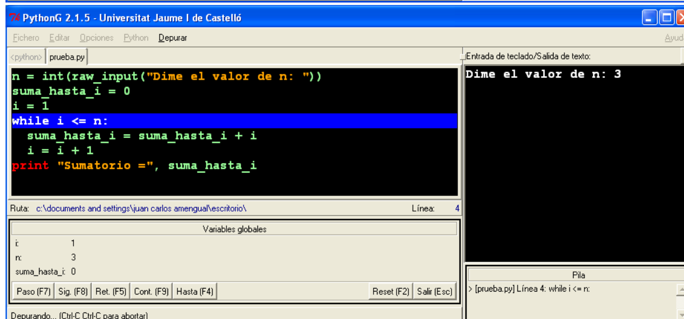
Paso 2. Escribimos el número 3 (la traza la haremos para este valor de `n`) y pulsamos la tecla de retorno de carro. En la zona gris Variables globales, observamos que se ha creado la variable `n` y que contiene el valor leído: `3`. Además, en la pestaña activa de edición se ha resaltado la segunda línea y en el panel gris de abajo a la derecha, etiquetado con el rótulo Pila, ha cambiado a 2 el número de línea del fichero y se muestra la instrucción correspondiente. Ello indica que va a ejecutarse la línea 2 del programa, que es una asignación de un valor a una variable.



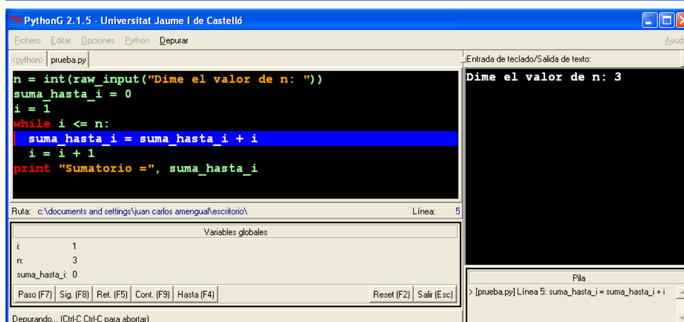
Paso 3. Pulsamos Sig. (F8). En la zona Variables globales, observamos que se ha creado la variable `suma_hasta_i` y que contiene el valor `0` asignado en el programa. Además, en la pestaña activa de edición se ha resaltado la tercera línea y en el panel gris Pila de abajo a la derecha ha cambiado a 3 el número de línea del fichero, apareciendo el código de la instrucción que va a ejecutarse en la que se asigna el valor `1` a la variable `i`.



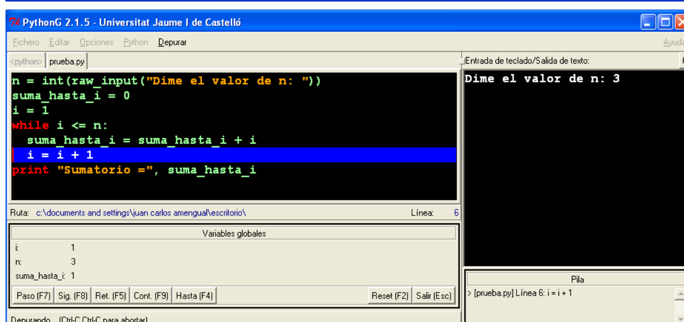
Paso 4. Pulsamos Sig. (F8). En la zona Variables globales, vemos que ha aparecido la variable `i` con el valor `1` asignado al ejecutar la línea 3. Además, en la pestaña activa de edición se ha resaltado la cuarta línea y en el panel gris Pila el número de línea del fichero ahora es 4, mostrando el código de la instrucción que va a ejecutarse: la sentencia de repetición `while`.



Paso 5. Pulsamos nuevamente Sig. (F8). Como se cumple la condición del bucle (`i` vale `1` y `n` vale `3`), se pasa a ejecutar *por primera vez* las líneas 5 y 6. Así, vemos la línea 5 resaltada en la pestaña activa de edición e indicada en el panel gris Pila como la siguiente línea a ejecutar.



Paso 6. Pulsamos Sig. (F8). En la zona Variables globales, vemos que la variable `suma_hasta_i` ha cambiado su valor a `1` al acumularle el valor de `i` en la línea 5. Ahora la línea que se ejecutará es la 6, en la que se incrementa el valor de `i` en `1`, tal como muestra la zona Pila.



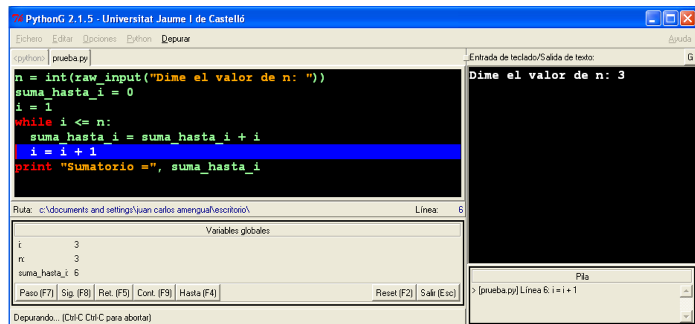
Paso 7. Pulsamos, como siempre, Sig. (F8). En la zona Variables globales, la variable i ahora vale 2. La línea resaltada en la pestaña activa de edición e indicada en el panel Pila como la siguiente línea a ejecutar es otra vez la 4. Tras ejecutar la última línea dependiente (la 6), la ejecución vuelve a la línea de while para decidir si hay que continuar iterando o no. Como i vale 2, la condición sigue cumpliéndose.

Paso 8. Pulsamos una vez más Sig. (F8). Al cumplirse la condición, se debe repetir (ejecutar por segunda vez) el cuerpo del bucle (líneas 5 y 6). Así, vemos que la siguiente línea a ejecutar es la 5, en la que se acumulará el valor de i (que ahora vale 2) a suma_hasta_i.

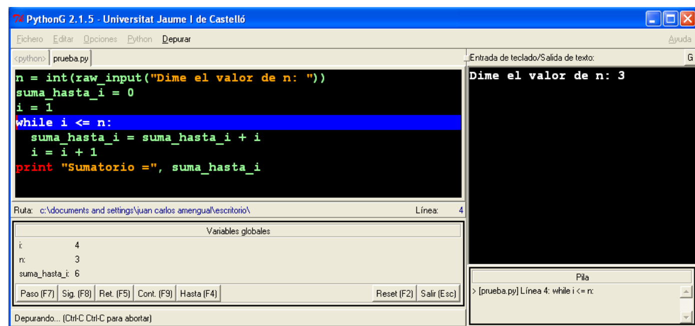
Paso 9. Pulsamos Sig. (F8). En la zona Variables globales, la variable suma_hasta_i ha pasado a valer 3 tras ejecutar la línea 5. Ahora se ejecutará la línea 6, como podemos observar, y se incrementará otra vez el valor de i en 1.

Paso 10. Pulsamos nuevamente Sig. (F8). En la zona Variables globales, la variable i ha cambiado su valor a 3. La línea resaltada en la pestaña activa de edición e indicada en el panel Pila como la siguiente a ejecutar vuelve a ser la 4. Tras la línea 6, se regresa a la línea de while para decidir otra vez si hay que continuar iterando o no. La variable i vale 3, así pues la condición aún sigue cumpliéndose.

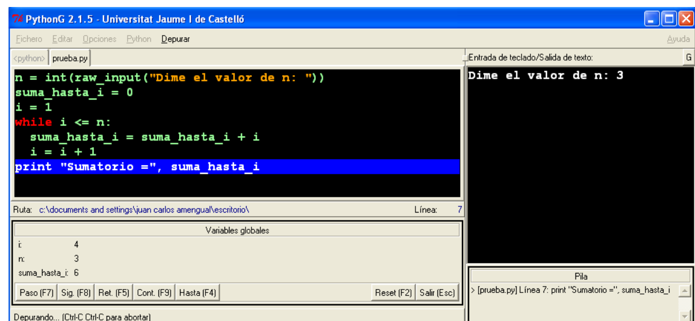
Paso 11. Pulsamos Sig. (F8). Al cumplirse la condición, se itera (por tercera vez) el cuerpo del bucle (líneas 5 y 6). Así pues, se volverá a ejecutar la línea 5, acumulándose el actual valor de i (3) a suma_hasta_i.



Paso 12. Pulsamos Sig. (F8). En la zona Variables globales, vemos que `suma_hasta_i` ahora vale 6 tras ejecutar la acumulación de la línea 5. A continuación, se ejecutará la línea 6 para incrementar nuevamente el valor de `i` en `1`.



Paso 13. Pulsamos Sig. (F8). En la zona Variables globales, la variable `i` ha alcanzado el valor `4`. Vemos una vez más resaltada en la pestaña activa de edición e indicada en el panel Pila la línea 4 como la siguiente a ejecutar. Un nuevo regreso a la línea de `while` para decidir si hay que continuar iterando o no. Pero esta vez es la última, ya que el valor 4 de `i` hará que la condición *ya no se cumpla*.



Paso 14. Pulsamos Sig. (F8). En efecto, la ejecución ha saltado a la última línea del programa, la 7, como podemos ver en la pestaña activa de edición y en el panel Pila. La falsedad de la condición de `while` detiene sus iteraciones, siguiendo la ejecución con la instrucción posterior a `while`. Así, se ejecutará `print` para mostrar el resultado por pantalla.

Finalización. Pulsamos Sig. (F8) por última vez. En la ventana Entrada de teclado/Salida de texto:, vemos que ha aparecido una línea que informa del resultado del sumatorio desde 1 hasta 3 y que ha sido escrita por la sentencia `print`. El resultado mostrado es el valor alcanzado por la variable `suma_hasta_i` a lo largo de las sucesivas acumulaciones realizadas. También desaparecen los paneles grises y el menú **Depurar**. Esto es lo que ocurre al finalizar (o interrumpir) la ejecución de la traza: PythonG vuelve al modo "normal" de edición en la pestaña activa.

Prueba a realizar mediante PythonG distintas trazas de este programa dándole otros valores cuando pida el número `n` (pero no le des un valor muy alto, si no será muy pesado). Asimismo, prueba a realizar algunas trazas de programas que hemos visto anteriormente y de los que veremos a partir de ahora.

25.2. Algunos programas sencillos

Vamos a estudiar algunos ejemplos que nos irán introduciendo distintos usos de la sentencia `while`, así como algunos aspectos que debemos tener en cuenta al utilizarla. Primero, vamos conocer una nueva forma de realizar un control de datos erróneos en un programa. Date cuenta que el control de datos erróneos que habíamos realizado hasta ahora, mediante sentencias de selección, permite dar un mensaje indicativo del valor erróneo que se ha introducido, pero los programas finalizan sin poder realizar el cálculo que en principio se deseaba hacer.

Ahora que podemos utilizar sentencias de repetición, podemos realizar un control de datos erróneos mucho

mejor. Es decir, podemos repetir la introducción de datos mientras sean incorrectos. Así, un programa no finalizará al introducir un dato erróneo, sino que pedirá y leerá repetidamente el dato mientras sea erróneo. Y en cuanto sea correcto, finalizará el bucle y realizará el cálculo deseado.

Ejemplo: El siguiente programa es una ampliación del programa básico para calcular el área de un círculo a partir de su radio. Introducimos en él una sentencia de repetición que lleva anidadas sentencias para indicar el error cometido, y pedir y leer nuevamente el dato. Esta sentencia de repetición se ejecutará mientras el radio leído sea negativo, lo cual expresamos en la condición del bucle. En cuanto se introduzca un valor positivo del radio, el bucle finalizará. Más aún, si el valor leído la primera vez ya es positivo, el bucle ni siquiera llegará a ejecutarse.

```
print "Cálculo del área de un círculo a partir de su radio."
from math import pi
r = int(raw_input('Dime el radio (en centímetros): '))
while r < 0:
    print "El radio no puede ser negativo."
    r = int(raw_input('Dime el radio (en centímetros): '))
area = pi * r ** 2
print 'Su área mide', area, 'centímetros cuadrados.'
```

Prueba a realizar una traza de este programa, leyendo sucesivamente para `r` los valores -12, -6 y 5.

Fíjate que en este programa hemos escrito (exactamente igual) dos veces la instrucción para leer el radio: una antes de iniciar el bucle y otra incluida en él. Así, la primera instrucción de lectura siempre se ejecutará y, si el dato es correcto la primera vez que se introduce, el bucle no se ejecutará, y por tanto, la lectura incluida en él tampoco. Podemos realizar de otra manera el control de datos erróneos con una sentencia de repetición.

Ejemplo: En esta nueva versión escribimos una única línea de lectura del radio, que necesariamente debe estar incluida en el bucle para poder controlar el valor del radio y repetirla si éste fuese negativo.

```
print "Cálculo del área de un círculo a partir de su radio."
from math import pi
r = -1
while r < 0:
    r = int(raw_input('Dime el radio (en centímetros): '))
area = pi * r ** 2
print 'Su área mide', area, 'centímetros cuadrados.'
```

Pero date cuenta que ahora al ejecutar el programa, se intentará comprobar el valor de `r` en la condición de `while` *antes* de realizar su primera lectura. Ello conduciría a un error de ejecución del programa si `r` no tuviese un valor asignado, ya que en ese caso `r` no existiría y el intérprete produciría un error de nombre.

Además de asignarle un valor inicial a `r`, es importante considerar *qué valor* le asignamos. Como este valor inicial no será un valor útil para el cálculo que realizará el programa, y por tanto, pretenderemos que la línea de lectura anidada a `while` llegue a ejecutarse al menos una vez (para que el usuario pueda introducir el radio que desee), el valor inicial de `r` sí debe ser útil para esta pretensión. Es decir, el valor inicial debe ser tal que *cumpla la condición de* `while` *la primera vez*, para así forzar al menos una iteración, y con ella, al menos una ejecución de la línea de lectura. En este caso, cualquier valor negativo produce el efecto deseado.

Observa que esta versión no es equivalente del todo a la anterior, en cuanto a la interacción del programa con el usuario en la ventana de ejecución. Esta versión no indica qué está sucediendo para que el programa pida sucesivamente el radio, cuando los valores introducidos son negativos. Simplemente, lo pide sucesivas veces. Haría falta una sentencia `print`, como la que aparecía en la versión anterior, para realizar esta labor de aviso.

Esta sentencia `print` debe situarse tras la línea de lectura, pero si se ejecutase sin más tras la lectura, escribiría el mensaje de aviso también cuando el valor de `r` recién leído fuese positivo. Así, tras la lectura debe emplearse una sentencia de selección para comprobar el valor de `r` recién leído y escribir el mensaje de aviso sólo si `r` es negativo. La versión que obtenemos finalmente es la siguiente.

```
print "Cálculo del área de un círculo a partir de su radio."
from math import pi
r = -1
while r < 0:
    r = int(raw_input('Dime el radio (en centímetros): '))
    if r < 0:
        print "El radio no puede ser negativo."
area = pi * r ** 2
print 'Su área mide', area, 'centímetros cuadrados.'
```

Compara esta versión con la del ejemplo anterior. Ejecuta ambas introduciéndoles las mismas secuencias de valores. Observa su idéntica apariencia al ejecutarse en la ventanita para Entrada de teclado/Salida de texto: y sus diferentes acciones al realizar trazas con dichas secuencias de valores.

Vamos a ver ahora cómo gestionar de diferentes formas alguna de las variables empleadas en la condición que escribimos en la línea de `while`. En el primer ejemplo que hemos visto, una variable `i` incrementaba su valor de 1 en 1 y otra `n` mantenía siempre el mismo valor, y la condición establecía que el bucle se ejecutase mientras `i` no superase a `n`. En el segundo ejemplo, una variable `r` podía tomar sucesivamente cero o más valores (leídos) negativos, hasta tomar el primer valor positivo, momento en el que el bucle finalizaba porque así lo establecía la condición.

Es importante fijarse en cómo evolucionan los valores de esas variables y su relación en la condición. Podemos gestionar las variables que forman parte de la condición de muy distintas formas, lo que nos permite realizar tratamientos iterativos de muy diversa índole.

Ejemplo: Vamos a desarrollar un primer programa para calcular el factorial de un número que se leerá por teclado. En general, el factorial de un número es:

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \cdots \cdot (n-1) \cdot n$$

siendo $0! = 1$ (caso particular). Si nos fijamos, resulta que el cálculo del factorial de un número n se puede ir acumulando de forma similar a como lo hacíamos en el cálculo del sumatorio desde 1 hasta n; es decir, se multiplicará como si los productos estuviesen parentizados de la siguiente manera:

$$n! = ((\cdots(((1 \cdot 2) \cdot 3) \cdot 4) \cdot \cdots) \cdot (n-1)) \cdot n$$

Con esta primera idea, realizamos un programa que mediante un bucle irá incrementando el valor de una variable `contador`, de 1 en 1 desde 1 hasta n, y realizando sucesivos productos, para al final contener el valor del factorial.

```
n = int(raw_input("Dime el valor de n: "))
factorial = 1
contador = 1
while contador <= n:
    factorial = factorial * contador
    contador = contador + 1
print "Factorial("+str(n)+") =", factorial
```

Compara este programa con el del cálculo del sumatorio. En este programa el valor de `contador` va creciendo desde 1 hasta el valor leído de `n`, con lo que los productos parciales se obtienen de forma creciente.

Pero alternativamente, podemos plantear un bucle que obtenga los productos parciales de forma decreciente, comenzando con un valor de `contador` igual a `n` y decrementándolo de 1 en 1 hasta llegar a 1. Así, se multiplicará como si los productos estuviesen parentizados de esta otra manera:

$$n! = ((\cdots(((n \cdot (n-1)) \cdot (n-2)) \cdot (n-3)) \cdot \cdots) \cdot 2) \cdot 1$$

El segundo programa que planteamos para realizar el cálculo del factorial de un número `n`, siguiendo esta estrategia de obtener los productos parciales de forma decreciente, es el siguiente:

```
n = int(raw_input("Dime el valor de n: "))
factorial = 1
contador = n
while contador > 0:
    factorial = factorial * contador
    contador = contador - 1
print "Factorial("+str(n)+") =", factorial
```

Compara este programa con el anterior, realizando trazas con ellos para un mismo valor leído de `n`.

A continuación, tienes otra manera distinta de manipular los valores que va tomando sucesivamente la variable que se emplea en la condición del bucle.

Ejemplo: Vamos a realizar un programa que permita contar el número de dígitos de un número entero positivo introducido por teclado. Si, por ejemplo, el número leído fuese el 17938, su número de dígitos sería 5. Pero, ¿cómo podemos calcularlo? Observa las siguientes divisiones enteras:

Dividendo	Divisor	Resto	Cociente
17938	10	8	1793
1793	10	3	179
179	10	9	17
17	10	7	1
1	10	1	0

Dividiendo sucesivamente por 10, primero el número leído, y después los cocientes que se vayan calculando, obtenemos números que tienen un dígito menos cada nueva división. Fíjate que la última división da 0 como cociente, y también que contando el número de divisiones realizadas obtenemos el número de dígitos del número inicial.

El programa lo realizaremos siguiendo esta idea. Se leerá el número entero positivo e inicializaremos a 0 una variable llamada `num_digitos`, con la que contaremos el número de divisiones. Después comenzará una sentencia de repetición, en la que cada vez se dividirá el último cociente por 10 y se incrementará `num_digitos` en 1. Mientras el cociente que se obtenga sea mayor que 0, seguiremos repitiendo las sentencias anidadas, y en cuanto sea 0, finalizará el bucle. Al acabar el bucle, se habrá contado el número de divisiones realizadas, y por tanto, el número de dígitos del número inicial.

```
numero = int(raw_input("Dime un número positivo: "))
num_digitos = 0
cociente = numero
while cociente > 0:
    cociente = cociente / 10
    num_digitos = num_digitos + 1
print "El número", numero, "tiene", num_digitos, "dígitos."
```

Fíjate que la variable `cociente` se emplea en el control del bucle, pero sus sucesivos valores no se emplean en el cálculo que se va realizando, al contrario de lo que se hacía con las variables `i` y `contador` en los programas de cálculo del sumatorio y el factorial, respectivamente. La variable `cociente` influye en el cálculo del número de dígitos tan sólo por el número de iteraciones que hace que se realicen en el bucle.

Otra cuestión muy importante a tener en cuenta cuando se utilizan bucles con condición (sentencias `while`), es que hay que asegurarse de que la condición llegará a cumplirse en tiempo finito. Es decir, debe garantizarse que el bucle terminará. Para ello, en general, alguna de las sentencias del cuerpo del bucle tiene que modificar (en algún momento) el valor de alguna variable que afecte al resultado de la evaluación posterior de la condición.

Se puede observar en el ejemplo del sumatorio que la condición comprueba si la variable `i` es menor o igual que la variable `n`, y dentro del bucle se incrementa (en 1, cada vez) el valor de `i`. De esta forma, llegará un momento en que el valor de `i` sobrepasará al de `n`, con lo que acabará el bucle. Veamos un par de ejemplos típicos de errores en el diseño de bucles que dan lugar a *bucles infinitos*, o sea, bucles que no acaban nunca.

Ejemplo: El siguiente programa pretende calcular el sumatorio desde 1 hasta un número dado n. De hecho, fíjate que es prácticamente el mismo que vimos anteriormente.

El error que se ha cometido al escribir este programa consiste en que no se incrementa la variable `i` dentro del bucle. Así, en cada iteración, `i` siempre tiene el valor 1. Y si la variable `n` tiene un valor mayor o igual que 1, entonces el bucle no acabará nunca, ya que la condición se evaluará siempre a *cierto*, y nunca cambiará a *falso*.

```
n = int(raw_input("Dime el valor de n: "))
suma_hasta_i = 0
i = 1
while i <= n:
    suma_hasta_i = suma_hasta_i + i
print "Sumatorio =", suma_hasta_i
```

Añadiendo la instrucción `i = i + 1`, como segunda instrucción anidada a la sentencia `while`, corregimos el error de este programa, con lo que obtenemos el programa que ya vimos que realizaba correctamente el cálculo del sumatorio.

Ejemplo: En este otro programa, una variable que afecta a la condición sí se modifica, para que ésta pueda llegar a ser *falso* en algún momento. Pero la condición no está bien expresada y, por tanto, puede no llegar a incumplirse.

Este programa pretende calcular la suma de todos los números impares comprendidos entre 1 y un número leído `n`. Así, si `n` fuese 7 se realizaría la suma $1+3+5+7$. Para realizar el cálculo sigue la misma estrategia que el programa de cálculo del sumatorio desde 1 hasta `n`, pero ahora incrementa la variable `i` de 2 en 2 desde 1, con lo que `i` va tomando todos los valores impares hasta alcanzar `n+2`. Si `n` es 7, el bucle acabará cuando, tras acumular el valor 7, `i` tome el valor 9 y en ese momento se compuebe que la condición se incumple (`9 != 7 + 2` es *falso*). Así, el bucle acabará habiendo acumulado todos los números impares desde 1 hasta 7 (9 no lo habrá sumado). Similarmente, el programa funcionará correctamente con cualquier número `n` que sea impar.

```
n = int(raw_input("Dime el valor de n: "))
suma_impar_hasta_i = 0
i = 1
while i != n + 2:
    suma_impar_hasta_i = suma_impar_hasta_i + i
    i = i + 2
print "Sumatorio de números impares =", suma_impar_hasta_i
```

¿Qué sucede si `n` es par? La condición, tal como está expresada ahora, exige que el valor de `i` sea exactamente igual al valor de `n+2` para que la condición se evalúe a *falso*. Así pues, cuando el valor de `n` sea par, esta igualdad no se producirá nunca, ya que los valores de `i` pasarán de largo y se producirá otro bucle infinito.

Cambiando la condición por `i <= n`, el programa funcionará correctamente con cualquier valor de `n`, par o impar, ya que de esta manera en cuanto `i` sobrepase a `n`, el bucle acabará, habiendo sumado todos los números impares menores o iguales que `n`.

Se debe tener en cuenta qué condición se escribe para controlar la ejecución del bucle, así como los valores iniciales de las variables que afectan a la condición y los que irán tomando sucesivamente al iterar el bucle. Todo ello debe estar adecuadamente relacionado para que el bucle tenga una ejecución correcta.

Otra característica de los bucles con condición inicial que conviene tener siempre en cuenta es el hecho de que, si la primera vez que se evalúa la condición el resultado es *falso*, entonces el bucle se ejecuta cero veces (no se ejecuta). Este aspecto también es importante tenerlo presente al diseñar un bucle.

Por ejemplo, fíjate en el primer programa que vimos para realizar un control de errores en la lectura del valor del radio. Allí, si el primer valor leído del radio ya es positivo, la condición del bucle siguiente se incumple la primera vez y el bucle no se ejecuta. El cálculo posterior se realiza correctamente tras no ejecutarse el bucle. Similarmente, en los programas que calculan el factorial de `n`, si el valor de `n` es 0 entonces el bucle no se ejecuta. El valor inicial de la variable `factorial` es 1, con lo que, tras no ejecutar el bucle, se escribe el valor correcto de 0!.

Ejercicio 41. Observa el bucle del programa que calcula el número de dígitos de un número entero positivo. ¿Qué sucede si el valor leído del teclado es 0? ¿Cuenta bien el número de dígitos de 0?

Si el programa no cuenta correctamente el número de dígitos de 0, modifícalo para preveer este caso.

Ejercicio 42. Escribe un programa que lea dos números c y n, calcule el valor de

$$\sum_{i=1}^{n} c$$

mediante una sentencia `while` y escriba el resultado por la pantalla. c puede ser cualquier número flotante y n debe ser un número entero positivo ($n > 0$). Además, el programa se asegurará mediante un bucle de validación de datos que el valor de n sea positivo.

Ejercicio 43. Escribe un programa que lea un número entero positivo n, calcule el valor de

$$\sum_{i=1}^{n} i^2$$

mediante una sentencia `while` y escriba el resultado por la pantalla. Además, el programa se asegurará mediante un bucle de validación de datos que el valor de n sea positivo.

Ejercicio 44. Escribe un programa que lea m valores enteros por el teclado, calcule la suma, la media, el máximo y el mínimo de los m valores, y escriba los resultados por la pantalla. El programa leerá primero el valor de m, asegurándose mediante un bucle de validación de datos que éste es mayor o igual que 1.

25.3. Mejorando la lectura de listas

Hasta conocer la sentencia de repetición `while`, la lectura de una lista podíamos hacerla definiendo previamente su longitud y escribiendo explícitamente en el programa instrucciones para leer cada uno de sus elementos. Por ejemplo, podíamos leer una lista de cuatro elementos enteros de la siguiente forma:

```
milista = [0] * 4
milista[0] = int(raw_input("Dime el elemento 0: "))
milista[1] = int(raw_input("Dime el elemento 1: "))
milista[2] = int(raw_input("Dime el elemento 2: "))
milista[3] = int(raw_input("Dime el elemento 3: "))
```

Estas últimas cuatro instrucciones, que leen un valor entero del teclado y lo asignan a una posición determinada de la lista, son esencialmente la misma acción. En ellas únicamente varía el índice de la posición en la que se situará el elemento leído y el mensaje que muestra la instrucción `raw_input` por pantalla al usuario.

Los índices de una lista de n elementos son enteros entre 0 y $n-1$, y ya hemos visto, en los ejemplos anteriores de la sentencia `while`, cómo dar un valor inicial a una variable e ir incrementándola en un bucle hasta alcanzar un valor final.

Así, podemos dar el valor inicial 0 a una variable y mediante un bucle ir incrementándola de 1 en 1 hasta alcanzar el índice del último elemento de la lista. Al tiempo que el bucle itera, podemos ir leyendo los elementos de la lista y utilizando el valor de la variable para indexar la posición en la que se situará cada elemento leído. También aprovechamos ese valor para modificar ligeramente el mensaje de la instrucción `raw_input`. El código siguiente, al ejecutarse, realiza las mismas acciones que la secuencia de instrucciones anterior.

```
milista = [0] * 4
i = 0
while i < 4:
    milista[i] = int(raw_input("Dime el elemento "+str(i)+": "))
    i = i + 1
```

Si deseamos leer una lista de muchos más elementos que 4, tan sólo tenemos que modificar este número en la creación de la lista y en la condición de `while`. Antes, hubiésemos tenido que escribir tantas líneas de lectura como elementos hubiese que leer. Sin embargo, esta forma de leer una lista, aunque evita tener que escribir repetidamente líneas de lectura de elementos, requiere que la longitud de la lista esté establecida en el propio programa.

Ejercicio 45. La siguiente secuencia de instrucciones también permite leer una lista de 4 elementos, pero basándose en la operación de añadir elementos a una lista en lugar de hacerlo en la operación de modificar. La longitud de la lista queda preestablecida por el número de líneas de lectura que se realizan.

```
milista = []
milista = milista + [int(raw_input("Dime el elemento 0: "))]
milista = milista + [int(raw_input("Dime el elemento 1: "))]
milista = milista + [int(raw_input("Dime el elemento 2: "))]
milista = milista + [int(raw_input("Dime el elemento 3: "))]
```

Escribe el código Python que, utilizando una sentencia de repetición `while`, realice al ejecutarse las mismas acciones que la secuencia de instrucciones anterior. Es decir, que mediante un bucle lea una lista de 4 elementos basándose en añadir elementos a la lista.

En ocasiones, puede ser necesario leer una lista cuya longitud sea desconocida en el momento de escribir el programa para que esta longitud pueda ser distinta cada vez que éste se ejecute. Por ejemplo, si queremos leer una lista con los precios individuales de los productos que adquirimos cada vez que vamos a comprar al supermercado, querremos que esta lista pueda variar su longitud cada vez que el programa se ejecute, ya que en cada compra podemos adquirir distinto número de productos.

Podemos leer una lista de tamaño variable, comenzando con una lista vacía y añadiéndole elementos mediante un bucle mientras éstos cumplan alguna condición. Por ejemplo, podemos leer números y añadirlos a la lista mientras éstos sean positivos. En cuanto se introduzca el primer número negativo, la lectura finaliza y la lista queda con todos los números positivos introducidos hasta ese momento. El siguiente código lee una lista con todos los números flotantes positivos introducidos hasta que se teclee uno negativo.

```
milista = []
i = 0
elem = float(raw_input("Dime el elemento "+str(i)+": "))
while elem >= 0:
    milista = milista + [elem]
    i = i + 1
    elem = float(raw_input("Dime el elemento "+str(i)+": "))
```

Fíjate que, en este código, se utiliza una variable auxiliar `elem` para leer el número flotante. Es necesario utilizar una variable auxiliar, puesto que hasta que no leemos el número no podemos hacer comprobaciones sobre él y sólo si es positivo se añadirá a la lista; si es negativo, no. Nota que en el mensaje de petición del número sí se indica la posición en la que se situará, pero si el número introducido y almacenado en `elem` en ese momento es negativo, incumplirá la condición de `while` y hará que el bucle finalice. La lista quedará con los elementos añadidos hasta ese momento y sin incluir este último, que se hubiese situado en la posición indicada en el mensaje de haber sido positivo.

Ejercicio 46. Escribe un código Python que permita leer una lista de tamaño variable cuyos elementos sean cadenas. La finalización de la lectura de la lista vendrá indicada por la cadena vacía; o sea, por la pulsación de la tecla de retorno de carro directamente tras la petición de un nuevo elemento (sin teclear ningún otro carácter).

26. Sentencia de repetición `for`

Python proporciona otra sentencia de repetición: la sentencia `for`, que es un iterador sobre secuencias. Es decir, es una sentencia que permite "visitar" todos y cada uno de los elementos que contiene un secuencia y en el orden en el que están dispuestos. La sentencia `for` tiene la siguiente forma general:

```
for variable in secuencia:
    instrucciones
```

La *`secuencia`* puede ser cualquier expresión en Python que proporcione como resultado una secuencia. De esta manera, el funcionamiento de la sentencia es:

- Cada vez que se ejecuta la línea `for variable in secuencia:`
  - Si es la primera vez, se asignará a `variable` el primer elemento de la `secuencia` y *se ejecutarán* las `instrucciones`.
  - Para cualquier otra vez, se asignará a `variable` el siguiente elemento (respecto al último asignado) de la `secuencia` y *se ejecutarán* las `instrucciones`.
- Cuando ya no quedan elementos en la `secuencia`, la sentencia `for` finaliza y la ejecución proseguirá con la sentencia que haya escrita después.

Expresado informalmente, las `instrucciones` se ejecutan *para todo elemento en la secuencia*, y dejan de ejecutarse cuando ya no quedan elementos por "visitar". Para nosotros, la `secuencia` de una sentencia `for` podrá ser una cadena o una lista. Vamos a ver ejemplos utilizando ambas.

Ejemplo: El siguiente bucle:

```
for car in 'Tila':
    print "La variable  car  ahora contiene el carácter:", car
```

produce la siguiente salida por la ventana de ejecución:

```
La variable  car  ahora contiene el carácter: T
La variable  car  ahora contiene el carácter: i
La variable  car  ahora contiene el carácter: l
La variable  car  ahora contiene el carácter: a
```

Fíjate que en cada iteración de la sentencia `for` se ha asignado a la variable uno de los caracteres de la cadena, de manera que, ordenadamente y de uno en uno, ha ido conteniendo a todos ellos hasta terminarse la cadena. Igualmente ocurre cuando la secuencia es una lista.

Ejemplo: El siguiente bucle:

```
for num in [100, 500, 1000]:
    print "Precio:", num, "euros sin IVA,", num*1.16, "euros con IVA"
```

produce la siguiente salida por la ventana de ejecución:

```
Precio: 100 euros sin IVA, 116.0 euros con IVA
Precio: 500 euros sin IVA, 580.0 euros con IVA
Precio: 1000 euros sin IVA, 1160.0 euros con IVA
```

En cada iteración del bucle, se ejecutan las instrucciones dependientes de la línea de `for`, y si éstas realizan algún cálculo empleando la variable, lo harán con el valor de la lista asignado en esa iteración. *Las variables a las que una sentencia* `for` *asigna valores, únicamente deben consultarse en las instrucciones dependientes y no debe asignárseles ningún otro valor.* Por ejemplo, el valor de la variable `num`, en el ejemplo anterior, se usa para efectuar los cálculos dentro del bucle, pero este valor nunca es modificado por las instrucciones dependientes de `for`.

Observa que, al igual que ocurría con `if` y `while`, la línea de `for` debe finalizar en dos puntos `:`, y las instrucciones dependientes, que van a constituir el cuerpo del bucle, deben estar indentadas un nivel más que el nivel de indentación de `for`. En el siguiente ejemplo, en el que se procesa una cadena con un bucle `for`, se anida una selección múltiple en el bucle, lo que conduce a aumentar el nivel de indentación de las instrucciones que primero dependen del bucle y después dependen de la selección.

Ejemplo: Vamos a realizar un programa que leerá una cadena de caracteres y, a continuación, contará cuántos de esos caracteres son dígitos, cuántos son letras mayúsculas y cuántos son letras minúsculas. Finalmente, escribirá las cantidades obtenidas de dígitos y letras mayúsculas y minúsculas.

El procedimiento que va a seguir el programa, tras leer la cadena, comienza por dar el valor 0 a tres contadores, que van representar el número de dígitos (`num_digit`), el número de letras mayúsculas (`num_mayus`) y el número de letras minúsculas (`num_minus`) que se han observado en la cadena tras haberla recorrido hasta un determinado carácter. Antes de comenzar a visitar los caracteres de

la cadena, aún no se ha observado ningún carácter de los tipos indicados, por eso los contadores se inician con 0. Tras visitar todos los caracteres de la cadena, estos contadores contendrán la cantidad de caracteres de cada tipo que hay en la cadena.

Mediante una sentencia `for` se recorren uno a uno los caracteres de la cadena. Cada carácter se clasificará en uno de los tipos indicados (dígitos, mayúsculas o minúsculas) con una selección múltiple, y una vez clasificado en el tipo adecuado se incrementará el contador correspondiente. Si el carácter no es un dígito, una mayúscula o una minúscula, entonces no se hace absolutamente nada.

```
from string import digits, uppercase, lowercase
cadena = raw_input("Dime una cadena cualquiera: ")
num_digit = 0
num_mayus = 0
num_minus = 0
for caracter in cadena:
    if caracter in digits:
        num_digit = num_digit + 1
    elif caracter in uppercase:
        num_mayus = num_mayus + 1
    elif caracter in lowercase:
        num_minus = num_minus + 1
print "Número de dígitos:", num_digit
print "Número de letras mayúsculas:", num_mayus
print "Número de letras minúsculas:", num_minus
```

En las condiciones de la selección múltiple se comprueba la pertenencia del carácter a una de las cadenas predefinidas en el módulo `string`, cada una de las cuales contiene todos los caracteres del tipo correspondiente, utilizando el operador `in`. Fíjate que en la sentencia `for` también se utiliza la palabra `in`, aunque ahí no tiene significado por sí sola como sí lo tiene en el operador de pertenencia `in`.

En la última alternativa de la selección múltiple, tras comprobar si el carácter es un dígito o si es una letra mayúscula, hay que comprobar explícitamente si el carácter es una minúscula con una claúsula `elif` y la correspondiente condición. No podemos utilizar la claúsula `else` ya que aquí no es adecuada. Hay otros caracteres que no son dígitos, mayúsculas o minúsculas, y que llevarían a incrementar el contador `num_minus` erróneamente si se utilizase `else` como última alternativa.

Realiza una traza de este programa, mediante el "modo depuración" de PythonG, introduciéndole la cadena "`Aula: TD2304`".

Veamos otro ejemplo más. Esta vez procesaremos una lista mediante una sentencia `for`.

Ejemplo: Ahora desarrollaremos un programa que primero leerá una lista de números enteros y después otro número entero, y finalmente mostrará por pantalla aquellos números de la lista que son múltiplos del otro.

La lista de números que se leerá será una lista de tamaño variable, empleando un bucle `while`, tal y como vimos anteriormente. Se leerán sólo números positivos para la lista, con lo que el final de la introducción de elementos se indicará con un número negativo.

Posteriormente, se recorrerá la lista de números con una sentencia `for`. Para cada número, se descubrirá si es múltiplo del número dado aparte, y si es así, se escribirá por pantalla.

¿Cómo descubrimos si un número es múltiplo de otro? Un número es múltiplo de otro cuando es divisible por él, es decir, cuando el resto que se obtiene al dividirlo por él es 0. Así pues, ésta es la condición que escribimos en la sentencia de selección anidada al bucle. Sólo los números de la lista que sean divisibles por el número dado aparte serán múltiplos suyos, y sólo estos se escribirán por pantalla.

```
lista = []
numero = int(raw_input("Dime el primer número: "))
while numero >= 0:
    lista = lista + [numero]
```

```
    numero = int(raw_input("Dime el siguiente número: "))
divisor = int(raw_input("Dime un número para descubrir sus múltiplos: "))
for numero in lista:
    if numero% divisor == 0:
        print "Múltiplo:", numero
```

Realiza una traza de este programa, mediante el "modo depuración" de PythonG, introduciéndole la lista de números 3, 25, 12, 15, 17, 33, 50, 95, -4, y después el número 5.

Ejercicio 47. Realiza un programa que lea una cadena y un carácter, y a continuación cuente las veces que ese carácter aparece en la cadena y muestre el resultado por pantalla. El programa anterior que contaba los dígitos, mayúsculas y minúsculas de una cadena te será de ayuda para realizar éste.

26.1. La función interna `range`

Disponemos de una función interna que resulta especialmente útil para ser empleada en la realización de bucles `for`: la función `range`. Veamos primero cómo funciona aisladamente esta función, y después cómo incorporarla en una sentencia `for`.

`range` es una función que devuelve una lista de números enteros, y los números que forman esta lista se determinan a partir del argumento o de los argumentos pasados a la función.

Podemos utilizar `range` con un solo argumento, que será un número n. En este caso, la lista devuelta consistirá en todos los números enteros desde 0 hasta $n - 1$.

Ejemplo:
```
»> range(5)
[0, 1, 2, 3, 4]
»> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

También podemos utilizar `range` con dos argumentos, que serán dos números m y n. En este caso, la lista devuelta estará compuesta por todos los números enteros desde m hasta $n - 1$.

Ejemplo:
```
»> range(-2, 5)
[-2, -1, 0, 1, 2, 3, 4]
»> range(4, 10)
[4, 5, 6, 7, 8, 9]
```

Nota que, cuando utilizamos dos argumentos m y n, el segundo tiene el mismo significado que cuando sólo usamos un argumento n: uno más que el último número incluido en la lista. Cuando utilizamos dos, el primer argumento es el primer número incluido en la lista, pero cuando usamos uno, el primer número incluido en la lista es siempre 0.

Finalmente, podemos utilizar `range` con tres argumentos, que serán tres números m, n e i. En este caso, la lista devuelta estará compuesta por los números enteros que comienzan en m, son menores o iguales que $n - 1$, y la diferencia entre cada dos números consecutivos es i.

Ejemplo:
```
»> range(1, 13, 3)
[1, 4, 7, 10]
»> range(0, 21, 4)
[0, 4, 8, 12, 16, 20]
```

En otras palabras, i es el incremento que hay que aplicar a cada número de la lista para calcular el siguiente, comenzando en m y finalizando antes de alcanzar o rebasar a n. i también puede ser un decremento, es decir, puede ser un número negativo. En este caso, m deberá ser mayor que n, y la lista devuelta será decreciente.

Ejemplo:
```
»> range(10,0,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
»> range(15,2,-3)
[15, 12, 9, 6, 3]
```

Fíjate que utilizar `range` con tres argumentos siendo el tercero 1 es equivalente a utilizarla con dos, y que utilizarla con dos siendo el primero 0 es equivalente a utilizarla con uno. Utilizarla con tres argumentos siendo el primero 0 y el tercero 1 también es equivalente a utilizarla con uno.

Ejemplo:
```
»> range(0,5,1)
[0, 1, 2, 3, 4]
»> range(0,5)
[0, 1, 2, 3, 4]
»> range(5)
[0, 1, 2, 3, 4]
```

Como `range` devuelve una lista de enteros resulta fácil imaginar cómo se puede incorporar esta función en una sentencia `for`: como una secuencia sobre la que iterar.

Ejemplo: El siguiente bucle muestra todos los múltiplos de 3 comprendidos entre 1 y 300.

```
for num in range(1,301):
    if num% 3 == 0:
        print "Múltiplo de 3:", num
```

Así, podemos emplear `range` en un bucle `for` siempre que queramos procesar una lista de enteros que se pueda "construir" con esta función.

Por otra parte, recuerda que `len(s)` obtenía la longitud de una secuencia `s`. De esta manera, podemos generar la lista de índices de la secuencia `s` (cadena o lista) con `range(len(s))`. Y empleando la lista de índices de la secuencia en un bucle `for` podemos acceder uno a uno a sus elementos.

Ejemplo: El siguiente programa es una versión equivalente del programa que vimos anteriormente para contar los dígitos, las mayúsculas y las minúsculas que contenía una cadena introducida por el teclado.

Observa que la única diferencia de esta versión con la anterior está en que en la sentencia `for` utilizamos una variable `indice` para recorrer los índices de `cadena`, los cuales se obtienen de la lista generada por `range(len(cadena))`, y con cada valor asignado a `indice` accedemos al correspondiente carácter con `cadena[indice]`. Antes utilizábamos una variable `caracter` para recorrer directamente los caracteres de `cadena`, y se utilizaba directamente esta variable para referenciar a cada carácter de la cadena.

```
from string import digits, uppercase, lowercase
cadena = raw_input("Dime una cadena cualquiera: ")
num_digit = 0
num_mayus = 0
num_minus = 0
for indice in range(len(cadena)):
    if cadena[indice] in digits:
        num_digit = num_digit + 1
    elif cadena[indice] in uppercase:
        num_mayus = num_mayus + 1
    elif cadena[indice] in lowercase:
        num_minus = num_minus + 1
print "Número de dígitos:", num_digit
print "Número de letras mayúsculas:", num_mayus
print "Número de letras minúsculas:", num_minus
```

Realiza una traza de este programa con la cadena "`Aula: TD2304`", y compara su funcionamiento con el de la versión anterior.

Recorrer cadenas o listas mediante sus índices ofrece más posibilidades de tratamiento de las mismas en un bucle `for` que recorrerlas accediendo directamente a sus elementos. Acceder directamente a sus elementos obliga a visitar la secuencia completa, visitando todos y cada uno de sus elementos. Si queremos visitar sólo una parte de los elementos de la secuencia (la primera mitad, el último tercio, uno de cada dos, etc.), debemos recorrerla mediante sus índices utilizando la función `range` (con los argumentos apropiados) en un bucle `for`. Después

veremos algunos programas que recorren secuencias mediante sus índices para realizar un tratamiento adecuado al problema que deben resolver.

Ejercicio 48. Escribe un bucle utilizando `for` y `range` que imprima por pantalla todos los divisores del número 1000.

26.2. Bucles equivalentes con `for` y `while`

Algunos tipos de bucles realizados con la sentencia `while` pueden escribirse de forma equivalente y más concisa mediante la sentencia `for` y la función `range`. Todos los bucles escritos con `while` que se ajusten a la siguiente forma genérica:

```
variable = valor inicial
while variable < valor final:
    instrucciones
    variable = variable + incremento
```

siendo *valor inicial* < *valor final* y el *incremento* un número positivo, son equivalentes a:

```
for variable in range( valor inicial, valor final, incremento ):
    instrucciones
```

Similarmente, todos los bucles realizados con `while` que se ajusten a esta otra forma genérica:

```
variable = valor inicial
while variable > valor final:
    instrucciones
    variable = variable - decremento
```

siendo *valor inicial* > *valor final* y el *decremento* un número positivo, son equivalentes a:

```
for variable in range( valor inicial, valor final, -decremento ):
    instrucciones
```

Vamos ver versiones realizadas con `for` y `range` equivalentes a dos programas que anteriormente hemos escrito con `while`.

Ejemplo: En este primer ejemplo, presentamos la versión equivalente con `for` y `range` del programa que calculaba el sumatorio de todos los números desde 1 hasta `n`.

Fíjate que la condición que allí escribimos en `while` era `i <= n`, la cual es equivalente a `i <n+1`, y con ésta podemos aplicar directamente el primer esquema general de equivalencia introducido.

```
n = int(raw_input("Dime el valor de n: "))
suma_hasta_i = 0
for i in range(1, n+1, 1):
    suma_hasta_i = suma_hasta_i + i
print "Sumatorio =", suma_hasta_i
```

Observa también que, puesto que el incremento que se realizaba a la variable `i` dentro del bucle `while` era 1, podemos escribir `range(1, n+1)` en lugar de `range(1, n+1, 1)`.

Ejemplo: Como segundo ejemplo, vamos a ver la versión equivalente con `for` y `range` del programa que calculaba el factorial de un número `n` de forma decreciente. Esta versión se obtiene aplicando estrictamente el segundo esquema general de equivalencia comentado.

```
n = int(raw_input("Dime el valor de n: "))
factorial = 1
for contador in range(n, 0, -1):
    factorial = factorial * contador
print "Factorial("+str(n)+") =", factorial
```

Realiza trazas de estos programas y de las versiones anteriores escritas con `while`, utilizando los mismos valores leídos de `n`, para comparar su funcionamiento y comprobar su equivalencia en la práctica. Por otra parte, fíjate que este tipo de bucles controlados por una variable que parte de un valor inicial entero, alcanza un valor final entero y los sucesivos valores que toma crecen o decrecen una cantidad constante entera, se expresan de manera más natural y se leen con más facilidad cuando están escritos con `for` y `range` que cuando lo están con `while`: toda la información de control del bucle aparece en una sola línea.

Ejercicio 49. Anteriormente hemos visto un código Python realizado con una sentencia `while` que permitía leer una lista de tamaño preestablecido, creando primero la lista con una cierta longitud y unos valores por defecto, y modificándola después con los elementos leídos. Escribe un código equivalente con `for` y `range` para leer una lista de tamaño preestablecido.

Ejercicio 50. Escribe un programa que lea un número flotante x y un número entero positivo n, calcule el valor de

$$\sum_{i=1}^{n} x^i$$

mediante una sentencia `for` y la función `range`, y escriba el resultado por la pantalla. Además, el programa se asegurará mediante un bucle de validación de datos que el valor de n sea positivo.

Ejercicio 51. Un *lote* de puntos en el plano euclídeo es una colección formada por exactamente 50 puntos. Dados un punto p del plano, de coordenadas genéricas (x, y), y un lote, se dice que éste *está emparentado* con el punto si al menos la mitad más uno de los puntos que componen el lote están relacionados con p. Recuerda que dos puntos del plano euclídeo están relacionados si la distancia (euclídea) que los separa es menor o igual que un cierto valor umbral introducido por el usuario (ejercicio 24).

Escribe un programa que lea las coordenadas del punto p, un valor umbral y un lote de puntos, y que determine si el lote de puntos introducido está emparentado o no con p.

Toma como punto de partida el programa que resuelve el ejercicio 24 y usa una instrucción de repetición (se recomienda usar `for`) para no sólo leer los puntos que componen el lote, sino también calcular el número de esos puntos que están relacionados con p. Finalmente, al acabar el bucle que procesa los puntos del lote, introduce una instrucción de selección para sacar el mensaje apropiado por pantalla, indicando si el lote está emparentado o no con p.

Vamos a finalizar esta sección dedicada a la sentencia `for`, viendo dos programas que gestionan listas y cadenas recorriéndolas mediante sus índices. Ya comentamos que esta forma de gestión brinda más posibilidades de tratamiento que accediendo directamente a sus elementos. En uno de estos programas necesitamos recorrer tres listas al mismo tiempo en un solo bucle, y en el otro necesitamos recorrer una cadena desde sus dos extremos al mismo tiempo, para resolver los problemas planteados.

Elegir una u otra forma de recorrer las secuencias dependerá del tratamiento que haya que realizar sobre las secuencias para resolver el problema planteado. Todo tratamiento que pueda hacerse recorriendo la secuencia accediendo directamente a sus elementos, puede hacerse también recorriéndola mediante sus índices. Pero a la inversa, no es posible. Sin embargo, siempre que la solución al problema se base en visitar uno a uno, de izquierda a derecha, todos los elementos de una sola secuencia y teniendo únicamente que consultarlos, el programa resultará más natural y fácil de leer si se realiza con un bucle que acceda directamente a los elementos de la secuencia. Anteriormente ya hemos visto ejemplos de esta forma de recorrido de cadenas y listas. Repásalos y compáralos, desde el punto de vista que estamos comentando aquí, con los dos siguientes ejemplos.

Ejemplo: Vamos a escribir un programa que leerá dos vectores con las mismas dimensiones, v_1 y v_2, y dos coeficientes, α y β. A continuación, calculará un tercer vector, v_3, que será una combinación lineal de los dos anteriores, de la siguiente forma:

$$v_3 = \alpha \cdot v_1 \ + \ \beta \cdot v_2$$

Finalmente, el programa escribirá el resultado, v_3.

Como los tres vectores deben tener el mismo número de dimensiones, el programa leerá primero este dato, y después los tres vectores se crearán a partir del número de dimensiones introducido.

Los vectores se representan en el programa mediante listas. Así, tras leer el número de dimensiones, se leen dos listas de tamaño predeterminado, `v1` y `v2`, cuyos elementos serán las coordenadas de los vectores. La lectura de estas listas se realiza mediante bucles con `for` y `range` equivalentes al que vimos con `while`.

Mediante un bucle `for` se obtiene el vector `v3` como combinación líneal de `v1` y `v2`. El vector `v3` se obtiene calculando cada una de sus coordenadas `i` como combinación lineal de las correspondientes coordenadas `i` de los vectores `v1` y `v2`. Para ello, la sentencia `for` realiza un recorrido por los índices de las tres listas, que son los mismos en las tres, y cada índice `i` lo utiliza al mismo tiempo para consultar `v1[i]` y `v2[i]`, al calcular la combinación lineal, y para almacenar el resultado en `v3[i]`.

```
n = int(raw_input("Número de dimensiones de los vectores: "))
v1 = [0] * n
for i in range(n):
    v1[i] = float(raw_input("Coordenada "+str(i)+" del vector 1: "))
v2 = [0] * n
for i in range(n):
    v2[i] = float(raw_input("Coordenada "+str(i)+" del vector 2: "))
alfa = float(raw_input("Valor de alfa: "))
beta = float(raw_input("Valor de beta: "))

v3 = [0] * n
for i in range(n):
    v3[i] = alfa * v1[i] + beta * v2[i]

print "Combinación lineal de los vectores, v3 =", v3
```

Realiza una traza de este programa con $v_1 = (1, 2, 3, 4)$, $v_2 = (5, 6, 7, 8)$, $\alpha = 2$ y $\beta = 1$.

Fíjate que, en este programa, el tipo de cálculo que hay que efectuar hace necesario trabajar con las tres listas al mismo tiempo en un solo bucle. Por otra parte, la coincidencia de los índices y el hecho de que el cálculo se realice a partir de la misma coordenada `i` de las tres listas, hacen que el cuerpo de este bucle sea sencillo de interpretar.

Ejercicio 52. Modifica el programa que calcula un vector combinación lineal de otros dos previamente leídos por teclado para que, en lugar de leer inicialmente el número de dimensiones y después crear los tres vectores a partir de éste, alternativamente proceda de la siguiente manera:

1.- Lea un primer vector de un número de dimensiones variable; es decir, que el usuario detenga la introducción de coordenadas cuando desee.

2.- Exija al usuario que el segundo vector tenga las mismas dimensiones que el primero, y lea este segundo vector.

3.- Lea los coeficientes α y β.

4.- Cree el tercer vector con las mismas dimensiones que los dos primeros y calcule la combinación lineal.

5.- Muestre por pantalla el vector resultante.

Ejemplo: Realicemos ahora un programa que lea una cadena que contenga sólo letras mayúsculas y minúsculas, y determine si la cadena es un *palíndromo* o no. El programa comprobará que la cadena introducida por el usuario contiene sólo letras. Si no es así, mostrará un mensaje de error, y únicamente si la cadena es válida, comprobará si es palíndromo e indicará el resultado por pantalla.

Una cadena de letras es un palíndromo si la primera letra es igual a la última, la segunda a la penúltima, la tercera a la antepenúltima, y así sucesivamente hasta alcanzar el centro de la cadena. Por ejemplo, las cadenas “`solos`” y “`dabalearrozalazorraelabad`” son palíndromos.

Para comprobar si la cadena leída tiene sólo letras, tenemos que revisar uno a uno todos sus caracteres. Date cuenta que con sólo un carácter que no sea letra la cadena ya no es válida, y que para serlo todos los caracteres tienen que ser letras. De esta manera, para realizar esta comprobación, vamos a seguir una estrategia típica en programación: suponer inicialmente que la propiedad se cumple para

todos los elementos (todos los caracteres son letras) y realizar un bucle en busca de un elemento que desmienta esta suposición (basta con encontrar un carácter que no sea letra).

Suponer que todos los caracteres son letras implica asignar a una variable un valor que significa que esta propiedad se cumple. Y para encontrar un carácter que no sea letra, realizamos un bucle que recorre toda la cadena y para cada carácter comprobamos si éste no es letra. En este caso, cambiamos el valor de la variable por otro que significa que la propiedad no se cumple, pero en caso contrario, no se hace nada. Es decir, en este bucle sólo los caracteres que no son letras provocan el cambio de valor de la variable; los que son letras no provocan nada. Así, al acabar el bucle, si todos los caracteres eran letras, la variable no habrá cambiado de valor y significa que la propiedad se cumple; pero con sólo uno que no fuese letra, la variable sí habrá cambiado de valor y significa que la propiedad no se cumple. Observa en el programa cómo implementan esta estrategia las cuatro líneas posteriores a la lectura de la cadena.

A continuación en el programa, si la cadena resulta no ser válida escribimos el mensaje de error, pero si es válida pasamos a comprobar si es un palíndromo. Este control de errores lo realizamos mediante una selección y es similar a otros que ya hemos realizado anteriormente.

Para comprobar si la cadena es un palíndromo, adoptamos la misma estrategia que hemos seguido para comprobar si tenía sólo letras: suponer que lo es, y buscar si los caracteres desmienten esta suposición. Hemos dicho que una cadena sería un palíndromo si la primera letra es igual a la última, la segunda a la penúltima, la tercera a la antepenúltima, y así sucesivamente hasta alcanzar el centro de la cadena. Pues lo que tenemos que hacer es contrastar cada carácter desde el principio de la cadena con su correspondiente desde el final, para ver si en alguno de estos emparejamientos, los caracteres son distintos. Sólo en este caso, modificaremos el valor de la variable que representa el cumplimiento o no de la propiedad, para invalidar nuestra suposición inicial de que la cadena era palíndromo.

Fíjate que, tal como se deben emparejar los caracteres para ver si son iguales o distintos, al carácter con índice `i` en la primera mitad de la cadena le corresponde el carácter `len(cadena) - i - 1` en la segunda mitad de la cadena (`0` y `len(cadena) - 1`, `1` y `len(cadena) - 2`, etc.)

```
from string import letters
cadena = raw_input("Dime una cadena (sólo letras): ")
solo_letras = 1
for car in cadena:
    if not car in letters:
        solo_letras = 0

if solo_letras == 1:
    es_palindromo = 1
    for i in range(len(cadena)/2):
        if cadena[i] != cadena[len(cadena) - i - 1]:
            es_palindromo = 0
    if es_palindromo == 1:
        print "La cadena SI es un palíndromo."
    else:
        print "La cadena NO es un palíndromo."
else:
    print "La cadena introducida no contiene sólo letras."
```

Observa en este ejemplo que, para recorrer la cadena sólo hasta la mitad y acceder al mismo tiempo a dos de sus caracteres (comprobando si es palíndromo), lo hemos hecho mediante sus índices. Pero recorrer toda la cadena consultando uno sólo de sus caracteres cada vez (comprobando si tiene sólo letras), hemos podido realizarlo accediendo directamente a sus elementos.

Ejercicio 53. Escribe un programa que lea dos puntos, p y q, de un sistema de n dimensiones, y calcule y muestre por pantalla el punto medio entre ambos. El programa leerá primero el número de dimensiones n y después dos listas de n elementos que representarán a los puntos.

---

27. Sentencias de repetición anidadas

Al igual que ocurría con la sentencia de selección, las sentencias de repetición pueden aparecer dentro de otras sentencias de repetición. Y en general, podrán aparecer dentro de cualquier otra sentencia que admita instrucciones anidadas, y contener en el cuerpo del bucle cualquier otra sentencia. Cuando se inserta un bucle dentro de otro, se dice que son *sentencias de repetición anidadas* o *compuestas*, o también *bucles anidados*.

Como son muchas las posibles combinaciones de anidamiento de bucles y selecciones, veremos algunos ejemplos para hacernos una idea de las posibilidades de utilización de los mismos.

Ejemplo: Vamos a realizar un programa que escribe todas las tablas de multiplicar del 1 al 10.

Para ello, anidamos un bucle (al que llamamos bucle interno) dentro de otro (al que llamamos bucle externo). Como hay que escribir 10 tablas, el bucle externo se encarga de gestionar el número de tabla, `i`, que varía desde 1 hasta 10, y mediante sus instrucciones anidadas se escribe una tabla en cada iteración.

Para escribir cada tabla, una instrucción de escritura nos muestra el número `i` de la tabla que va a escribirse. Y a continuación, un bucle interno recorrerá todos los números `j`, que también van de 1 a 10, produciendo las líneas que forman la tabla `i`, mediante una sentencia de escritura anidada a este bucle interno. El programa que resulta es el siguiente.

```
for i in range(1,11):
    print "Tabla del", i
    for j in range(1,11):
        print i, "*", j, "=", i*j
```

Puedes ver a continuación un extracto de la salida por pantalla que producirá este programa. En la ventana de ejecución verás las tablas dispuestas verticalmente, una tras otra. Aquí, hemos fragmentado la salida por tablas y hemos dispuesto algunas en horizontal.

```
Tabla del 1     Tabla del 2                     Tabla del 10
1 * 1 = 1       2 * 1 = 2                       10 * 1 = 10
1 * 2 = 2       2 * 2 = 4                       10 * 2 = 20
1 * 3 = 3       2 * 3 = 6                       10 * 3 = 30
1 * 4 = 4       2 * 4 = 8                       10 * 4 = 40
1 * 5 = 5       2 * 5 = 10       ...........    10 * 5 = 50
1 * 6 = 6       2 * 6 = 12                      10 * 6 = 60
1 * 7 = 7       2 * 7 = 14                      10 * 7 = 70
1 * 8 = 8       2 * 8 = 16                      10 * 8 = 80
1 * 9 = 9       2 * 9 = 18                      10 * 9 = 90
1 * 10 = 10     2 * 10 = 20                     10 * 10 = 100
```

Fíjate en una cuestión muy importante cuando se trabaja con bucles anidados, y que ya has podido apreciar en el ejemplo anterior: *en cada iteración del bucle externo se ejecuta totalmente el bucle interno*. Al igual que una instrucción de lectura de datos o una asignación se ejecutan completamente en cada iteración de un bucle que las contenga, si un bucle está anidado a otro, el bucle interno (con sus instrucciones anidadas) constituye una acción que debe completarse en cada iteración del bucle externo.

Anteriormente, hemos visto un programa que leía una cadena, comprobaba si ésta contenía sólo letras y, si así era, comprobaba si la secuencia de letras era un palíndromo y mostraba el resultado. Para comprobar si la cadena tenía sólo letras y si era un palíndromo utilizábamos bucles que recorrían la cadena. En el próximo ejemplo vamos a incluir este programa como parte del que vamos a realizar, de manera que vamos a ver estos bucles, que realizan acciones complejas pero que ya conocemos cuales son, incluídos dentro de otro. Ahora, estos bucles serán bucles internos que completarán sus acciones en cada iteración del bucle externo.

Ejemplo: Vamos a realizar un programa que leerá repetidamente cadenas hasta que se le introduzca la cadena vacía, momento en el que finaliza el programa. Para cada cadena leída, comprobará si contiene sólo letras y, si es así, comprobará si es un palíndromo y mostrará el resultado. Además, el programa contará el número de palíndromos que aparezcan en la secuencia de cadenas que se le introduzca, y también lo mostrará por pantalla.

Para leer repetidamente cadenas utilizaremos una sentencia `while`, cuya condición expresará que la lectura debe continuar mientras la cadena no sea vacía. Anidadas a este bucle, incluímos la comprobación de si la cadena tiene sólo letras y, cuando es así, la comprobación de si es palíndromo y la escritura de su resultado. Mira el programa que resulta.

```
from string import letters

num_palindromos = 0
cadena = raw_input("Dime una cadena (sólo letras): ")
while cadena != '':
    solo_letras = 1
    for car in cadena:
        if not car in letters:
            solo_letras = 0

    if solo_letras == 1:
        es_palindromo = 1
        for i in range(len(cadena)/2):
            if cadena[i] != cadena[len(cadena) - i - 1]:
                es_palindromo = 0
        if es_palindromo == 1:
            print "La cadena SI es un palíndromo."
            num_palindromos = num_palindromos + 1
        else:
            print "La cadena NO es un palíndromo."
    else:
        print "La cadena introducida no contiene sólo letras."

    cadena = raw_input("Dime otra cadena (sólo letras): ")

print num_palindromos, "cadenas introducidas eran palíndromos."
```

Date cuenta que las sentencias anidadas al bucle `while` son prácticamente las que constituían el programa que ya vimos. Las comprobaciones que ahora se realizan para cada cadena leída son las mismas que antes se hacían para una sola cadena. Así, los bucles que realizan la comprobaciones de si la cadena tiene sólo letras y si es palíndromo son bucles internos que, al ejecutar el programa, completarán su tarea en cada iteración del bucle externo, o sea, para cada cadena leída en el bucle `while`.

Para contar el número de palíndromos, introducimos la variable `num_palindromos` que sólo se incrementará cada vez que se haya comprobado que una cadena es un palíndromo. Observa dónde está anidado el incremento de `num_palindromos`. Esta variable se inicializa a 0 antes de comenzar a leer cadenas y su valor final se escribe al acabar el bucle `while`.

A continuación, se muestra una ejecución del programa introduciéndole unos valores concretos. Realiza una traza del programa y analiza las acciones que va realizando en función de los datos introducidos y los mensajes que produce al ejecutarse.

```
Dime una cadena: salas
La cadena SI es un palíndromo.
Dime otra cadena: b+a
La cadena introducida no contiene sólo letras.
Dime otra cadena: nocion
La cadena NO es un palíndromo.
Dime otra cadena: azuza
La cadena SI es un palíndromo.
Dime otra cadena:
2 cadenas introducidas eran palíndromos.
```

Vamos a ver un último ejemplo en el que vamos a anidar un bucle de control de datos dentro de otro bucle. Así, las restantes acciones que se realizan en el bucle externo se realizan sólo con datos correctos.

Ejemplo: Este programa va a leer un número de alumnos, asegurándose de que es mayor que 0, y después va a leer las calificaciones de los alumnos, asegurándose de que cada una de ellas está entre 0 y 10. El programa acumulará las notas leídas, para finalmente calcular la nota media, y obtendrá los índices de los alumnos cuya nota es 10.

```
num_alum = 0
while num_alum <= 0:
    num_alum = int(raw_input("Dime el número de alumnos: "))
    if num_alum <= 0:
        print "El número de alumnos debe ser mayor que 0."

suma_notas = 0
indices_matriculas = []
for i in range(1,num_alum+1):
    nota = float(raw_input("Calificación del alumno "+str(i)+": "))
    while nota < 0  or  nota > 10:
        print "La calificación debe estar entre 0 y 10."
        nota = float(raw_input("Calificación del alumno "+str(i)+": "))

    suma_notas = suma_notas + nota

    if nota == 10:
        indices_matriculas = indices_matriculas + [i]

print "Nota media de la clase:", suma_notas / num_alum
print "Índices de alumnos con matrículas de honor:", indices_matriculas
```

A continuación, se muestra una ejecución del programa introduciéndole unos valores concretos. Realiza una traza del programa y analiza las acciones que va realizando en función de los datos introducidos y los mensajes que produce al ejecutarse.

```
Dime el número de alumnos: -2
El número de alumnos debe ser mayor que 0.
Dime el número de alumnos: 4
Calificación del alumno 1: 12
La calificación debe estar entre 0 y 10.
Calificación del alumno 1: 10
Calificación del alumno 2: 7.5
Calificación del alumno 3: -5
La calificación debe estar entre 0 y 10.
Calificación del alumno 3: 15
La calificación debe estar entre 0 y 10.
Calificación del alumno 3: 5
Calificación del alumno 4: 10
Nota media de la clase: 8.125
Índices de los alumnos con matrículas de honor: [1, 4]
```

Ejercicio 54. Escribe un programa que lea dos números enteros positivos n y m, calcule el valor de

$$\sum_{i=1}^{n}\sum_{j=1}^{m} i \cdot j$$

mediante dos sentencias `for` anidadas y escriba el resultado por la pantalla. Además, el programa se asegurará mediante bucles de validación de datos que los valores de los números n y m que se introduzcan sean positivos.

Ejercicio 55. Escribe un programa que lea un número entero positivo n, y calcule y escriba el sumatorio de todos los factoriales desde 1 hasta n. O sea, el programa debe calcular:

$$\sum_{i=1}^{n} i\,! \;=\; 1! \;+\; 2! \;+\; 3! \;+\; 4! \;+\; \cdots \;+\; (n-1)! \;+\; n!$$

Además, el programa se asegurará mediante un bucle de validación de datos que el valor del número n que se introduzca sea positivo.

Ejercicio 56. En el ejercicio 35 se pedía hacer un programa que fuese capaz de actuar de árbitro en el juego de *piedra, papel y tijera*. En este ejercicio tienes que:

1.- Modificar el programa que resolvía dicho problema para que se celebren partidas de exactamente 5 jugadas. El programa resultante de esta modificación debe llevar la cuenta del número de partidas ganadas por cada jugador, de tal forma que, además de mensajes comunicando el resultado parcial de cada jugada, pueda mostrar el resultado final: número de jugadas ganadas por cada jugador en la partida y número de jugadas que terminaron en empate. Se recomienda utilizar un bucle `for` para realizar esta modificación.

2.- Modificar el programa que resolvía dicho problema para que se disputen jugadas hasta que un jugador gane 5. En dicho momento el programa debe mostrar un mensaje indicando el tanteador final, además de los mensajes indicando el resultado parcial de cada jugada. Se recomienda utilizar un bucle `while` para hacer esta modificación.

28. Sesión de problemas 5

Ejercicio 57. En el ejercicio 36 de la sesión de problemas 4, utilizábamos instrucciones de selección para realizar un control de errores de los valores de `a` y `b` introducidos, y calcular el logaritmo de `a/b` *sólo* en el caso de que estos valores fuesen mayores que cero.

Vamos a realizar un programa que no sólo detecte los valores erróneos de `a` y `b` sino que, además, *vuelva* a pedirlos, tantas veces como sea necesario, mientras sean erróneos. El programa procederá a efectuar el cálculo correspondiente sólo después de haber conseguido que el usuario introduzca valores correctos. Podemos expresar en “lenguaje natural” un posible esquema de solución de la siguiente manera:

asignar un valor inicial *adecuado* para `a`
mientras el valor de `a` sea menor o igual que 0 **ejecutar**
    pedir y leer el valor de `a`
    si `a` es menor o igual que 0 **entonces**
        escribir mensaje de error en pantalla
    (en el caso en que `a` sea mayor que 0 no hacemos *nada*)
asignar un valor inicial *adecuado* para `b`
mientras el valor de `b` sea menor o igual que 0 **ejecutar**
    pedir y leer el valor de `b`
    si `b` es menor o igual que 0 **entonces**
        escribir mensaje de error en pantalla
    (en el caso en que `b` sea mayor que 0 no hacemos *nada*)
calcular el logaritmo de `a`
calcular el logaritmo de `b`
calcular el logaritmo de `a/b`
escribir el resultado en pantalla

Escribe el programa en Python que implementa este esquema.

Ejercicio 58. El siguiente programa lee del teclado las coordenadas de dos puntos del plano bidimensional. A continuación, el programa calcula la distancia euclídea entre dichos puntos y muestra el resultado por pantalla. Este programa soluciona el problema planteado en el ejercicio 18 de la sesión de problemas 2, empleando listas para representar los puntos y calculando la raíz cuadrada mediante el operador de exponenciación.

```
p = [0] * 2
p[0] = float(raw_input("Coordenada X del primer punto: "))
p[1] = float(raw_input("Coordenada Y del primer punto: "))
q = [0] * 2
q[0] = float(raw_input("Coordenada X del segundo punto: "))
q[1] = float(raw_input("Coordenada Y del segundo punto: "))

valor = 0
valor = valor + (p[0] - q[0]) ** 2
valor = valor + (p[1] - q[1]) ** 2
distancia = valor ** (1.0 / 2)
print "Distancia euclídea entre p y q:", distancia
```

Modifica este programa para que calcule la distancia euclídea entre dos puntos cualesquiera en el plano n-dimensional, donde n indica el número de coordenadas (dimensión del espacio de representación) de cada punto, y muestre el resultado por pantalla. Recuerda que la distancia euclídea entre dos puntos, p y q, de n dimensiones se calcula como:

$$\sqrt[n]{\sum_{i=1}^{n}(p_i - q_i)^2}$$

Date cuenta que puedes calcular la raíz n-ésima mediante el operador de exponenciación, de igual manera que se calcula la raíz cuadrada en el programa dado.

El nuevo programa debe leer primero el número de dimensiones n que van a tener los puntos, y después las dos listas de n coordenadas cada una, que representarán dichos puntos. A continuación, mediante un bucle calculará la suma de diferencias al cuadrado de las coordenadas i-ésimas respectivas. Al terminar el bucle, calculará la raíz n-ésima de dicha suma. Finalmente, escribirá el resultado. El programa que vimos para obtener un vector combinación lineal de otros dos te será muy útil para realizar éste.

Ejercicio 59. En el siguiente programa, la lista `coste_tramo` contiene el coste en euros de cada uno de los 20 tramos de una autopista. Así, recorrer el tramo `i` de la autopista tiene un coste de `coste_tramo[i]` euros. El programa calcula el coste de recorrer una serie de tramos consecutivos de la autopista, comenzando en el tramo `ini` y acabando en el tramo `fin`.

```
coste_tramo = [1.67, 2.43, 1.45, 1.82, 2.08, 2.16, 1.96, 1.75, 1.57, 2.01,
               2.23, 2.11, 1.95, 1.74, 2.01, 1.67, 2.05, 1.45, 2.35, 1.78]
ini = int(raw_input("Tramo de entrada a la autopista: "))
fin = int(raw_input("Tramo de salida de la autopista: "))
coste_autop = 0.0
for i in range(ini, fin+1):
    coste_autop = coste_autop + coste_tramo[i]
print "Coste del recorrido por la autopista:", coste_autop, "euros"
```

Realiza las siguientes modificaciones al programa.

1.- El programa dado no realiza un control de los valores introducidos en `ini` y `fin`. Así, si el valor de `ini` o el de `fin` excede los límites del rango de índices válidos de `coste_tramo` se producirá un error en la ejecución del programa. Añade bucles de control de los valores leídos que aseguren que, cuando el programa vaya a calcular el coste del recorrido, ambos contendrán índices válidos de `coste_tramo`.

2.- En el programa, el bucle que acumula el coste de los tramos recorridos desde `ini` hasta `fin`, sólo realiza correctamente la acumulación de costes si `ini` es menor o igual que `fin`. Esto significa que este programa

sólo permite calcular el coste de un recorrido por la autopista si éste se realiza en uno de los dos sentidos de circulación.

Amplia el programa para que permita calcular el coste de un recorrido en cualquiera de los dos sentidos de circulación. Para ello, el nuevo programa deberá comparar los valores de `ini` y `fin`. Si el primero es menor o igual que el segundo, calculará el coste del recorrido (en un sentido) igual que lo hace el programa dado. Pero en caso contrario, calculará el coste del recorrido (en sentido contrario) mediante un bucle decreciente que acumule costes desde el tramo mayor (`ini`) hasta el tramo menor (`fin`).

3.- La empresa concesionaria de la autopista ha decidido promocionar el uso de la misma realizando una oferta: por cada tres tramos recorridos el tercero es gratuito. Modifica el programa para que calcule el coste de un recorrido por la autopista sin acumular el tercero de cada tres que se utilicen. Para ello, puedes introducir en el bucle de acumulación un contador del número de tramos utilizados, y comprobar su valor para decidir en cada iteración si acumulas el coste del tramo correspondiente o no.

Ejercicio 60. Vamos a utilizar las listas de Python para representar conjuntos matemáticos. En el siguiente programa, primero se leen los elementos de dos conjuntos en dos listas, y después simplemente se escriben las dos listas que representan estos conjuntos. Fíjate que la lectura de los conjuntos es similar a la lectura de listas que conocemos, pero introduciendo una comprobación: solamente si el elemento recién leído no está ya en la lista, se añade a ésta. Así, conseguimos que en las listas no haya elementos repetidos, con lo que representan adecuadamente los conjuntos.

```
cto1 = []
elem = int(raw_input("Dime un elemento del conjunto 1: "))
while elem >= 0:
    if not elem in cto1:
        cto1 = cto1 + [elem]
    elem = int(raw_input("Dime otro elemento del conjunto 1: "))

cto2 = []
elem = int(raw_input("Dime un elemento del conjunto 2: "))
while elem >= 0:
    if not elem in cto2:
        cto2 = cto2 + [elem]
    elem = int(raw_input("Dime otro elemento del conjunto 2: "))

print "Conjunto 1:", cto1
print "Conjunto 2:", cto2
```

Amplia el programa para que:

1.- Obtenga un nuevo conjunto que represente la unión de los dos conjuntos leídos y la muestre por pantalla. El conjunto unión contendrá a todos los elementos de ambos conjuntos, pero sin elementos repetidos.

2.- Obtenga un nuevo conjunto que represente la intersección de los dos conjuntos leídos y la muestre por pantalla. El conjunto intersección contendrá sólo a los elementos que aparecen en los dos conjuntos, pero sin repetirlos.

3.- Obtenga un nuevo conjunto que represente la diferencia de los dos conjuntos leídos, o sea, el primero menos el segundo, y la muestre por pantalla. El conjunto diferencia contendrá sólo a los elementos que están en el primero pero no en el segundo.

Combina la utilización de bucles, para recorrer los elementos de los conjuntos, con selecciones y el operador `in` para implementar estas tres operaciones sobre conjuntos.

Ejercicio 61. Realiza un programa que permita validar una clave, de manera que pida repetidamente introducirla mientras no cumpla las siguientes tres restricciones:

- tener al menos 6 caracteres de longitud,
- tener al menos 3 letras (mayúsculas o minúsculas), y

- tener al menos 2 dígitos.

El programa informará sobre la restricción que incumple la clave cada vez que no sea válida, y cuando ésta se acepte, dará un mensaje confirmando su validez.

Repasa el programa dado en un ejemplo para contar el número de dígitos, mayúsculas y minúsculas en una cadena. Repasa también el programa que resolvía el ejercicio 38 de la sesión de problemas 4. Ambos te serán de gran ayuda para realizar éste.

Ejercicio 62. Escribe la salida que producirá en la ventana de ejecución el siguiente programa Python, el cual coincide básicamente con el planteado en el ejercicio 40 de la sesión de problemas 4, pero empleando un bucle `for` para recorrer la lista de películas en lugar de ir una por una. Recuerda que tienes que justificar adecuadamente la respuesta.

```
pelis = ["The Wicker Man", "El Laberinto del Fauno", "Gracias por Fumar",
         "La Dalia Negra", "Infiltrados"]
puntos = [2,8,8,5,8]
buenas = []
regulares = []
malas = []
for i in range(len(pelis)):
  if puntos[i] >= 0 and puntos[i] <= 3:
    malas = malas + [pelis[i]]
  elif puntos[i] >= 4 and puntos[i] <= 6:
    regulares = regulares + [pelis[i]]
  else:
    buenas = buenas + [pelis[i]]
print "Pelis malas:"
for peli in malas:
  print "     " + peli
print
print "Pelis regulares:"
for peli in regulares:
  print "     " + peli
print
print "Pelis buenas:"
for peli in buenas:
  print "     " + peli
print
```

Ejercicio 63. Escribe la salida que producirá en la ventana de ejecución el siguiente programa Python. Observa que el programa no solicita al usuario que introduzca ningún dato. Debes indicar *exactamente* y sin que haya margen para la duda, la salida del programa: no uses abreviaturas ni nada por el estilo. **Importante**: *debes justificar brevemente* la respuesta.

```
claves = ["Pésima", "Mala", "Regularcilla", "Buena", "Muy Buena",
          "Extraordinaria, maravillosa"]
pelis = ["Million Dollar Baby", "11:14 Destino Fatal", "El Hundimiento",
         "Entre Copas", "Largo Domingo de Noviazgo", "Mar Adentro",
         "El Reino de los Cielos", "Primer", "Intermission", "Alejandro",
         "Un Canguro Superduro", "Independence Day", "Madagascar"]
puntos = [5,3,4,3,5,4,1,4,2,1,0,0,2]
for i in range(1,len(pelis)):
    m = puntos[i-1]
    pos = i-1
    for j in range(i,len(pelis)):
```

```
        if puntos[j] > m:
            m = puntos[j]
            pos = j
    if pos <> i-1:
        aux = puntos[i-1]
        aux2 = pelis[i-1]
        puntos[i-1] = puntos[pos]
        pelis[i-1] = pelis[pos]
        puntos[pos] = aux
        pelis[pos] = aux2
for i in range(len(pelis)):
    print "La película", pelis[i], "es", claves[puntos[i]]
```

Autoría:

Estos apuntes han sido editados por el profesor Juan Carlos Amengual Argudo, y su redacción supervisada por él mismo y el profesor Roberto Solana Montero, ambos profesores de la 509, a partir del material elaborado por el profesor Antonio Castellanos López, que fue profesor de esta asignatura hasta el curso 2002/2003.