Ficha 6

Estructuras Repetitivas: El Ciclo *while*

1.] Introducción.

Todos los problemas y ejercicios analizados hasta ahora tenían, en última instancia, un elemento en común: en todos los casos se trataba de problemas cuya solución básica y más elemental nunca implicó *repetir la ejecución de un bloque de instrucciones*. Los algoritmos diseñados para resolver esos problemas eran de *naturaleza no repetitiva.*

Sin embargo, basta observar con cuidado a nuestro alrededor para notar que la mayor parte de las tareas que realizamos a diario incluyen algún tipo de *repetición de acciones* en cierto momento: Quien hable por teléfono debe *repetir* la operación de marcar un dígito en el teclado telefónico (aunque no siempre necesariamente el mismo dígito), tantas veces como dígitos tenga en total el número con el que se quiere comunicar. Quien tenga la mala fortuna de pinchar un neumático mientras va manejando su automóvil, debe cambiar la goma pinchada, pero eso implica *repetir* la operación de desajustar una tuerca y retirarla, tantas veces como tuercas tenga esa goma. El sólo hecho de leer un libro implica *repetir* la acción de dar vuelta una página, tantas veces como páginas se alcance a leer. Un empleado administrativo que reciba la orden de llenar formularios mientras dure su turno de trabajo, está realizando una tediosa *tarea repetitiva*: toma un formulario de la pila de formularios en blanco, lo llena siguiendo ciertas pautas, y lo deja a un lado en otra pila de formularios ya completados. Luego repite la secuencia y así prosigue hasta que su turno termina.

Los ejemplos citados en el párrafo anterior son muestras de *situaciones repetitivas cotidianas*. Pero el hecho es que cuando se desarrollan programas para una computadora estas situaciones algorítmicas repetitivas son también muy comunes. De hecho, *lo más común* es enfrentar problemas cuya solución requiera alguna clase de repetición en la ejecución de ciertas instrucciones para llegar al resultado deseado, o al menos para llegar con mayor eficiencia a dicho resultado[1].

En ese sentido, en programación un *ciclo* es una *instrucción compuesta* que permite la repetición controlada de la ejecución de cierto conjunto de instrucciones en un programa, determinando si la repetición debe detenerse o continuar de acuerdo al valor de cierta condición que se incluye en el ciclo [1]. Los ciclos también se designan como *estructuras repetitivas*, o bien como *instrucciones repetitivas*. Básicamente, en todo lenguaje de programación un ciclo consta de dos partes:

[1] Si de repetir acciones se trata, no podemos menos que referir la fantástica película *Groundhog Day* (en español conocida como *Hechizo del Tiempo*) del año 1993, dirigida por *Harold Ramis* y protagonizada por *Bill Murray* y *Andie MacDowell*. Un locutor de televisión engreído y cascarrabias fue condenado a revivir una y otra vez el mismo día que más odiaba, sin que lo sepa nadie aparte de él mismo, y sólo pudo escapar a ese castigo cuando finalmente cambió y se convirtió en una buena persona… después de miles y miles de repeticiones de la misma jornada y los mismos eventos. Impecable. Extraordinaria. Quizás lo mejor de *Bill Murray*.

- El *bloque de acciones* o *cuerpo del ciclo*, que es el conjunto de instrucciones cuya ejecución se pide repetir.
- La *cabecera del ciclo*, que incluye una *condición de control* y/o elementos adicionales en base a los que se determina si el ciclo continúa o se detiene.

Las instrucciones repetitivas o ciclos son implementados por los diversos lenguajes en formas que varían ligeramente de un lenguaje a otro. El lenguaje Python implementa dos tipos de ciclos mediante instrucciones específicas y muy flexibles, designándolos respectivamente como ciclo *for* y como ciclo *while* [2]. En esta Ficha 6 analizaremos la forma de uso y aplicación del ciclo *while*, y en la Ficha 7 veremos la forma de trabajo y aplicación del ciclo *for*.

2.] El ciclo *while* en Python.

Hemos sugerido que en muchas situaciones necesitaremos desarrollar programas que ejecuten en forma repetida un conjunto de instrucciones. Tomemos como ejemplo el mismo programa básico que hemos mostrado en la Ficha 5 para introducir el tema de las variables de conteo, en el que se cargaban tres números y se pedía determinar cuántos números negativos se ingresaron. El programa básico que se mostró en la Ficha 5 era el siguiente:

Figura 1: Uso básico de un contador (tomado de la Ficha 5).

```
__author__ = 'Catedra de AED'


a = 0
num = int(input('Ingrese un número: '))
if num < 0:
    a = a + 1

num = int(input('Ingrese otro: '))
if num < 0:
    a = a + 1

num = int(input('Ingrese otro: '))
if num < 0:
    a = a + 1

print('Cantidad de negativos cargados:', a)
```

Si bien el programa citado funciona perfectamente bien, encierra dos grandes limitaciones:

1. ¿Qué pasaría si en lugar de sólo *tres números* tuviéramos que cargar *mil números*? Resultaría muy poco práctico desarrollar un programa con mil instrucciones de lectura y mil instrucciones condicionales. ¿Y si en lugar de *mil números* fueran *diez mil*, o *cien mil*, o *un millón*?
2. ¿Qué pasaría además, *si no supiésemos la cantidad exacta de números que se deben cargar* al momento de desarrollar el programa? Por ejemplo, es común que el enunciado o requerimiento no exprese cuántos números vendrán y en cambio indique que *se carguen* números *hasta que en algún momento se ingrese el cero*. Intente resolver este problema ***solamente*** en base a estructuras secuenciales y condicionales y descubrirá que no es posible.

Las *instrucciones repetitivas* o *ciclos* están previstas para que el programador pueda resolver situaciones como estas con sencillez. Como dijimos, Python provee dos tipos de ciclos: el *ciclo while* y el *ciclo for*.

En muchas ocasiones necesitamos plantear un ciclo que ejecute en forma repetida un bloque de acciones pero sin conocer previamente la cantidad de vueltas a realizar. Para estos casos la mayoría de los lenguajes de programación, y en particular Python, proveen un ciclo designado como *ciclo while*, aunque como veremos, puede ser aplicado sin ningún problema también situaciones en las que se conoce la cantidad de repeticiones a realizar.

Como todo ciclo, un *ciclo while* está formado por una *cabecera* y *un bloque o cuerpo de acciones*, y trabaja en forma general de una manera muy simple. La *cabecera* del ciclo contiene una *expresión lógica* que es evaluada en la misma forma en que lo hace una instrucción condicional *if*, pero con la diferencia que el *ciclo while* ejecuta su bloque de acciones en *forma repetida* siempre que la expresión lógica arroje un valor verdadero. Así como un *if* hace una *única* evaluación de la expresión lógica para saber si es verdadera o falsa, un *ciclo while* realiza *múltiples* evaluaciones: cada vez que termina de ejecutar el bloque de acciones vuelve a evaluar la expresión lógica y si nuevamente obtiene un valor verdadero repite la ejecución del bloque y así continúa hasta que se obtenga un falso [3].

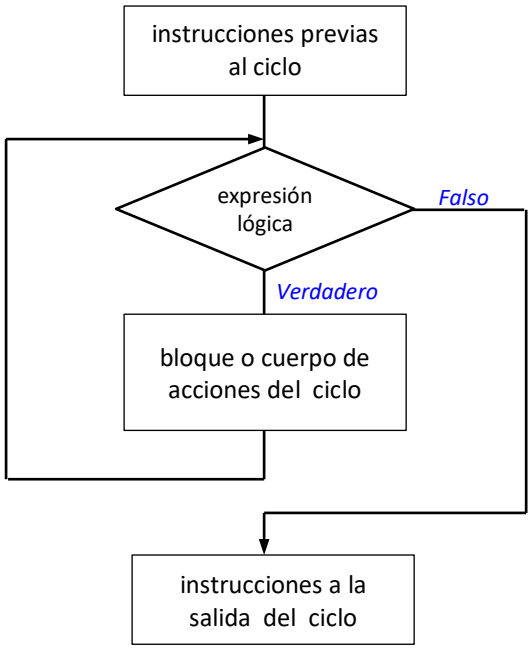
En Python el *ciclo while* se escribe básicamente en la forma siguiente:

```
while expresión_lógica:
    bloque de acciones
```

El programador no debe escribir ninguna instrucción especial dentro del bloque de acciones para indicar que el ciclo debe volver a la cabecera y evaluar nuevamente la expresión lógica: ese retorno es automático e implícito.

La lógica esencial de funcionamiento de un *ciclo while* se refleja muy bien en la *forma clásica* de *graficarlo* en un diagrama de flujo (ver *Figura 2*). La expresión lógica conforma la cabecera del ciclo, y su valor determina si el ciclo continúa ejecutando el bloque de acciones, o se detiene. En la primera oportunidad que esa expresión lógica sea falsa, el ciclo se detendrá y el programa continuará ejecutando las instrucciones que figuren a la salida del ciclo. Pero si la expresión lógica es verdadera, ejecutará el bloque y luego automáticamente volverá a chequear la expresión lógica de la cabecera, repitiendo el esquema.

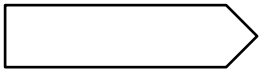
Figura 2: Diagrama de flujo de un ciclo while (diagramación clásica)



Note especialmente que si al evaluar la expresión lógica *por primera vez* esta fuese falsa, entonces el bloque de acciones no sería ejecutado y el programa continuaría con las instrucciones a la salida del ciclo. Debido a este comportamiento, se dice que el *ciclo while* es un ciclo de *tipo [0, N]*: el bloque de acciones puede llegar a ejecutarse entre 0 y *N* veces, siendo *N* un número entero positivo normalmente desconocido (pero que indica un número finito de repeticiones) [1].

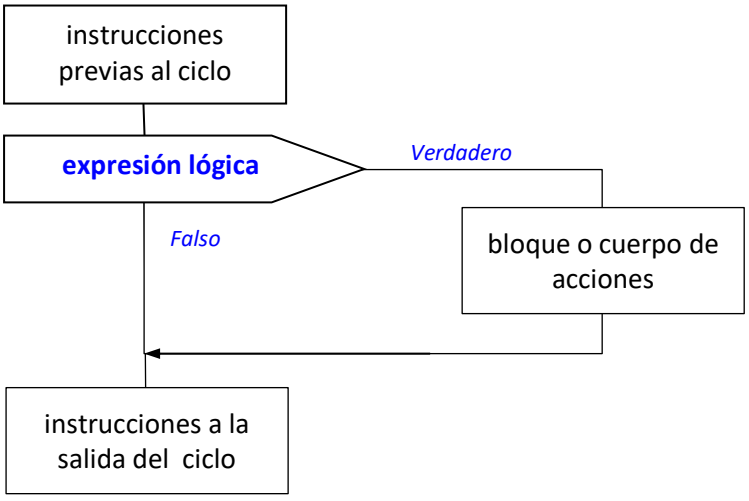
Si bien la forma clásica de graficar un *ciclo while* (en rigor, un ciclo *[0, N]*) es muy clara en cuanto a su lógica, se puede plantear alternativamente un gráfico basado en otro símbolo no estándar pero específico para el planteo de ciclos, como es el siguiente:

Figura 3: Símbolo alternativo (no estándar) para graficar un ciclo en un diagrama de flujo.



El diagrama completo alternativo [1] sería el que se muestra en la *Figura 4*. La *forma alternativa* (o *indentada*) de diagramar el ciclo no es tan clara ni tan directa en cuanto a la lógica, pero mantiene coherencia con el tipo de símbolo usado para graficar un ciclo (distinguiéndolo sin ambigüedades del rombo usado para graficar una instrucción condicional) y mantiene visualmente el esquema de indentación que luego será usado al escribir el programa. *Pero queda claro que tanto los estudiantes como los profesores pueden optar por la forma que deseen*: recuerde que el diagrama de flujo es una *técnica auxiliar* de representación de algoritmos. Si en el contexto en que se esté trabajando se necesitase mantener consistencia en cuanto al tipo de diagrama a usar, entonces los miembros de ese equipo simplemente deberán acordar a qué reglas atenerse y qué tipo de esquema gráfico aplicar.

Figura 4: Diagrama de flujo de un ciclo while (diagramación alternativa o indentada)



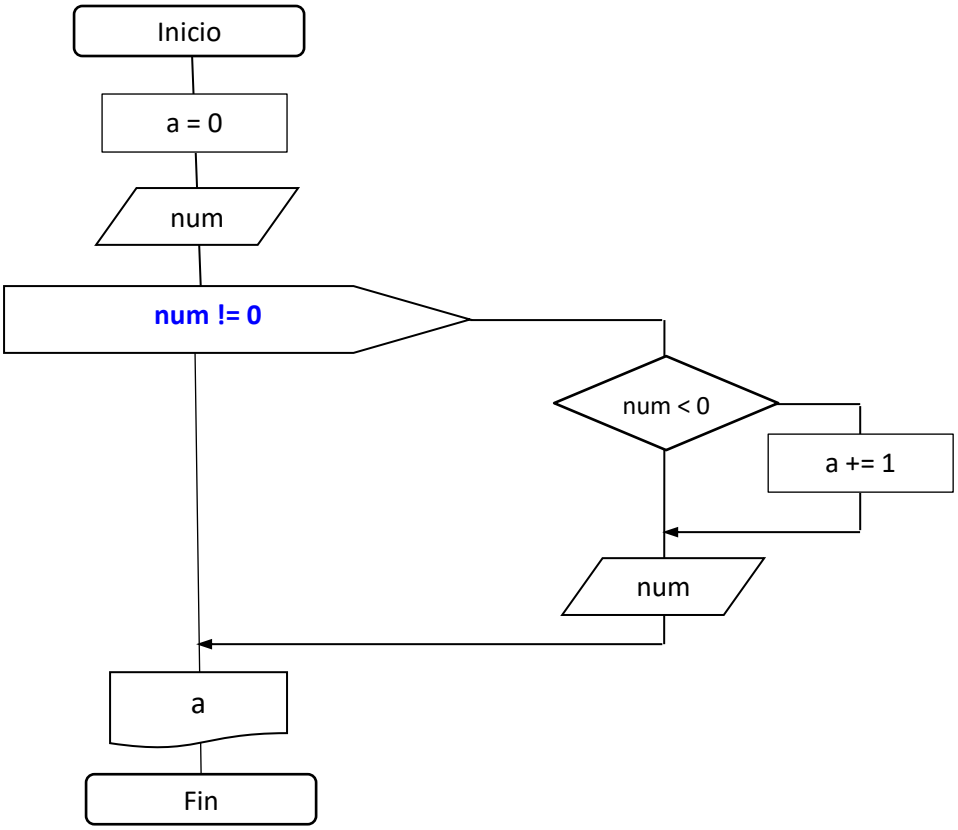
Volviendo al problema de cargar un conjunto de números y determinar cuántos eran negativos, supongamos ahora que desconocemos cuántos serán los números a cargar, pero que en su lugar sabemos que el usuario *ingresará un cero cuando ya no le queden números por ingresar*. Este planteo (como veremos) sería imposible de resolver (al menos en forma

simple y directa) mediante un *ciclo for* en Python[2] porque se desconoce la cantidad de repeticiones que debería hacer ese ciclo.

En un caso como este, el *ciclo while* se aplica con naturalidad. Si en el enunciado del problema no se indica la cantidad de datos a procesar o la cantidad de repeticiones a realizar, pero en cambio se indica alguna condición que especifique cómo debe cortar el ciclo, entonces esa condición se designa en forma general como *condición de corte*, y en base a ella debe deducirse y escribirse la cabecera del *ciclo while* para controlar su avance.

El programador sólo debe recordar que un *ciclo while* en Python, repite la ejecución del bloque si la expresión lógica de su cabecera es verdadera, y corta si es falsa. Por lo tanto, para plantear la cabecera del ciclo debe pensar en la siguiente pregunta general: *¿qué es lo que debe ser cierto para que el ciclo **continúe**?* Y la expresión lógica que obtenga es la que debe usar en la cabecera del ciclo. Por lo tanto, si se indica que la carga de datos *terminará* cuando se ingrese un cero, entonces *lo que debe ser cierto para que el ciclo **continúe*** es que el número ingresado ***no*** sea cero. La *Figura 5* muestra el diagrama de flujo propuesto.

Figura 5: Diagrama de flujo (forma indentada) del problema de contar negativos usando doble lectura.



En base a este diagrama mostramos ahora el programa completo para el conteo de los números negativos, replanteado para usar un ciclo *while*, sin conocer cuántos números vendrán pero sabiendo que al ingresar un 0 la carga debe terminar:

[2] Y algo similar ocurriría en otros lenguajes en los que el *ciclo for* está diseñado para desplegar una cantidad exacta de repeticiones, como en el lenguaje *Pascal*. Sin embargo, note que en otros lenguajes (como *C*, *C++* o *Java*) la instrucción *for* se implementa en forma tan amplia, que no hay problema en aplicarla en este tipo de situaciones y en cualquier otra que requiera simplemente un ciclo.

```
__author__ = 'Catedra de AED'


a = 0
num = int(input('Ingrese un numero (o cero para terminar): '))
while num != 0:
    if num < 0:
        a += 1
    num = int(input('Ingrese otro numero (o cero para terminar): '))

print('Cantidad de negativos cargados:', a)
```

El programa utiliza un *ciclo while* para la carga y procesamiento de los números. Por otra parte, puede notarse que se carga un valor de *num* antes de comenzar el ciclo, y otro valor de *num* inmediatamente antes de finalizar el bloque de acciones. La primera lectura se realiza porque de otro modo la variable *num* podría no existir o podría tener un valor residual cuando se verifique la expresión lógica de control del *ciclo while* por primera vez, y podría esa verificación dar resultados no deseados. Cuando en el ciclo se pregunta si *num* es diferente de cero, *num debe existir y tener siempre un valor controlado por el programador*. La segunda lectura, efectuada dentro del bloque de acciones al final del mismo, se efectúa porque en caso de no hacerla la variable *num* seguiría teniendo el *mismo valor* que se cargó en la primera lectura, y eso provocaría que la condición de control se vuelva a evaluar en verdadero, comenzando una serie infinita de repeticiones para el mismo valor.

En un ciclo bien planteado, el valor de la variable o variables que se usan en la expresión lógica de control debe cambiar adecuadamente dentro del bloque de acciones, pues de otro modo, si el valor nunca cambia, el ciclo repetirá una y otra vez, entrando en una situación de *ciclo infinito*. El esquema anterior, que incluye dos lecturas para evitar los problemas descriptos, es de uso muy común cuando se procesan conjuntos de valores usando un *ciclo while*, sin conocer cuántos valores vendrán, y se designa como ***proceso de carga por doble lectura*** [1].

Debe el programador tener especial cuidado cuando usa ciclos, para evitar caer en *ciclos infinitos*. Si en el ejemplo anterior se elimina la segunda lectura, se cae irremisiblemente en un ciclo infinito. Por otra parte, el ciclo infinito podría provocarse en situaciones donde no haya necesariamente lectura por teclado de la variable de control. Por ejemplo, el siguiente ciclo *está mal planteado*, debido a que la variable *c* empieza valiendo uno (por asignación), pero nunca cambia luego de valor:

```
c = 1
while c != 0:
    x = int(input('Ingrese un valor: '))
```

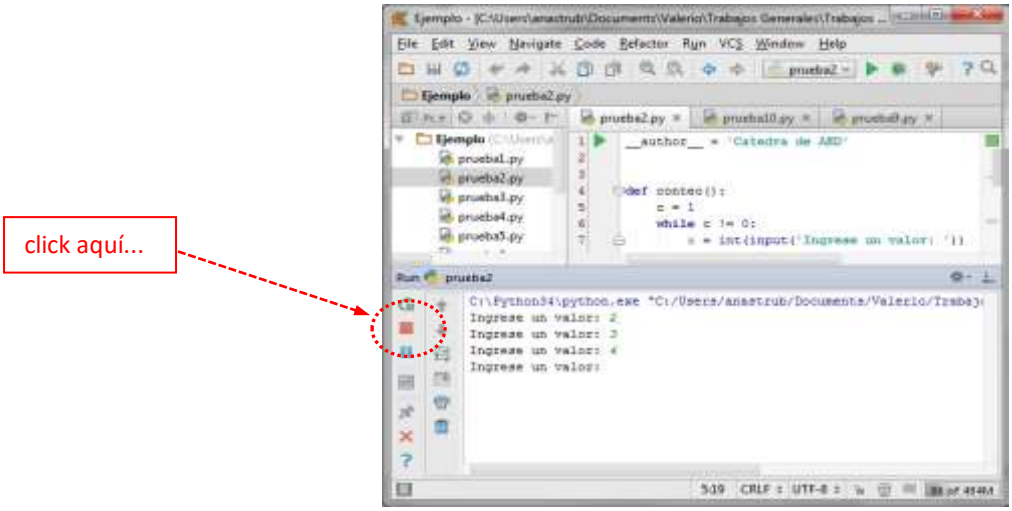
En este caso se produce un error muy común: se usa una variable en la condición de control, pero dentro del bloque de acciones del ciclo se cambia el valor *de otra* (*u otras*) variable/s. Este segmento de programa provocará que aparezca en pantalla un mensaje solicitando cargar un valor, se cargará ese valor, y luego se pedirá otro, y luego otro más, y así sucesivamente, sin terminar nunca el programa.

¿Qué hacer si un programa entra en un ciclo infinito? La manera de interrumpir un programa (ya sea porque por alguna razón no pueda finalizar normalmente o simplemente porque el programador desea interrumpirlo), suele consistir en una combinación de teclas que fuerza al programa a finalizar. En nuestro caso, si el programa se ejecuta desde *PyCharm*, la

combinación de teclas a usar es: *<Ctrl> + <F2>*. Es decir, se presiona la tecla *Control* (ubicada normalmente a los lados de la barra espaciadora del teclado), y *sin soltarla* se presiona *también* la tecla *F2* (la tecla de *Función 2,* por lo general ubicada en la primera fila de arriba del teclado). Esta operación suele funcionar sin problemas en *PyCharm*, cancelando el programa. Por cierto, es siempre una buena idea *grabar* los programas *antes* de intentar ejecutarlos...

En algunas ocasiones si el programa entra en ciclo infinito o pareciera estar *clavado* (aparentemente funcionando pero sin respuesta alguna al usuario) y nada funciona para detenerlo, no queda otro remedio que intentar interrumpirlo desde el sistema operativo (usando el *administrador de tareas* o similar) o incluso apagar el equipo y volverlo a encender, pero esta operación implica que si el programa no fue grabado, se perderá todo lo que el programador haya escrito. En *PyCharm* también se puede intentar interrumpir el programa usando el mouse: a la izquierda del panel de salida de *PyCharm*, apunte con el mouse al pequeño ícono en forma de *cuadrado de color rojo*, y haga *click* sobre el. Esto equivale a pulsar la combinación *<Ctrl> + <F2>* y el programa en curso se interrumpirá:

Figura 6: Interrupción de un programa en PyCharm.



Por supuesto, podemos utilizar un *ciclo while* para resolver problemas donde ***sí*** se conoce de antemano la cantidad de iteraciones a realizar, tal como lo hubiéramos hecho con un *ciclo for*, aunque al costo de un poco más de trabajo ya que, como veremos, el *ciclo for* se plantea de forma más automática mientras que en el *ciclo while* tendremos que hacer en forma manual el conteo de vueltas.

Por ejemplo, el siguiente script Python usa un *ciclo while* para mostrar los primeros diez múltiplos de un número *x* que se carga por teclado:

```
x = int(input("Ingrese un número: "))
i = 1
while i <= 10:
    print(x, " * ", i, " = ", x*i)
    i += 1
```

En este caso, la variable *i* es un contador que sirve para contar la cantidad de vueltas del ciclo y a la vez para realizar el cálculo de cada multiplicación. Antes de que comience el ciclo se le da el valor 1 con lo que la expresión lógica de control del ciclo entrega un valor verdadero y se ejecuta por primera vez el bloque de acciones del ciclo. En la primera

instrucción de ese bloque se calcula la multiplicación y se muestra el resultado y en la segunda la variable *i* se incrementa en una unidad justo antes de que la expresión lógica de control del ciclo vuelva a evaluarse y esto es lo que permite contar las repeticiones del ciclo. Cuando el valor del contador *i* llegue a valer 11, la expresión lógica de control del ciclo será falsa y el ciclo se detendrá.

Como ya se mencionó antes, el *ciclo while* es lo suficientemente versátil como para poder aplicarse en cualquier situación que requiera repetir la ejecución de un bloque de acciones, aunque el *ciclo for* que veremos se adapte mejor a situaciones en las que se conoce la cantidad de repeticiones a realizar. El programador decidirá cuál de ambas herramientas es la que mejor le resulte según el problema que esté analizando.

A modo de ejemplo de aplicación, consideremos el siguiente problema sencillo (que además servirá para volver a aplicar el concepto de *variable centinela* o *variable bandera*):

Problema 15.) *Cargar por teclado una lista de números enteros que irán llegando uno a uno, y que representan cantidades mensuales de automóviles nuevos vendidos en el país durante cierto período. Para indicar que la carga de datos debe finalizar, se ingresará el valor -1 (menos uno) (note que el valor 0 (cero) puede ser un dato valido: es perfectamente posible que no haya habido ventas en un mes determinado). Se pide:*

a. *Determinar cuántas de esas cantidades fueron mayores o iguales que 0 pero menores que 10000 unidades, cuántas fueron mayores o iguales a 10000 pero menores que 15000, y cuántas fueron mayores o iguales que 15000.*
b. *Determinar la cantidad total de automóviles nuevos que se vendieron en el país.*
c. *Determinar si se ingresó al menos una cantidad igual a 0 o no. Informe con un mensaje simple en pantalla.*

Discusión y solución: Este es un problema típico de procesamiento de una sucesión de números enteros con fines estadísticos. Como no sabemos cuántos números entrarán como dato, pero sabemos que al cargar un -1 se debe terminar el proceso, entonces podemos usar un *ciclo while* y un esquema de *carga por doble lectura*.

El punto a) solicita contar cuántas cantidades hay en cada uno de tres posibles intervalos de valores, y para eso necesitaremos tres contadores (que llamaremos *c1*, *c2* y *c3*).

El punto b) en última instancia pide *acumular* todos los números cargados y por eso en cada vuelta usaremos un acumulador (que llamaremos *t*) para ir sumando todos los números que se ingresen.

Y para cumplir con el punto c), usaremos una bandera (que llamaremos *ok*) que nos permitirá dejar marcado si se cargó al menos una cantidad igual a 0.

El programa completo se muestra a continuación, y luego del mismo haremos algunas observaciones:

```
__author__ = 'Catedra de Algoritmos y Estructuras de Datos'

# inicialización de contadores, acumuladores y banderas...
c1, c2, c3, t = 0, 0, 0, 0
ok = False

# proceso de carga por doble lectura...
# ... hacer la primera lectura...
cant = int(input('Ingrese cantidad vendida (con -1 termina el proceso): '))
```

```
while cant != -1:

    # punto a): chequear en qué intervalo está cada cantidad y contar...
    if 0 <= cant < 10000:
        c1 += 1
    elif 10000 <= cant < 15000:
        c2 += 1
    elif cant >= 15000:
        c3 += 1

    # punto b): acumular cada cantidad...
    t += cant

    # punto c): chequear si cant es 0, y marcar con un flag...
    if cant == 0:
        ok = True

    # hacer la segunda lectura...
    cant = int(input('Ingrese otra cantidad (con -1 termina): '))

# visualización de resultados... punto a)
print()
print('Cantidad de valores >= 0 pero < 10000:', c1)
print('Cantidad de valores >= 10000 pero < 15000:', c2)
print('Cantidad de valores >= 15000:', c3)

# visualización de resultados... punto b)
print('Cantidad total de vehículos vendidos:', t)

# visualización de resultados... punto c)

# recuerde que lo que sigue es equivalente a if ok == True:
if ok:
    print('Se registró al menos una cantidad de ventas igual cero')
else:
    print('No se registró ninguna cantidad de ventas igual cero')
```

La lógica general es bastante directa: a través del ciclo *while* se procede a cargar por teclado los valores de las cantidades vendidas usando para ello la variable *cant* y un mecanismo de *doble lectura*. El ciclo controla que el número ingresado en *cant* sea diferente de -1 (que es el valor de corte). Dentro del ciclo, cada valor de *cant* se controla con un esquema de condiciones anidadas para saber en qué intervalo se encuentra ese valor y así poder contarlo en alguno de los tres contadores *c1*, *c2* o *c3*. Inmediatamente luego el valor de *cant* simplemente se suma en el acumulador *t* para ir obteniendo así el total general de vehículos vendidos:

```
# inicialización de contadores, acumuladores y banderas...
c1, c2, c3, t = 0, 0, 0, 0
ok = False

# proceso de carga por doble lectura...
# ... hacer la primera lectura...
cant = int(input('Ingrese cantidad vendida (con -1 termina el proceso): '))
while cant != -1:

    # punto a): chequear en qué intervalo está cada cantidad y contar...
    if 0 <= cant < 10000:
        c1 += 1
    elif 10000 <= cant < 15000:
```

```
        c2 += 1
    elif cant >= 15000:
        c3 += 1

    # punto b): acumular cada cantidad...
    t += cant
```

Para determinar si alguno de los valores cargado en *cant* en alguna de las vueltas del ciclo fue igual a cero, se usa una bandera que hemos llamado *ok*. Antes del ciclo, esa bandera se inicializa con el valor *False*, para indicar que hasta ese momento no se ha detectado la carga de un valor igual a 0 en *cant*. Si en alguna iteración del ciclo se detecta la carga de un 0 en *cant*, el valor del flag *ok* se vuelve a *True* y se lo deja en ese estado hasta el final:

```
ok = False

# proceso de carga por doble lectura...
# ... hacer la primera lectura...
cant = int(input('Ingrese cantidad vendida (con -1 termina el proceso): '))
while cant != -1:

    # ...

    # punto c): chequear si cant es 0, y marcar con un flag...
    if cant == 0:
        ok = True

    # hacer la segunda lectura...
    cant = int(input('Ingrese otra cantidad (con -1 termina): '))
```

Al finalizar el ciclo (por la carga del valor -1) se muestran los valores de los tres contadores y el del acumulador (note la llamada a *print()* sin pasarle parámetros, que se hace para simplemente dejar una línea en blanco en la salida):

```
# visualización de resultados... punto a)
print()
print('Cantidad de valores >= 0 pero < 10000:', c1)
print('Cantidad de valores >= 10000 pero < 15000:', c2)
print('Cantidad de valores >= 15000:', c3)

# visualización de resultados... punto b)
print('Cantidad total de vehículos vendidos:', t)
```

Finalmente, para se controla el valor final del flag *ok*: si ese valor es *True*, significa que al menos una vez se cargó el valor 0 en la variable *cant* durante el ciclo. Pero si el valor de *ok* es *False* significa que *ok* mantuvo el valor inicial que se le asignó al comenzar el programa y por lo tanto, nunca se cargó un cero:

```
# visualización de resultados... punto c)
if ok:
    print('Se registró al menos una cantidad de ventas igual cero')
else:
    print('No se registró ninguna cantidad de ventas igual cero')
```

Veamos ahora otro ejemplo de aplicación en el que también se procesa una secuencia números, pero algo más complejo:

Problema 16.) *Cargar por teclado un conjunto de números enteros, uno a uno. **La carga sólo debe terminar cuando se ingrese el cero**. Determine si los números que se ingresaron eran todos positivos o todos negativos (sin importar en qué orden hayan entrado). Por ejemplo, la secuencia {8, 4, 3, 7} cumple con la consigna indicada (son todos positivos). La secuencia {-2, -1, -5} también cumple (son todos negativos), pero esta otra secuencia {10, -8, 2, 12} no cumple (hay números de distinto signo entremezclados). Si todos los números eran efectivamente del mismo signo, muestre también la suma de esos números (no mostrar la suma si había números de signos diferentes entremezclados).*

Discusión y solución: Este es un problema típico de procesamiento de una sucesión de números enteros. Como no sabemos cuántos números entrarán como dato, pero sabemos que al cargar un 0 se debe terminar el proceso, entonces podemos usar un *ciclo while* y un esquema de carga por doble lectura. Y como además finalmente se pide acumular los números cargados, en cada vuelta usaremos un acumulador para ir sumando los números que se ingresen (por ahora no nos preocuparemos del hecho de que esos números podrían ser de signos opuestos).

Si llamamos *actual* a la variable de carga y *suma* al acumulador, el ciclo podría comenzar planteándose así:

```
# acumulador para los números del mismo signo...
suma = 0

# cargar el primero...
actual = int(input('Cargue un número entero (con 0 corta): '))
while actual != 0:

    # acumular el número ingresado
    suma += actual

    # cargar el siguiente...
    actual = int(input('Cargue otro (con 0 corta): '))
```

Para controlar si los números son o no del mismo signo, podemos aplicar una técnica muy común entre los programadores para dejar algún tipo de señal o aviso lógico, que le indique al programador si ha ocurrido o no algún suceso en particular.

Procederemos para eso de la siguiente manera: sea la variable booleana *ok* que comenzaremos asignando con el valor *True* antes de comenzar el ciclo de carga. La idea es que el valor de esa variable indique lo que hasta ese momento se sabe respecto de los signos de los números cargados: usamos el valor *True* para registrar que hasta ese momento, todos los números que se hayan visto eran efectivamente del mismo signo (lo cual es efectivamente así si sólo se cargó uno). Si a lo largo del ciclo, se descubre que los números cargados eran de signos diferentes, en ese momento el valor de *ok* se cambia a *False* y se avisa con eso que se ha descubierto una anomalía. Cuando el ciclo termina (al cargarse el 0), el valor final de *ok* nos dice si los números cargados eran del mismo signo (*ok = True*) o no (*ok = False*). El esquema (casi en pseudocódigo…) podría quedar modificado así:

```
# si ok es True, todos son del mismo signo hasta aquí...
ok = True

# acumulador para los números cargados...
suma = 0

# carga del primer número...
actual = int(input('Cargue un número entero (con 0 corta): '))
```

```
while actual != 0:

    # la pregunta que sigue está en pseudocódigo por ahora...
    si hubo signos diferentes:
        ok = False

    # acumular los números, sin importar si son del mismo signo...
    suma += actual

    # cargar el siguiente...
    actual = int(input('Cargue otro (con 0 corta): '))

# si al cortar el ciclo, ok es True
# entonces la secuencia estaba bien...
if ok:
    print('Todos los números eran del mismo signo')
    print('La suma de esos números es:', suma)
else:
    print('Los números no eran todos del mismo signo')
```

El acumulador *suma* sólo se muestra si al finalizar el ciclo se comprueba que todos los números eran del mismo signo. De otro modo, su valor simplemente se ignora.

Note que una vez que *ok* cambia a *False*, no vuelve ya a cambiar a *True*: si cambió a *False* al menos una vez, significa que al menos una vez se detectaron números con signos cambiados y esa anomalía es definitiva: no importa como siga la carga de la sucesión, el valor de *ok* seguirá en *False*.

Para determinar si en algún momento hay un cambio de signo, la idea sería poder contar en cada vuelta con dos números en lugar de sólo uno. Por caso, sabemos que la variable *actual* contiene el número que se acaba de cargar en esa iteración, pero podríamos usar una segunda variable llamada ***anterior*** para guardar en ella el valor del número que se cargó en la vuelta anterior. Eso es fácil de hacer: al cargar el primer número, se asigna ese mismo en ***anterior*** (en la primera vuelta y sólo en ella las variables actual y ***anterior*** contendrán el mismo número) y a partir de allí se almacena en ***anterior*** el valor de *actual* inmediatamente antes de cargar el siguiente valor en *actual*:

```
# si ok es True, todos son del mismo signo hasta aquí...
ok = True

# acumulador para los números cargados...
suma = 0

# carga del primer número...
actual = int(input('Cargue un número entero (con 0 corta): '))

# inicializar anterior con el mismo primer número…
anterior = actual

while actual != 0:

    # la pregunta que sigue está en pseudocódigo por ahora...
    si hubo signos diferentes:
        ok = False

    # acumular los números, sin importar si son del mismo signo...
    suma += actual

    # guardar el actual para ser el anterior en la vuelta siguiente...
```

```
        anterior = actual

        # cargar el siguiente...
        actual = int(input('Cargue otro (con 0 corta): '))

    # si al cortar el ciclo, ok es True
    # entonces la secuencia estaba bien...
    if ok:
        print('Todos los números eran del mismo signo')
        print('La suma de esos números es:', suma)
    else:
        print('Los números no eran todos del mismo signo')
```

Y como ahora tenemos el valor *actual* y el de la vuelta *anterior*, sólo debemos controlar si ambos son del mismo signo o no. Si ***no*** fuesen del mismo signo, tendríamos detectada una anomalía y *ok* cambiaría a *False*. La forma más sencilla se determinar si dos números son del mismo signo o no (sabiendo que ninguno de ellos es cero) es multiplicarlos entre sí: si el resultado es positivo, los números eran del mismo signo (los dos positivos o los dos negativos); pero si el resultado es negativo, entonces ambos eran de signos opuestos. El esquema puede entonces replantearse así, en forma de programa definitivo:

```
__author__ = 'Catedra de AED'


print('Procesamiento de números del mismo signo...')

# si ok es True, todos son del mismo signo hasta aquí...
ok = True

# acumulador para los números cargados...
suma = 0

# carga del primer número...
actual = int(input('Cargue un número entero (con 0 corta): '))

# variable para tener el anterior en cada vuelta...
anterior = actual
while actual != 0:

    # controlar los signos del numero actual y el anterior
    if actual * anterior < 0:
        # si el producto es < 0, eran de distinto signo...
        # cambiar el valor de ok...
        ok = False

    # acumular los números, sin importar si son del mismo signo...
    suma += actual

    # poner el actual como el anterior de la vuelta siguiente...
    anterior = actual

    # cargar el siguiente...
    actual = int(input('Cargue otro (con 0 corta): '))

# si al cortar el ciclo, ok es True
# entonces la secuencia estaba bien...
if ok:
    print('Todos los números eran del mismo signo')
    print('La suma de esos números es:', suma)
```

```
else:
    print('Los números no eran todos del mismo signo')
```

Anexo: Temas Avanzados

En general, cada Ficha de Estudios podrá incluir a modo de anexo un capítulo de *Temas Avanzados*, en el cual se tratarán temas nombrados en las secciones de la Ficha, pero que requerirían mucho tiempo para ser desarrollados en el tiempo regular de clase. El capítulo de *Temas Avanzados* podría incluir profundizaciones de elementos referidos a la programación en particular o a las ciencias de la computación en general, como así también explicaciones y demostraciones de fundamentos matemáticos. En general, se espera que el alumno sea capaz de leer, estudiar, dominar y aplicar estos temas aún cuando los mismos no sean específicamente tratados en clase.

a.) Tratamiento de números complejos en Python.

En muchos problemas se requerirá trabajar con cálculos de raíces de distintos números, y sabemos que una raíz de orden par (raíz cuadrada, raíz cuarta, etc.) de un número negativo no está definida en el campo de los números reales (ya que no existe un *número real* tal que elevado al cuadrado dé como resultado un número negativo).

En álgebra y análisis matemático el campo de los *números complejos* (que surgió como una extensión del campo de los *números reales*), permite trabajar con raíces de orden par de números negativos introduciendo un elemento llamado la *unidad imaginaria*, cuyo valor se asume como igual a la *raíz cuadrada de (-1)* y se denota como *i* en matemática o también como *j* en el área de la ingeniería. Obviamente, el valor efectivo de la raíz cuadrada de (-1) no se calcula, sino que se deja expresado como $j = \sqrt{-1}$ y luego se arrastra ese valor como parte de las operaciones algebraicas a realizar y así se puede continuar adelante con ciertos cálculos algebraicos y reducir algunas expresiones *llevándolas al campo complejo*. Por ejemplo, si se pide calcular la raíz cuadrada de (-4) (que no tiene solución en el campo real) se puede proceder así en el campo complejo:

$$\begin{aligned} \sqrt{(-4)} &= \sqrt{(4 * (-1))} \\ &= \sqrt{4} * \sqrt{(-1)} \\ &= 2 * j \end{aligned}$$

En general, un número complejo *c* es de la forma *(a + bj)* donde *a* es un número en coma flotante (designado como la *parte real* del complejo) y *b* es otro número en coma flotante (designado como la *parte imaginaria* del complejo). El factor *j* representa la unidad imaginaria $\sqrt{-1}$ ya citada. Tanto la parte real como la imaginaria pueden valer 0, con lo cual se tendrían números complejos tales como *(0 + 3j)* o como *(4 + 0j)* y está claro que en estos casos, la parte igual a cero puede obviarse. En nuestro caso, el valor de $\sqrt{(-4)}$ es el complejo *(0 + 2j)* que fue expresado como *2j*.

El hecho es que en muchos lenguajes de programación se producirá un error en tiempo de ejecución (el programa se interrumpirá) si se intenta calcular la raíz cuadrada (o cualquier raíz de orden par) de un número negativo. En otros lenguajes, el programa no se interrumpirá pero la variable donde se almacene el resultado quedará asignada con una constante que representa un valor indefinido (el cual no podrá usarse para cálculos posteriores).

Pero en Python *el resultado será efectivamente un número complejo*: Python provee un tipo especial de dato numérico llamado *complex* (que en realidad es lo que se conoce como una

clase) que permite representar números complejos de la forma *a + bj* tal como se indicó más arriba [3]. El factor *j* representa la unidad imaginaria y es automáticamente gestionado por Python. Por lo tanto, cualquier expresión que pudiese dar como resultado un número complejo, en Python efectivamente funcionará y entregará el complejo como resultado, sin que el programador deba preocuparse por detalles de validación. En la siguiente secuencia de instrucciones, se está intentando calcular la raíz cuadrada del valor -4 para asignar el resultado en *x* (recuerde que calcular la raíz cuadrada es lo mismo que elevar a la potencia 0.5):

```
x = (-4)**0.5
print('x:', x)
```

El resultado de ejecutar este script Python, es la siguiente salida en consola estándar (remarcamos la parte real en azul y la parte imaginaria en rojo, para mayor claridad):

```
x: (1.2246467991473532e-16+2j)
```

Como se ve, Python realiza el cálculo obteniendo un valor de tipo *complex*, y lo mismo ocurrirá cada vez que se quiera calcular cualquier raíz de valor par (raíz cuarta, raíz sexta, etc.) de un número negativo. Note que el número en coma flotante que representa la *parte real* del complejo, está expresado en este caso en *notación científica*: el valor flotante

```
1.2246467991473532e-16
```

equivale a:

$$1.2246467991473532 * 10^{-16}$$

lo que finalmente equivale al número:

```
0.0000000000000001224646799147353 2
```

que es prácticamente igual a cero.

Sabemos que $\sqrt{-4} = (0 + 2j)$ y no *(1.2246467991473532e-16+2j)* (que fue el resultado entregado por nuestro script). En la práctica, y como vimos, la *parte real* del complejo calculado por Python es igual a cero hasta una precisión de 15 dígitos de mantisa (o de decimales a la derecha del punto) y el resto de esos dígitos pueden ser despreciados por redondeo: a los efectos prácticos, el complejo entregado por Python equivale a *(0 + 2j)* y esto no contradice lo que hemos calculado en forma manual[3].

Por otra parte, si un programador necesita asignar directamente un número complejo en una variable, en Python tiene un par de mecanismos para hacerlo recurriendo a la ya citada clase *complex*. Una forma se basa en el siguiente ejemplo [3]:

```
c1 = complex(3.4, 4)
print('c1:', c1)
```

La función *complex()* en este caso toma dos parámetros: el primero (3.4 en el ejemplo) será la *parte real* del complejo a crear, y el segundo (4 en este caso) será la *parte imaginaria*. el factor *j* será automáticamente incluido por Python. La salida de este script será la siguiente:

```
c1: (3.4+4j)
```

[3] La forma en que se representan internamente los números en coma flotante en un computador lleva a mínimos errores de precisión que se hacen más evidentes mientras más dígitos a la derecha de la coma se miren, pero en muchas situaciones prácticas estos errores de precisión pueden despreciarse (como en el caso del ejemplo analizado).

Por otra parte, la misma función *complex()* puede ser usada de forma que se le envíe una cadena de caracteres representando al complejo, y no ya los dos números:

```
c2 = complex('2-5j')
print('c2:', c2)
```

El script anterior producirá la siguiente salida en consola estándar:

```
c2: (2-5j)
```

Sólo debemos hacer notar que si se crea un complejo usando una cadena de caracteres como parámetro de la función *complex()*, no deben introducirse espacios en blanco a los lados del signo + o del signo – que separa ambas partes del complejo. El mismo ejemplo pero expresado con espacios:

```
c2 = complex('2 - 5j')
print('c2:', c2)
```

producirá un error en tiempo de ejecución como éste:

```
Traceback (most recent call last):
  File "C:/ Ejemplo/prueba.py", line 10, in <module>
    c2 = complex('2 + 5j')
ValueError: complex() arg is a malformed string
```

Todo lo que hemos introducido hasta aquí respecto del tratamiento de números complejos en Python, puede ser aplicado ahora en el análisis y solución de un problema clásico del Álgebra:

Problema 17.) *Un polinomio de segundo grado tiene la forma* $p(x) = ax^2 + bx + c$ *donde* **a**, **b**, *y* **c** *son valores constantes conocidos como los coeficientes del polinomio y* **x** *es la variable independiente. Se supone que el coeficiente* **a** *es diferente de cero (pues de lo contrario el término* ax^2 *desaparece y el polinomio se convierte en un de primer grado).*

Si el polinomio se iguala a cero, se obtiene una ecuación de segundo grado: $ax^2 + bx + c = 0$ *y resolver la ecuación es el proceso de hallar los valores de* **x** *que hacen que efectivamente el polinomio valga cero.*

Por el Análisis Matemático se sabe que todo polinomio de grado **n** *tiene exactamente* **n** *raíces (reales y/o complejas) para su ecuación y por lo tanto, una ecuación de segundo grado tiene dos raíces a las que tradicionalmente se designa como* **x1** *y* **x2**. *El problema de encontrar estos dos valores fue estudiado desde varios siglos antes de Cristo al menos por los babilonios, los indios y los árabes (de hecho, nuestro ya conocido Al-Jwarizmi contribuyó de forma significativa) y la fórmula de cálculo es bien conocida:*

$$x_{1-2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

El valor de **x1** *se obtiene usando el signo + (más) en el numerador, y el valor de* **x2** *se obtiene usando el signo – (menos). Desarrolle un programa que dados los valores de los coeficientes* **a**, **b** *y* **c** *(y suponiendo que* **a** *será diferente de cero) calcule y muestre las dos raíces* **x1** *y* **x2** *de la ecuación, incluso si las mismas son complejas. El programa debe permitir resolver más de una ecuación: al finalizar con una de ellas, el mismo programa debe preguntar al usuario si desea continuar con otra, y en ese caso cargar otro juego de coeficientes y volver a resolver.*

Discusión y solución: Está claro que los datos a ingresar son los valores de los coeficientes *a*, *b* y *c* y que estos valores serán *números reales* (de coma flotante). Los resultados a obtener

también son claros: los valores de las raíces *x1* y *x2* que podrán ser *números reales* o eventualmente *números complejos*.

El hecho que puede llevar a que las raíces sean números complejos surge de la presencia del cálculo de la raíz cuadrada que se ve en el numerador de la fórmula. La expresión $b^2 - 4ac$ (que se conoce como el *discriminante* de la ecuación) podría dar como resultado un valor negativo, y en ese caso la raíz cuadrada no puede calcularse en el campo real. En consecuencia, si el discriminante fuese negativo las raíces de la ecuación estarían en el campo complejo e incluso en ese caso nuestro programa debe mostrar sus valores.

Si programador está trabajando con Python, entonces la solución a este problema es directa y lineal, ya que Python calculará esa raíz cuadrada y obtendrá resultados reales o complejos según sea el caso. Hay poco que discutir aquí, y mostramos la solución como un script directo:

```
__author__ = 'Cátedra de AED'

seguir = 's'
while seguir == 's' or seguir == 'S':

    # titulo y carga de datos...
    print('Raíces de la ecuación de segundo grado...')
    a = float(input('a: '))
    b = float(input('b: '))
    c = float(input('c: '))

    # procesos: aplicar directamente las fórmulas...
    x1 = (-b + (b**2 - 4*a*c)**0.5) / (2*a)
    x2 = (-b - (b**2 - 4*a*c)**0.5) / (2*a)

    # visualización de resultados...
    print('x1:', x1)
    print('x2:', x2)

    seguir = input('Desea resolver otra ecuacion? (s/n): ')

print('Gracias por utilizar nuestro programa...')
```

El programa incluye un ciclo *while* que permite resolver una ecuación y ofrecer al usuario la posibilidad de resolver otra. La variable *seguir* comienza con el valor 's' (abreviatura de 'sí') para forzar al ciclo a entrar en la primera vuelta y resolver la primera ecuación. Pero luego de resolverla, al final del bloque de acciones del ciclo se muestra un mensaje al usuario preguntándole si desea resolver otra o no. La respuesta del usuario se carga por teclado en la variable *seguir*, asumiendo que cargará una 's' si quería responder que 'sí', o una 'n' para responder que 'no'. Al retornar a la cabecera del ciclo, se chequea el valor de la variable *seguir*: si la misma llegó valiendo una 's' o una 'S' (habilitando que tanto la minúscula como la mayúscula sean válidas) el ciclo ejecutará otra vuelta, cargando nuevamente valores en la variables a, b y c y resolviendo la nueva ecuación. El ciclo se detendrá cuando el usuario *no cargue* una 's' ni una 'S': en los hechos, terminará con *cualquier letra* diferente de esas dos.

En cuanto al resto del programa, es interesante remarcar que las variables *x1* y *x2* serán asignadas con los valores correctos, sean estos de tipo *float* o de tipo *complex*, y esto es así debido al hecho (que ya hemos estudiado) de ser Python un lenguaje de *tipado dinámico* [2].

Ahora bien: ¿qué pasaría en el caso que el lenguaje de programación usado no tuviese disponible la clase *complex*, lista para usar e integrada en la forma de trabajo de sus operadores aritméticos? Incluso si el programador sabe que puede usar Python, podría presentarse este caso como un desafío práctico o como un objetivo didáctico. En resumen: ¿cómo plantear (incluso en Python) el cálculo de las raíces de la ecuación de segundo grado, sin usar la clase *complex* de Python?

Ahora el trabajo es un poco más desafiante, ya que el programador deberá evitar el cálculo de la raíz cuadrada si el discriminante es negativo, y en ese caso hacer los cálculos de los valores de la parte real y la parte imaginaria para finalmente mostrar todo en la consola de salida. Por lo pronto, entonces, sería prudente calcular el valor del discriminante y dejarlo alojado en una variable:

```
d = b**2 - 4*a*c
```

La instrucción anterior calcula el valor del discriminante y lo deja almacenado en una variable *d*. Recuerde que el discriminante no incluye el cálculo de la raíz, sino sólo el valor de la expresión $b^2 - 4ac$.

Por otra parte, si el valor del discriminante *d* es mayor o igual a cero, entonces el cálculo de las raíces no presenta problema alguno y puede hacerse exactamente en la misma forma que se aplicó en nuestro primer script:

```
if d >= 0:
    # raices reales...
    x1 = (-b + d**0.5) / (2*a)
    x2 = (-b - d**0.5) / (2*a)
```

La secuencia anterior hace los cálculos de ambas raíces en forma directa, aplicando las fórmulas, pero usando el valor del discriminante que fue almacenado antes en la variable *d*. Con esto se logra algo de ahorro de código fuente y también un poco de ahorro de tiempo al ejecutar el programa, ya que no tendrá que volver a calcular dos veces más el discriminante.

Para el cálculo de las raíces complejas tendremos algo de trabajo extra. La raíces serán complejas sólo si el discriminante *d* es negativo, y en este caso ambas raíces tendrán la misma *parte real* y la misma *parte imaginaria*, pero aplicando la suma en *x1* y la resta en *x2* (se dice esos dos números complejos son *complejos conjugados*). Para deducir cuánto vale entonces la parte real *pr* de ambos complejos, y cuánto vale la parte imaginaria *pi* de ambos, nos basamos en el siguiente desarrollo, que se aplica en general y sin importar por ahora si el discriminante es negativo o no (ver *Figura 7*):

Figura 7: Desarrollo del cálculo de las raíces de la ecuación de segundo grado

$$x_{1-2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x_{1-2} = -\frac{b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

$$x_{1-2} = -\frac{b}{2a} \pm \frac{\sqrt{\mathrm{d}}}{2a}$$

Si en la última expresión del desarrollo anterior el valor de *d* es negativo, entonces el factor √*d* no puede calcularse en el campo real. Ambas raíces serán complejas y el cálculo debe seguir en el campo complejo. Para eso, el discriminante *d* debe cambiar de signo, y hacer aparecer la unidad imaginaria *j*. El cambio de signo del valor contenido en una variable *d*, se puede hacer simplemente así:

```
d = -d
```

Observe: si el valor inicial era d = 4, entonces –d es –4. Pero si el valor era d = –4, entonces el valor –d es: –d = – (–4) o sea –d = 4. El valor cambiado de signo (–d) se asigna en la propia variable d y eso provoca el cambio del signo del discriminante a los efectos del cálculo. Y entonces, según la última parte del desarrollo que hicimos en la *Figura 7*, nos queda que la parte real *pr* de ambos complejos *x1*, *x2* es:

$$pr = -\frac{b}{2a}$$

Y la parte imaginaria *pi* de ambos complejos *x1*, *x2*, es:

$$pi = \frac{\sqrt{-d}}{2a} * j$$

De este modo, el cálculo de las raíces complejas puede verse como sigue, en la rama falsa de la condición:

```
if d >= 0:
     # raices reales...
     x1 = (-b + d**0.5) / (2*a)
     x2 = (-b - d**0.5) / (2*a)

else:
     # raices complejas...
     # calcular la parte real...
     pr = -b / (2*a)

     # calcular la parte imaginaria...
     # ...cambiando el signo del discriminante...
     pi = (-d)**0.5 / (2*a)

     # armar cadenas uniendo las partes a mostrar...
     x1 = '(' + str(pr) + '+' + str(pi) + 'j)'
     x2 = '(' + str(pr) + '-' + str(pi) + 'j)'
```

El resultado final en este caso está formado por cadenas de caracteres: en *x1* se asigna una cadena formada por la conversión a string de los números *pr* y *pi*, unida con un par de paréntesis, el signo más ('+') y la letra 'j' que representa a la unidad imaginaria. Algo similar se hace con *x2*, pero cambiando el signo más ('+') por el menos ('-').

Note que para armar cada cadena final, las distintas partes se unen o concatenan usando el *operador suma* (que justamente actúa como un concatenador cuando suma cadenas de caracteres) [3]:

```
x1 = '(' + str(pr) + '+' + str(pi) + 'j)'
x2 = '(' + str(pr) + '-' + str(pi) + 'j)'
```

Sin embargo, recuerde que *pr* y *pi* son variables que al momento de ejecutar estas dos instrucciones contienen números (y no cadenas). Si el intérprete encuentra que el operador suma debe "sumar" una cadena (como *')'*) con un número (como *pr*), el programa se interrumpirá con un mensaje de error: no pueden unirse o sumarse una cadena y un número. Para evitar este problema, se usa la función *str()* de la librería estándar de Python (ver Ficha 3, sección 3) que permite convertir un objeto cualquiera en una cadena de caracteres. Por ejemplo, si *pr* vale 2.45, entonces *str(pr)* retornará la cadena '2.45' que es el número original, pero en formato de string.

El programa completo para el cálculo de las raíces puede verse entonces así:

```
__author__ = 'Cátedra de AED'


seguir = 's'
while seguir == 's' or seguir == 'S':

    # título general y carga de datos...
    print('Raíces de la ecuación de segundo grado...')
    a = float(input('a: '))
    b = float(input('b: '))
    c = float(input('c: '))

    # procesos: calcular y controlar el discriminante...
    d = b**2 - 4*a*c

    # ... y calcular las raíces según corresponda...
    if d >= 0:
        # raices reales...
        x1 = (-b + d**0.5) / (2*a)
        x2 = (-b - d**0.5) / (2*a)

    else:
        # raices complejas...
        # calcular la parte real...
        pr = -b / (2*a)

        # calcular la parte imaginaria...
        # ...cambiando el signo del discriminante...
        pi = (-d)**0.5 / (2*a)

        # armar cadenas uniendo las partes a mostrar...
        x1 = '(' + str(pr) + '+' + str(pi) + 'j)'
        x2 = '(' + str(pr) + '-' + str(pi) + 'j)'

    # visualización de resultaddos...
    print('x1:', x1)
    print('x2:', x2)

    seguir = input('Desea resolver otra ecuacion? (s/n): ')

print('Gracias por utilizar nuestro programa...')
```

Este script une todas las piezas de la división en subproblemas que hemos propuesto (raíces reales calculadas como un proceso separado de las raíces complejas). Luego de cargar los valores de las variables *a*, *b*, *c* calcula el discriminante y deja ese valor en la variable *d*. Se usa una condición para verificar el signo de *d*, y se calculan las raíces reales o las raíces complejas

según sea el caso. Al final, se muestran los valores finales de *x1* y *x2*, que serán dos números reales si las raíces eran reales, o dos cadenas de caracteres representando a los dos complejos si las raíces eran complejas. Igual que antes, el programa incluye un ciclo *while* que permite al usuario del programa cargar una nueva ecuación si así lo desea.

Créditos

El contenido general de esta Ficha de Estudio fue desarrollado por el *Ing. Valerio Frittelli* para ser utilizada como material de consulta general en el cursado de la asignatura *Algoritmos y Estructuras de Datos* – Carrera de Ingeniería en Sistemas de Información – UTN Córdoba, en el ciclo lectivo 2019.

Actuaron como revisores (indicando posibles errores, sugerencias de agregados de contenidos y ejercicios, sugerencias de cambios de enfoque en alguna explicación, etc.) en general todos los profesores de la citada asignatura como miembros de la Cátedra, y particularmente los ingenieros *Silvio Serra*, *Analía Guzmán*, *Cynthia Corso*, *Karina Ligorria, Marcela Tartabini* y *Romina Teicher*, que realizaron aportes de contenidos, propuestas de ejercicios y sus soluciones, sugerencias de estilo de programación, y planteo de enunciados de problemas y actividades prácticas, entre otros elementos.

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] M. Pilgrim, "Dive Into Python - Python from novice to pro," 2004. [Online]. Available: http://www.diveintopython.net/toc/index.html.

[3] Python Software Foundation, "Python Documentation," 2018. [Online]. Available: https://docs.python.org/3/.