Ficha 8

Estructuras Repetitivas: Variantes

1.] Ciclos en Python: Variantes y elementos de control adicionales.

El bloque de acciones de un ciclo (*while* o *for*) en Python puede incluir una instrucción *break* para cortar el ciclo de inmediato sin retornar a la cabecera para evaluar la expresión lógica de control, en forma similar a otros lenguajes [1]. El siguiente script tiene el objetivo de cargar por teclado cinco números positivos y calcular la suma o acumulación de todos ellos. Pero el script contiene un *ciclo while* que se interrumpirá con una instrucción *break* si se carga por teclado un número cero o negativo, *aun cuando no se haya llegado a las 5 repeticiones que se esperaría en el ciclo*:

```
suma, i = 0, 1
while i <= 5:
    n = int(input('Ingrese un número mayor a cero: '))
    if n <= 0:
          break
    suma += n
    i += 1

print('La suma de los números ingresados es:', suma)
```

En el ejemplo anterior, el bloque de acciones del *ciclo while* carga por teclado un número *n*, y chequea con un *if* si ese número es cero o negativo. En caso de serlo, se ejecuta la instrucción *break* y su efecto es *cancelativo con respecto al ciclo*: el ciclo se interrumpe, y el programa continúa con la ejecución del *print()* que se encuentra a la salida del ciclo. Insistimos: se ejecuta el *break* y el ciclo se interrumpe, sin volver a la cabecera para chequear la expresión lógica de control del ciclo, por lo que su valor es ignorado en ese caso. Si el conjunto de números a procesar fuese {2, 4, 5, -2, 7, 1} este script cargaría los tres primeros y los acumularía sin problemas con el ciclo, pero el ciclo se interrumpiría en el cuarto (el -2) pues siendo negativo activaría la instrucción *break*. La salida de este programa para esa secuencia de números de entrada, sería:

```
La suma de los números ingresados es:  11
```

que es la suma 2 + 4 + 5, sin incluir al resto de los positivos del conjunto.

El siguiente programa pretende procesar uno por uno los caracteres de una cadena. Contará aquellos que representen letras minúsculas, y por separado contará los que representen mayúsculas. Los caracteres que no sean letras serán simplemente ignorados. Pero si aparece un punto (".") el *ciclo for* que itera sobre la cadena se interrumpirá con un *break*, aun cuando no se haya alcanzado todavía el último caracter de la cadena:

```
cadena = input('Ingrese una cadena: ')
minusculas = mayusculas = 0

for car in cadena:
```

```
        if car == '.':
            break

        if 'a' <= car <= 'z':
            minusculas += 1

        elif 'A' <= car <= 'Z':
            mayusculas += 1

    print('Cantidad de minúsculas:', minusculas)
    print('Cantidad de mayúsculas:', mayusculas)
```

Si la cadena ingresada por teclado fuese *ABCDabc34#.ABad* este programa mostraría una salida de la forma:

```
Ingrese una cadena: ABCDabc34#.ABad
Cantidad de minúsculas: 3
Cantidad de mayúsculas: 4
```

lo cual está de acuerdo con el objetivo: antes del *punto* hay cuatro letras mayúsculas y tres minúsculas. También hay dos números y un caracter numeral, pero tanto los números como el numeral son ignorados. Luego aparece el *punto* que da por terminado el *ciclo for* al ejecutar la instrucción ***break***. Y todo caracter que viniese después del *punto*, será ignorado (sea letra o no) ya que el ciclo de iteración terminó.

De forma similar, pero a la inversa, un ciclo cualquiera puede incluir una instrucción ***continue*** para *forzar una repetición del ciclo* sin terminar de ejecutar las instrucciones que queden por debajo de la invocación a ***continue*** [1]. El siguiente ejemplo es una variante del que mostramos antes para cargar números positivos con un *while*, pero haciendo ahora que el ciclo *fuerce la siguiente vuelta* si el valor cargado fue cero o negativo (pero observe que entonces en este caso, el script ***siempre*** pedirá los cinco números positivos, incluso si en el medio se cargó alguno negativo o cero):

```
suma = 0
i = 1

while i <= 5:
   n = int(input('Ingrese un número mayor a cero: '))
   if n <= 0 :
         continue
   suma += n
   i += 1

print('La suma de los números ingresados es:', suma)
```

Al ejecutarse la instrucción ***continue***, el ciclo no cortará su ejecución: volverá a la cabecera para volver a chequear la expresión lógica de control, pero no ejecutará en ese caso las instrucciones *suma += n* ni *i += 1*. Como de esta forma el contador *i* sólo se incrementa si entró un positivo, entonces ese ciclo se detendrá sólo si alguna vez se cargan cinco positivos.

A diferencia de otros lenguajes, en Python un ciclo (*for* o *while*) puede llevar opcionalmente una cláusula *else* en forma similar a una instrucción condicional, aunque el efecto de este *else* en un ciclo es bastante diferente a lo que ocurre con una condición común:

- ✓ En un ciclo *while*, las instrucciones de la rama *else* se ejecutan en el momento en que la expresión de control del ciclo se evalúa en *False* (sin importar en qué vuelta se obtuvo el *False*).
- ✓ En un ciclo *for*, las instrucciones de la rama *else* se ejecutan cuando el *for* termina de iterar sobre *todos* los elementos de la colección o secuencia dada.
- ✓ En ambos casos (tanto en un *while* como en un *for*) la rama *else* no será ejecutada si el ciclo terminó por acción de una instrucción *break*.

El siguiente programa es una variante del que mostramos antes para contar letras. Muestra el mismo ciclo *for* anterior, pero ahora con un *else*. Los resultados del conteo de letras se mostrarán sólo si el ciclo logró procesar toda la cadena (lo cual ocurrirá si la misma no tenía un punto): esas visualizaciones ahora están en el bloque *else* del *for*, por lo que sólo se ejecutarán si el ciclo cortó sin recurrir al *break*:

```
cadena = input('Ingrese una cadena: ')
minusculas = mayusculas = 0
for car in cadena:
    if car == '.':
        break

    if 'a' <= car <= 'z':
        minusculas += 1

    elif 'A' <= car <= 'Z':
        mayusculas += 1

else:
    print('Resultados (la cadena no contenía un punto):')
    print('Cantidad de minúsculas:', minusculas)
    print('Cantidad de mayúsculas:', mayusculas)

print('Proceso terminado')
```

Por último, veamos que en ocasiones el programador necesita *dejar vacío* el bloque de acciones de un ciclo o una rama de una condición y para casos así Python prevé el uso de la instrucción *pass*. Esa instrucción sirve para indicar al intérprete que simplemente considere vacío el bloque que la contiene:

```
# un ciclo que hace 10000 repeticiones sin bloque de acciones...
for i in range(10000):
    pass
```

En el ejemplo anterior, el ciclo *for* efectivamente hace 10000 repeticiones, pero no ejecuta ninguna acción adicional en cada una de ellas. El programa insumirá cierto tiempo en terminar de ejecutar este ciclo, y luego continuará normalmente con las instrucciones que sigan. Usar un ciclo con un elevado número de repeticiones pero con bloque de acciones vacío suele ser un truco empleado para provocar un breve retardo o *delay* en la ejecución del programa si el programador lo cree necesario.

La instrucción *pass* también puede usarse para dejar vacía una rama de una condición, como se ve en los siguiente ejemplos:

```
# una condición con rama verdadera en blanco…
if n1 > n2:
    pass
```

```
else:
    print('El primero no es el mayor')
```

2.] Programas controlados por menú de opciones.

Analizaremos ahora la forma de plantear lo que se conoce como un *programa controlado por menú de opciones*, que es en realidad la formalización de una técnica o esquema de *interfaz elemental de usuario*. Abordamos el tema a partir del siguiente enunciado:

Problema 21.) *Desarrollar un **Programa Controlado por Menú de Opciones**, que incluya opciones para realizar las siguientes tareas:*

1. *Cargar un valor entero n por teclado, y obtener la suma de los enteros del 1 al n.*
2. *Cargar un valor entero n por teclado, y obtener su factorial.*
3. *Cargar por teclado los coeficientes a, b, y c de un polinomio de segundo grado, y obtener el valor del polinomio en el punto x, siendo x un valor que también se carga por teclado.*

Discusión y solución: Un *programa controlado por menú de opciones* es aquel que, al comenzar, presenta en pantalla una lista de opciones (que los programadores designan justamente como un *menú de opciones*) para que el usuario del programa elija la acción que desea llevar a cabo. La mecánica básica es que al seleccionar una opción, el programa desarrolla la misma y luego *vuelve* a presentar el menú en pantalla, de modo que el usuario puede elegir otra opción (o la misma anterior si lo desea). El programa incluye una opción para finalizar la ejecución y *sólo finaliza si alguien selecciona esa opción de terminación* [2].

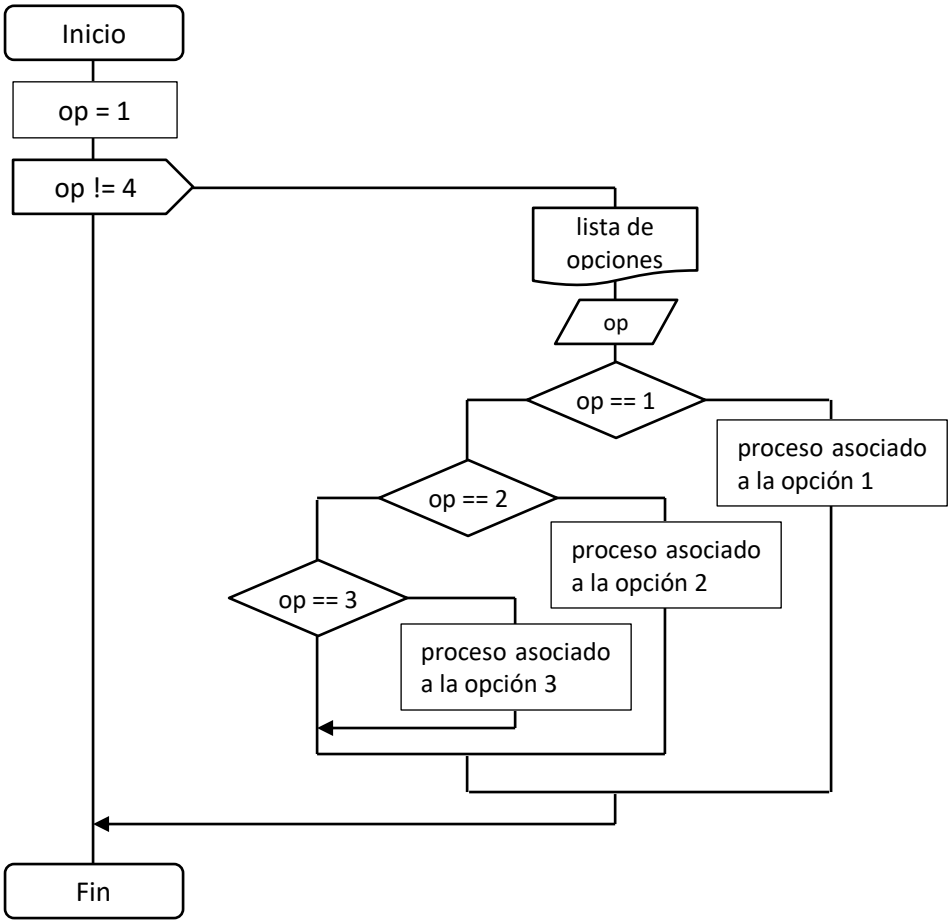
Si bien no es obligatorio, el hecho es que estos programas suelen usar un ciclo *[1, N]* para controlar todo el funcionamiento, debido a que la pantalla del menú principal debe mostrarse *al menos una vez* cuando el programa empieza, y luego *repetir* la presentación al terminar de desarrollar cada opción elegida por el usuario. Además, dentro del ciclo se utiliza un esquema de instrucciones condicionales anidadas, para chequear cuál fue la opción ingresada por el usuario y poder responder a ella con el proceso que corresponda.

Por lo tanto, nuestro programa usará un *ciclo while ajustado a operar en forma [1, N]* (esto es, planteado de forma tal que nos aseguremos que al chequear la condición de control la primera vez, esta sea verdadera) para controlar la repetición y un *esquema de condiciones if – elif* para determinar cuál es la opción elegida por el usuario.

El programa para el enunciado propuesto presentará una pantalla de *cuatro opciones*: las tres primeras (numeradas del 1 al 3) son las pedidas por el propio enunciado del ejercicio (y son llamadas en general *opciones operativas*) y la cuarta (numerada con el número 4) es la *opción de salida*. De este modo, el ciclo *while* controla que la opción ingresada sea distinta de 4, y en caso afirmativo vuelve a mostrar el menú. Sólo corta el proceso (y en este caso también el programa) cuando se selecciona la opción 4. Ante cualquier selección incorrecta, el programa vuelve a mostrar el menú, ignorando simplemente la entrada errónea. En la *Figura 1* (*página 153*) se muestra el diagrama de flujo general sugerido para el programa (sin detallar los procesos a realizar en cada rama).

Los procesos asociados a cada una de las opciones (salida verdadera de cada una de las condiciones del diagrama) son directos, y los analizaremos desde su planteo en código fuente, uno por uno.

Figura 1: Diagrama de flujo general sugerido para el programa controlado por menú de opciones.



En este momento, conviene aplicar un ya conocido detalle de control: Tanto para el cálculo de la suma (en la opción 1) como en el factorial (opción 2), se esperaría que el valor cargado por teclado para hacer el cálculo sea mayor o igual a 0. En ninguno de ambos casos sería admisible un valor negativo, y para controlar que eso no ocurra, podemos incluir un proceso de *validación* que realice la carga por teclado, pero verificando que el número cargado no sea negativo, pidiéndolo nuevamente en caso de serlo:

```
n = -1
while n < 0:
    n = int(input('Ingrese n (no negativo, por favor): '))
    if n < 0:
        print('Error... se pidio no negativo... cargue de nuevo...')
```

Ahora sí, comencemos por el proceso asociado a la opción 1: de acuerdo al enunciado, si se elige la opción 1 se debe cargar por teclado un número entero *n*, y calcular la suma de los números enteros desde el 1 hasta el *n*. Por ejemplo, para *n* = 5 la suma pedida es 1 + 2 + 3 + 4 + 5 = 15. En principio, podría calcularse usando un ciclo *for* ajustado para recorrer el intervalo *[1, n]* con una variable *i*, y acumular los valores de *i*. Un proceso que aplique la idea podría verse así:

```
ac = 0
for i in range(1, n+1):
    ac += i
```

Sin embargo, el proceso indicado demoraría lo que demore el ciclo *for* en terminar de recorrer el *range* de control, y si bien eso puede parecer despreciable, el hecho es que para un valor realmente grande de *n* la demora en el tiempo de ejecución comenzará a notarse... incluso una computadora se las verá en problemas cuando deba ejecutar un proceso con una enorme cantidad de pasos.

Es conveniente, para una buena formación a futuro, que el programador se acostumbre a pensar todo el tiempo en soluciones mejores (en este caso, más rápidas)… incluso cuando podría parecer que en el contexto del problema no vale la pena el esfuerzo. En este caso, existe al menos una manera mucho más rápida de hacer el cálculo, recurriendo a una fórmula directa para calcular esa suma. Puede demostrarse (por inducción matemática) [3] que:

$$1 + 2 + 3 + 4 + \ldots + (n-2) + (n-1) + n = \mathbf{n * (n+1) / 2}$$

Sólo a modo de ejemplo, si *n* = 5 entonces 1 + 2 + 3 + 4 + 5 = 5 * 6 / 2 = 30 / 2 = 15… y con este resultado a mano, ya no necesitamos perder el tiempo recorriendo un *range*, sino mostrar directamente el resultado del cálculo:

```
s = n * (n+1) // 2
print('Suma:', s)
```

La versión original del proceso que hacía la suma con un ciclo tiene un *tiempo de ejecución que crece en forma directamente proporcional al valor de n*. Pero la segunda versión se ejecuta en *tiempo constante*: no importa cuál sea el valor de *n*, el tiempo insumido en el cálculo es siempre el mismo.

El proceso a desarrollar para la opción 1 entonces podría quedar finalmente planteado así:

```
if opcion == 1:
       n = -1
       while n < 0:
          n = int(input('Ingrese n (no negativo, por favor): '))
          if n < 0:
              print('Error... se pidio no negativo... cargue de nuevo...')

       s = n * (n+1) // 2
       print('Suma de los enteros del 1 al', n, ':', s)
```

Respecto del proceso para la opción 2, el planteo es muy similar al de la opción 1, pero ahora se pide el *factorial* de *n* (y no la suma). El problema del *factorial* de *n* fue analizado y resuelto en una Ficha anterior, por lo que aquí simplemente reutilizaremos el proceso que propusimos en ese momento, con un único pequeño ajuste: el *range* a recorrer para obtener los factores del cálculo, está ajustado a *range(2, n+1)* en lugar de *range(1, n+1)*… ya que la multiplicación por 1 en realidad no aporta nada al producto final:

```
if opcion == 2:
       n = -1
       while n < 0:
          n = int(input('Ingrese n (no negativo, por favor): '))
          if n < 0:
              print('Error... se pidio no negativo... cargue de nuevo...')

       f = 1
       for i in range(2, n+1):
           f *= i

       print('Factorial de', n, ':', f)
```

Y en relación a la opción 3, el requerimiento es cargar por teclado los coeficientes *a*, *b*, y *c* de un polinomio *p* de segundo grado, más el valor de *x* en el cual se quiere evaluar ese polinomio, y simplemente retornar el valor *p(x)*. No se requiere en este caso validar las cargas por teclado, ya que ahora son admisibles números negativos, cero o positivos. Aquí también el cálculo es directo:

```
if opcion == 3:
    a = float(input('a: '))
    b = float(input('b: '))
    c = float(input('c: '))
    x = float(input('x: '))
    p = a*pow(x, 2) + b*x + c
    print('Valor de p(', x, '):', p)
```

Mostramos a continuación (por fin…) el programa completo que incluye el ciclo del menú de opciones:

```
op = 1
while op != 4:
     # visualizacion de las opciones...
     print('1. Suma de 1 al n')
     print('2. Factorial de n')
     print('3. Polinomio valuado en x')
     print('4. Salir')
     op = int(input('Ingrese el numero de la opcion elegida: '))

     # chequeo de la opcion elegida...
     if op == 1:
          # Calculo de la suma de 1 a n...
          n = -1
          while n < 0:
               n = int(input('Ingrese n (>=0, por favor): '))
               if n < 0:
                   print('Error... se pidio >=0... cargue de nuevo...')

          s = n * (n+1) // 2
          print('Suma de los enteros del 1 al', n, ':', s)

     elif op == 2:
          # Calculo del factorial de n...
          n = -1
          while n < 0:
               n = int(input('Ingrese n (>=0, por favor): '))
               if n < 0:
                   print('Error... se pidio >=0... cargue de nuevo...')
          f = 1
          for i in range(2, n+1):
             f *= i

          print('Factorial de', n, ':', f)

     elif op == 3:
          # Calculo del valor de un polinomio...
          a = float(input('a: '))
          b = float(input('b: '))
          c = float(input('c: '))
          x = float(input('x: '))
          p = a*pow(x, 2) + b*x + c
          print('Valor de p(', x, '):', p)
```

3.] Implementación de un juego sencillo: el *Número Secreto*.

Está claro que uno de los campos de aplicación más populares de las computadoras es el desarrollo de juegos. Ese segmento del mercado informático mundial mueve miles de millones de dólares al año[1]. Nuestra humilde contribución a ese gigantesco mercado comenzará con la implementación del juego del *Número Secreto*, en base al siguiente enunciado general:

Problema 22.) *El **Juego del Número Secreto** consiste en lo siguiente: la computadora tiene un número guardado (que el jugador obviamente no conoce) y el jugador debe tratar de adivinarlo. Si lo logra, gana el juego y la computadora le avisa en cuántos intentos lo hizo. Si no lo logra en cierta cantidad predefinida intentos, el juego termina y se avisa que el jugador ha perdido. La cantidad máxima de intentos que el jugador tendrá a su disposición es un número que debe cargarse por teclado antes de comenzar a jugar, al igual que el límite derecho del intervalo que contendrá al número secreto elegido por el computador (es decir, el usuario debe poder indicar el número estará entre 1 y 30 o bien entre 1 y 50 o bien en el intervalo que el propio usuario decida). Desarrolle un programa completo que implemente este juego.*

Discusión y solución: La primera cuestión a resolver es cómo hacer que la máquina *piense* un número sin que lo sepa el jugador, y de forma tal que cada vez que el programa se ejecute nuevamente el número pensado sea diferente... (de lo contrario, el segundo o tercer jugador tendría muy fácil el juego... sólo debería saber qué número le tocó al anterior jugador…) Y es obvio que lo único que tenemos que hacer es generar ese número en forma aleatoria, con alguna de las funciones que Python provee para ello en sus librerías estándar, como las funciones *random.random()*, *random.randrange()* o *random.randint()* (ver *Ficha 4*) [1] [4]. En el juego a implementar, querríamos que el número a adivinar sea entero y en el intervalo pedido por el usuario (por ejemplo, entre 1 y 50). Y como sabemos, la forma más simple de obtener un número entero aleatorio en el intervalo *[a, b]* en Python, consiste en invocar a la función *random.randint(a, b)*. Por ejemplo, la invocación

```
x = random.randint(1, 50)
```

en Python, hará que la variable *x* sea asignada con un número aleatoriamente elegido entre 1 y 50 (ambos incluidos).

En cuanto al programa que implementa el juego, la idea final es que al inicio se muestre un mensaje indicando entre qué valores está el número secreto. Por ejemplo, suponga que el número secreto guardado por la computadora es el 14. Entonces una sesión de juego podría ser la siguiente, suponiendo que la cantidad máxima de intentos sea 5 y el límite derecho del intervalo elegido por el usuario sea 50:

```
El número está entre 1 y 50.
[Intento: 1]  ==> Ingrese:  30
```

[1] Son numerosas y muy conocidas las películas cuyo tema central gira en torno a la visión fantástica que la industria del cine tiene acerca de los juegos de computadoras. Una de las primeras (ya algo vieja…) fue la película *WarGames* (en español conocida como *Juegos de Guerra*) de 1983, protagonizada por un muy joven *Matthew Broderick* y dirigida por *John Badham*. En esa película se narra la historia de un joven hacker que casi provoca una guerra mundial termonuclear, pero que finalmente termina evitándola… haciendo que la supercomputadora que estaba por lanzar el ataque juegue contra si misma una eterna serie de partidas empatadas de Tic-Tac-Toe (o TaTeTi) hasta que la máquina "aprendió" que en algunas situaciones no hay ganadores… y lo mejor entonces es no hacer nada.

```
El número está entre 1 y 30.
[Intento: 2]  ==> Ingrese:  10

El número está entre 10 y 30.
[Intento: 3]  ==> Ingrese:  15

El número está entre 10 y 15.
[Intento: 4]  ==> Ingrese:  14

Acertó!!! (en 4 intentos)
```

Un programa completo en Python que implementa este juego, se muestra a continuación:

```
import random

# Títulos y carga de datos básicos...
print('Juego del Número Secreto... Configuración Inicial...')
limite_derecho = int(input('El número secreto estará entre 1 y: '))
cantidad_intentos = int(input('Cantidad máxima de intentos: '))

# limites iniciales del intervalo de búsqueda...
izq, der = 1, limite_derecho

# contador de intentos...
intentos = 0

# bandera de estado: si es False, el
# número aún no ha sido encontrado...
encontrado = False

# el numero secreto...
secreto = random.randint(1, limite_derecho)

# el ciclo principal... siga mientras no
# haya sido encontrado el número, y la
# cantidad de intentos máxima no sea superada...
while not encontrado and intentos < cantidad_intentos:
    intentos += 1
    print('\nEl numero está entre', izq, 'y', der)

    # un valor para forzar al ciclo a ser [1, N]...
    # ... ver Ficha 7.
    num = izq - 1

    # carga y validación del número sugerido por el usuario...
    while num < izq or num > der:
        num = int(input('[Intento: ' + str(intentos) + '] => Ingrese: '))
        if num < izq or num > der:
            println('Error... le dije entre', izq, 'y', der, '...')

    # controlar si num es correcto y avisar en ese caso...
    if num == secreto:
        encontrado = True

    # ... pero si no lo es, ajustar los límites
    # del intervalo de búsqueda...
    elif num > secreto:
        der = num
    else:
        izq = num
```

```
# control final...
if encontrado:
    print('\nGenio!!! Acertaste en', intentos, 'intentos')
else:
    print('\nLo siento!!! Acabaron los intentos. El número era:', secreto)
```

Al inicio del programa se cargan dos valores que permiten configurar el arranque del juego:

- ✓ ***limite_derecho***: el límite derecho del intervalo donde estará el número secreto.
- ✓ ***cantidad_intentos***: la máxima cantidad de intentos que tendrá disponible el jugador.

Ambos valores son cargados por teclado antes de comenzar el ciclo general que controla el juego. Ese ciclo es el *motor del juego* propiamente dicho: muestra los mensajes de ayuda en pantalla a través de los valores de *izq* (cuyo valor inicial es 1) y *der* (valor inicial: *limite_derecho*). Dentro del ciclo se usa una *bandera de estado* llamada *encontrado* (valor inicial *False*) para marcar en todo momento si el número secreto fue encontrado o no.

El jugador carga un número en la variable *num* y se compara contra el número secreto (generado con *random.randint()* y almacenado en la variable *secreto*). Si *num* es igual a *secreto*, se marca ese hecho volviendo a *True* la bandera *encontrado*, y termina el juego. Si no, se determina si *num* es mayor o menor que *secreto*. Si fuera mayor, se ajusta el valor de *der* haciéndolo igual al número cargado (se reduce el intervalo de búsqueda desde la derecha). Y si fuera menor, se ajusta el valor de *izq* para hacerlo igual al número cargado (se reduce el intervalo de búsqueda desde la izquierda).

Un ciclo *while* forzado a ser *[1, N]* (vea el tema de los ciclos *[1, N]* en la Ficha 7) controla si el número fue adivinado o si se llegó al límite de intentos. En cualquiera de los dos casos, el proceso termina y se muestra el resultado adecuado en pantalla.

4.] Otro juego sencillo: Piedra, Papel y Tijera.

Hemos dicho que la implementación de juegos de computadoras es una de las actividades más desarrolladas (y rentables) del mundo, y que varios miles de millones de dólares se invierten y se ganan en esta actividad año tras año (a pesar de las copias ilegales distribuidas por la piratería). En la sección anterior hemos realizado un sencillo aporte a este segmento con el *Juego del Número Secreto*, y ahora proponemos la implementación de un clásico juego de manos: El *Juego de Piedra*, *Papel* y *Tijera*. El enunciado formal puede ser el siguiente:

Problema 23.) *Desarrollar un programa que implemente el conocido juego de manos llamado "Piedra , Papel y Tijera". En este juego participan dos jugadores, uno contra el otro. Cada uno de ellos, al mismo tiempo que el otro, debe mostrar con una de sus manos, alguna de las tres figuras básicas llamadas Piedra (la mano cerrada), Papel (la mano abierta y extendida) o Tijera (los dedos de la mano formando una V). Luego se comparan las figuras que cada uno mostró, y se determina el ganador de acuerdo a la siguiente secuencia de reglas generales:*

- *Piedra vence a Tijera (ya que Tijera se rompe si intenta cortar a Piedra)*
- *Tijera vence a Papel (ya que Tijera corta a Papel)*
- *Papel vence a Piedra (ya que Papel envuelve a Piedra)*

Típicamente, se juega "a la mejor de tres": los jugadores se enfrentan en tres jugadas, y se declara ganador al que gane en dos o más de esas tres.

Discusión y solución: El primer desafío en este tipo de programas es decidir la forma en que será modelado el sistema de pantallas para que el programa informe al usuario sobre lo que está ocurriendo y la forma de hacer las cargas de datos cuando el programa lo requiera (es decir, se debe decidir el modelado de lo que se conoce como la *Interfaz de Usuario* del programa o *User Interface (UI)* en inglés). En general, si la *UI* contendrá elementos visuales de alto nivel, como ventanas, botones, gráficos, posibilidad de uso del mouse u otros elementos de interacción, entonces se habla (en español) de la *Interfaz Gráfica de Usuario (IGU)* o bien (en inglés) de la *Graphic User Interface (GUI)*.

Obviamente, en las primeras semanas de un curso de introducción a la programación no se cuenta aún con conocimientos y formación en cuanto al empleo de elementos visuales de alto nivel (que Python provee), por lo que nuestro diseño de interfaz de usuario estará simplemente basado en la consola estándar.

En nuestra versión del juego, supondremos que *uno de los jugadores será el usuario humano*, y el *segundo jugador será la computadora*. Los estudiantes podrán luego dedicarse a intentar modificar esta versión para incluir la posibilidad de partidas "humano-humano" o "computadora-computadora". La idea entonces será la siguiente: cuando deba jugar el humano, el programa deberá solicitar que cargue por teclado un número entero, cuyo valor identificará a la figura que el humano quiere jugar (por ejemplo: *1 – Piedra*, *2 – Papel*, *3 – Tijera*). Los nombres de las tres figuras serán cadenas de caracteres en una tupla definida como una variable *descripcion* de uso general:

```
descripcion = 'Piedra', 'Papel', 'Tijera'
```

De este modo, el componente *descripcion[0]* queda asignado con la cadena *'Piedra'*, mientras que *descripcion[1]* queda asignado con *'Papel'* y *descripcion[2]* con *'Tijera'*. Sabiendo que la variable *descripción* existe, entonces el siguiente esquema de código permite hacer la carga por teclado de la figura elegida por el jugador humano:

```
humano = int(input('Ingrese 1 - Piedra, 2 - Papel o 3 - Tijera: '))
print('Usted eligió:', descripcion[humano - 1])
```

En este esquema, la variable *humano* se usa para almacenar el valor cargado por el usuario, que será un 1, un 2 o un 3. Luego de cargar ese valor, se muestra un mensaje informando cuál fue efectivamente la figura elegida. Note que el número que cargó el usuario en la variable *humano*, se usa como índice para entrar a la cadena que describe a la figura dentro de la tupla *descripcion*, *pero restando uno al valor* (ya que en una tupla, los índices comienzan desde el valor 0 y no desde el 1) [1]. Obviamente, en la versión definitiva del programa la carga de la jugada del jugador humano será realizada con un proceso de validación, para evitar que se ingresen números diferentes de 1, 2, o 3.

Una vez que se tiene cargada la jugada del usuario humano, debe jugar la computadora. La elección de la figura será realizada en este caso en forma aleatoria mediante la función *random.randint()* que hemos presentado en la Ficha anterior. El sencillo par de instrucciones que sigue muestra el mecanismo:

```
computadora = random.randint(1, 3)
print('La computadora eligió:', descripcion[computadora - 1])
```

La variable *computadora* se usa para almacenar el número que representa a la figura elegida en forma aleatoria. Ese número se obtiene con la invocación *random.randint(1, 3)* que obtiene un número entero aleatorio en el intervalo [1, 3]. Igual que antes, se muestra luego el nombre de la figura elegida, con la misma técnica usada en la elección del jugador humano.

El paso siguiente es determinar si hubo un ganador, y en ese caso, cuál de los dos jugadores ganó en esa jugada. Recordando que las variables *humano* y *computadora* contienen el *número* de la figura elegida por cada jugador, el siguiente fragmento de código compara esos números y determina el ganador:

```
if humano != computadora:
    if (humano == 1 and computadora == 3) \
            or (humano == 3 and computadora == 2) \
            or (humano == 2 and computadora == 1):
        ganador = 1
    else:
        ganador = -1
else:
    ganador = 0
```

La variable *ganador* se usa para guardar el resultado, codificándolo de la siguiente forma: si hubo un empate, el resultado se codifica como un 0. Si en cambio ganó el jugador humano, el resultado se almacena en la variable *ganador* como un 1. Y si el ganador fue la computadora, se almacena el valor -1 en la misma variable *ganador*. Esta manera de proceder asociando cada estado posible a distintos números enteros, es muy común en programación y de hecho lo hemos aplicado ya en este mismo programa al enumerar las descripciones de las figuras.

La instrucción condicional que determina si ganó el usuario humano o el usuario computadora, está basada en los números de las figuras. Sabemos que Piedra es 1 y que Tijera es 3, por lo cual si la expresión:

```
humano == 1 and computadora == 3
```

fuese cierta, significaría que el humano *(1: Piedra)* le ha ganado en esa jugada a la computadora *(3: Tijera)*. La condición evaluada es más extensa, ya que controla todas las combinaciones posibles de valores entre las variables humano y computadora, pero la lógica esencial es la misma que la que hemos descripto aquí. Confiamos en que los estudiantes podrán analizar el resto de esas combinaciones y asegurarse de que funcionan correctamente.

Note que en el planteo del código fuente la instrucción condicional completa aparece escrita en tres líneas diferentes. Esto es así debido a que la instrucción completa es tan larga que no cabe en una línea regular de 80 caracteres de ancho (que según las *recomendaciones PEP 8* debería ser el límite a respetar en una línea de código fuente). Si el programador decide partir una instrucción y seguir escribiendo la misma en la línea siguiente, se usa el separador \ (barra invertida) para hacer ese corte, en la forma que se mostró [1] [4].

Una vez que el programa ha determinado si hubo un ganador en una jugada, lo que sigue es llevar la cuenta del puntaje de cada jugador para poder al final determinar si hubo un ganador de la partida completa (recuerde: se hacen tres jugadas, y se declara ganador al que gane en dos o más de ellas). El script que sigue es el que lleva la cuenta:

```
if ganador == 1:
    contar_ganadas = contar_ganadas + 1
elif ganador == -1:
    contar_perdidas = contar_perdidas + 1
```

La idea general es simple: si la variable *ganador* quedó valiendo 1 entonces el ganador fue el humano y debe sumársele un punto. Pero si *ganador* quedó valiendo -1, el ganador de la jugada fue la computadora y el punto se debe sumar a ella. Las variables *contar_ganadas* y *contar_perdidas* son contadores que se usan para sumar los puntos que el humano ganó y perdió (respectivamente). Esas mismas variables permiten saber lo que ganó o perdió la computadora, tomándolas en forma invertida (lo que ganó el humano lo perdió la computadora, y lo que perdió el humano lo ganó la computadora: no es necesario usar dos variables adicionales). Obviamente, asumimos que el valor inicial de ambas variables misma es 0 (cosa que efectivamente haremos en el programa completo). Recuerde además, que la expresión *contar_ganadas = contar_ganadas + 1* es equivalente a *contar_ganadas += 1*.

El script anterior entonces, está usando dos contadores para llevar la cuenta de los puntos ganados o perdidos por el humano. Es importante que se comprenda que si se va a usar un contador en un programa, ese contador debe ser asignado con algún valor inicial válido (normalmente el 0) para garantizar que el conteo comience desde donde debe hacerlo y no desde un valor indefinido, o bien para garantizar que esa variable realmente exista antes de comenzar a contar. En nuestro caso *el valor inicial 0* de cada una se asigna (en principio) en el *programa completo*, en el mismo lugar donde se define la variable *descripción*:

```
print('Bienvenido al juego de Piedra - Papel - Tijera')

# inicializacion de variables descriptivas y contadores...
descripcion = 'Piedra', 'Papel', 'Tijera'
contar_ganadas = contar_perdidas = 0
```

Con todos estos procesos ya definidos, podemos controlar totalmente ***una*** jugada o ronda. Pero el juego consta de ***tres*** rondas, *con resultado al mejor de tres*. La simulación de las tres rondas (una luego de la otra), puede hacerse sin problema mediante un *ciclo for que ejecute tres iteraciones*:

```
# Inicializacion de variables descriptivas y contadores...
descripcion = 'Piedra', 'Papel', 'Tijera'
ganadas = perdidas = 0

# Simulacion de las tres rondas...
for ronda in range(1, 4):
    print('\nRonda', ronda)

    # Juega el humano...
    humano = 0
    while humano < 1 or humano > 3:
        humano = int(input('Ingrese 1 - Piedra, 2 - Papel o 3 - Tijera: '))
        if humano < 1 or humano > 3:
            print('Error... se pidió entre 1 y 3... cargue de nuevo...')
    print('Usted eligio:', descripcion[humano - 1])

    # Juega la computadora...
    computadora = random.randint(1, 3)
    print('La computadora eligió:', descripcion[computadora - 1])

    # Determinar si hubo un ganador y mostrar...
```

```
    if humano != computadora:
        if (humano == 1 and computadora == 3) \
                or (humano == 3 and computadora == 2) \
                or (humano == 2 and computadora == 1):
            ganador, mensaje = 1, 'Punto para el humano...'
        else:
            ganador, mensaje = -1, 'Punto para la computadora...'
    else:
        ganador, mensaje = 0, 'Empate en esta ronda...'
    print(mensaje)

    # Llevar la cuenta de los puntos ganados o perdidos...
    if ganador == 1:
        ganadas += 1
    elif ganador == -1:
        perdidas += 1
```

El proceso comienza definiendo las variables más generales (la tupla con la descripción de las figuras, y los dos contadores). Luego se activa el ciclo *for* para repetir tres veces tres veces la ejecución de una ronda completa (con lo que se logra entonces la simulación de las tres rondas del juego) y antes de terminar simplemente controla si alguno de los dos jugadores sumó al menos dos puntos, mostrando un mensaje con el resultado. Note que según el enunciado, sólo hay un ganador si alguno de los jugadores hizo dos o más puntos: en casos como que un jugador gane una ronda y se empate en las otras dos, el programa anunciará *que no hay ganador*.

Para terminar, note que si el segmento de código anterior hace todo este trabajo, *entonces en el programa completo lo único que nos restaría por hacer es controlar el resultado final al terminar el ciclo, e informar al ganador*. Mostramos el programa completo a continuación:

```
import random

# Titulo general...
print('Bienvenido al juego de Piedra - Papel - Tijera')

# Inicializacion de variables descriptivas y contadores...
descripcion = 'Piedra', 'Papel', 'Tijera'
ganadas = perdidas = 0

# Simulacion de las tres rondas...
for ronda in range(1, 4):
    print('\nRonda', ronda)

    # Juega el humano...
    humano = 0
    while humano < 1 or humano > 3:
        humano = int(input('Ingrese 1 - Piedra, 2 - Papel o 3 - Tijera: '))
        if humano < 1 or humano > 3:
            print('Error... se pidió entre 1 y 3... cargue de nuevo...')
    print('Usted eligio:', descripcion[humano - 1])

    # Juega la computadora...
    computadora = random.randint(1, 3)
    print('La computadora eligió:', descripcion[computadora - 1])

    # Determinar si hubo un ganador y mostrar...
    if humano != computadora:
        if (humano == 1 and computadora == 3) \
```

```
                or (humano == 3 and computadora == 2) \
                or (humano == 2 and computadora == 1):
            ganador, mensaje = 1, 'Punto para el humano...'
        else:
            ganador, mensaje = -1, 'Punto para la computadora...'
    else:
        ganador, mensaje = 0, 'Empate en esta ronda...'
    print(mensaje)

    # Llevar la cuenta de los puntos ganados o perdidos...
    if ganador == 1:
        ganadas += 1
    elif ganador == -1:
        perdidas += 1

# Determinacion del resultado final del juego...
if ganadas >= 2:
    print('\nWinner!! Usted ganó', ganadas, 'a', perdidas)
elif perdidas >= 2:
    print('Loser... Usted perdió', perdidas, 'a', ganadas)
else:
    print('\nNo hay ganador... jueguen de nuevo para decidir')

# cierre...
print('Fin del programa')
```

5.] Medición del tiempo de ejecución de un proceso en Python.

Si bien los programas que hasta ahora hemos desarrollado en el curso no tienen la complejidad ni el volumen de datos como para esperar tiempos de ejecución elevados o grandes demoras, el hecho es que a medida que se avance y se incorporen nuevas herramientas y técnicas para el desarrollo de programas (como ahora las estructuras repetitivas y más adelante la recursividad) esa situación irá cambiando.

Muchos programas tenderán a demorar cada vez más su tiempo de ejecución a medida que el tamaño del conjunto de datos crezca. Y si bien existen técnicas matemáticas formales (que oportunamente veremos) para estudiar el comportamiento de un algoritmo en circunstancias diversas, por ahora nos quedaremos en un plano más técnico y aplicativo como es el de *medir directamente el tiempo de ejecución de un proceso cualquiera en Python*.

En general, todo lenguaje provee funciones en sus librerías estándar que de una forma u otra permiten medir el tiempo que demora en ejecutarse un proceso. En Python, las funciones estándar provistas para esta tarea (y para otras relativas al manejo del tiempo) se encuentran incluidas en el módulo (o *librería de funciones*) llamado *time* [1]. Por lo tanto, un programa que necesite acceso a esas funciones deberá contener una *instrucción de importación* para ese módulo:

```
import time
```

Para proceder a la medición del tiempo que demora en ejecutarse un proceso cualquiera, los lenguajes de programación suele basase en lo que se designa como el *momento de origen* o *epoch*, que depende del sistema operativo, pero que en general suele ser el 1 de Enero de 1970, a las 0 horas. Si quiere conocer con precisión cuál es el *epoch* en su sistema, puede usar la función *time.gmtime()* pasándole un 0 como parámetro:

```
import time
print(time.gmtime(0))
```

La función *time.gmtime(t)* retorna una colección de valores que representan la fecha dada por la cantidad de segundos *t* tomada como parámetro. Si *t = 0*, entonces la fecha pedida coincide con el *momento epoch*. El pequeño script anterior, mostrará la siguiente salida (que aquí se muestra en dos líneas por razones de espacio) [1]:

```
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=0)
```

De todos modos, y como veremos, no debe preocuparnos mucho cuál fue exactamente el *momento epoch*.

Hasta la versión 3.3 de Python se contaba con la función *time.clock()* que retornaba un número en coma flotante indicando la cantidad de segundos que transcurrieron entre el *epoch* del sistema hasta el momento en que se invocó a la función. Pero a partir de la versión 3.3 de Python, esa función fue desestimada (se dice que se marcó como *obsoleta* o ***deprecated***) e incluso luego fue eliminada del módulo *time*.

A partir de la versión 3.3 de Python, se incluyó la función *time.perf_counter()* como una opción para reemplazar a *time.clock().* La función *time.perf_counter()* retorna un valor flotante que indica la cantidad de segundos que representa en ese instante el estado del reloj del sistema, pero no referido al *epoch*, sino al momento en que el procesador fue reiniciado. Por lo tanto, *time.clock()* retornaba un indicador de tiempo referido a un momento *único y real* (el momento *epoch*), pero *time.perf_counter()* retorna un lapso de tiempo que es relativo al momento en que el procesador activó su contador de tiempo (y ese contador se reinicia cada vez que el procesador se vuelve a encender o se reinicia).

Como sea, en la práctica la técnica básica para medir el tiempo que demora en ejecutarse un proceso consiste en invocar dos veces a *time.perf_counter()*: la primera vez inmediatamente antes de lanzar el proceso, y la segunda, inmediatamente después de finalizado el mismo. El tiempo total de ejecución del proceso medido, será entonces igual a la diferencia entre esos tiempos. Veamos un ejemplo muy simple:

```
import time

# medición del tiempo del proceso...
t1 = time.perf_counter()
x = 3 + 4**2
t2 = time.perf_counter()

# obtención del tiempo total...
tt = t2 - t1

# visualización del tiempo final
print('Tiempo total:', tt)
```

En el programa anterior, el proceso cuyo tiempo de ejecución se quiere medir es la sencilla instrucción *x = 3 + 4**2*. Para hacer la medición, se invoca a *time.perf_counter()* justo antes de lanzar esa instrucción, y también justo después de terminar de ejecutarla, almacenando los resultado en las variables *t1* y *t2*. La primera (*t1*) nos dice entonces cuánto tiempo (en segundos) transcurrió en el sistema hasta la primera llamada a *time.perf_counter()*; y la segunda (*t2*) nos indica el tiempo transcurrido hasta la segunda invocación. Por lo tanto, el cálculo de la diferencia ***tt = t2 – t1*** nos dará el tiempo exacto transcurrido entre ambas

llamadas, que no es otra cosa que el tiempo que demore en ejecutarse el proceso *x = 3 + 4**2*. En este caso, la ejecución del programa producirá una salida parecida a la siguiente:

```
Tiempo total: 4.105230186414398e-07
```

que como veremos en una Ficha posterior, equivale a un tiempo de 4.105230186414398 * 10^{-7} segundos, que es lo mismo que 0.000000410523018641439 8 segundos. Aquí debe entender que este número muy posiblemente será diferente cuando haga sus propias pruebas, ya que el tiempo de ejecución dependerá de muchos factores (el tipo de procesador, el nivel de ocupación del mismo al momento de ejecutar el script, etc.), pero puede esperar que el orden de magnitud del número obtenido sea muy similar (7 u 8 dígitos de mantisa con valor 0).

La técnica general para medir el tiempo de un proceso es esencialmente la que hemos mostrado, pero podemos ir un poco más lejos. Por lo pronto, está claro que el proceso a medir no tiene por qué ser una instrucción simple: puede medirse (por ejemplo) el tiempo de ejecución de una llamada a una función o de cualquier segmento de programa que pueda incluir entre dos llamadas a *time.perf_counter()*. El programa que sigue es una variante simple del programa anterior, y permite medir el tiempo de ejecución de un proceso que calcula y muestra el mayor entre dos números *n1* y *n2*:

```
import time

n1, n2 = 20, 10

# medición del tiempo del proceso...
t1 = time.perf_counter()
if n1 > n2:
    m = n1
else:
    m = n2
t2 = time.perf_counter()

# visualización del resultado del proceso...
print('Mayor:', m)

# obtención del tiempo total...
tt = t2 - t1

# visualización del tiempo final
print('Tiempo total:', tt)
```

La ejecución del programa produce una salida similar (en orden de magnitud) a la que mostramos aquí:

```
Tiempo total: 8.210460372828794e-07
```

Se puede apreciar que ahora la cantidad de dígitos con valor 0 a la derecha del punto sigue siendo 7, pero el primer decimal significativo vale ahora 8 (en lugar del 4 del ejemplo anterior). Conclusión: en este caso puntual, la ejecución de un proceso completo que incluye a una condición, le llevó a Python *el doble de tiempo* que ejecutar una instrucción simple que asigne en una variable el resultado de una expresión aritmética. Esto era esperable: no es lo mismo una sola instrucción simple, que una instrucción compuesta que incluye el chequeo de una condición y luego una asignación.

Aún más amplio resulta considerar la función *eval()* de la librería estándar de Python [1]. En su forma más simplificada, esta función toma como parámetro una *cadena de caracteres que exprese una instrucción* válida en Python, analiza esa expresión, y si efectivamente es válida la ejecuta y retorna el resultado de esa expresión. Ejemplo:

```
x = eval('3 + 4**2')
print('x:', x)
```

La salida en consola del script anterior será algo como *x: 19*.

De esta forma, podemos plantear un script que tome una cadena cualquiera (asignada o cargada por teclado) pero que represente una expresión válida en Python, y retorne el tiempo de ejecución de la expresión:

```
import time

expr = 'min(3, 7)'
t1 = time.perf_counter()
r = eval(expr)
t2 = time. perf_counter ()
print('Menor:', r)
print('Tiempo de ejecución:', t2 - t1)

expr = input('Ingrese una expresión válida en Python: ')
t1 = time. perf_counter ()
r = eval(expr)
t2 = time. perf_counter ()
print('Resultado:', r)
print('Tiempo de ejecución:', t2 - t1)
```

Si se ejecuta el script anterior, comenzará mostrando una salida de la parecida a esta:

```
Menor: 3
Tiempo de ejución: 3.489491498982916e-05
```

Luego pedirá que se cargue por teclado una expresión válida en Python. Si el usuario carga una cadena cualquiera que efectivamente represente una expresión ejecutable en Python (como puede ser la expresión lógica '*3 >= 8*') entonces el programa ejecutará esa expresión, mostrará el resultado de la misma (*False* en este caso…) y luego el tiempo insumido en esa ejecución:

```
Ingrese una expresión válida en Python: 3 >= 8
Resultado: False
Tiempo de ejecución: 7.389511409705563e-05
```

6.] Tipos generales de ciclos y su implementación en Python.

Hemos indicado en la *Ficha 6* que el *ciclo while* de Python es un ciclo de la forma *[0, N]* y hemos explicado que esto quería decir que el bloque de acciones podría ejecutarse entre 0 y *N* veces: si la condición de control es falsa en la primera evaluación, el bloque del ciclo no se ejecutará, pero se ejecutará una cantidad finita *N* de veces mientras la condición de control sea cierta.

Si bien el ciclo *for* de Python tiene una estructura general orientada hacia el recorrido o iteración de estructuras de datos como tuplas, rangos, cadenas o listas, puede verse con relativa sencillez que este ciclo *también es de la forma [0, N]*: intuitivamente, la primera aproximación se basa en lo que ocurre cuando se usa un *for* para intentar recorrer una

secuencia vacía: el bloque de acciones no se ejecuta y el programa salta directamente a la primera instrucción ubicada a la salida del ciclo. Podemos verlo claramente en este ejemplo:

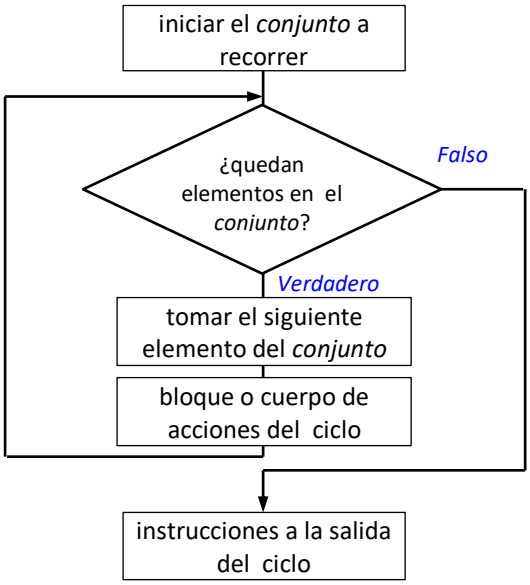
```
conjunto = ()
for x in conjunto:
    print('x:', x)

print('Programa terminado')
```

En el script anterior, la variable *conjunto* se inicializa como una *tupla vacía*, y luego se usa un ciclo for para recorrerla con la variable *x* como iteradora. En el bloque del ciclo se intentan mostrar los elementos de la tupla, uno a uno, y al terminar el ciclo se muestra un mensaje de terminación. Sin embargo, como la tupla *conjunto* está vacía, el ciclo no ejecuta nunca el bloque de acciones y sólo muestra el mensaje de terminación al final.

Sea cual fuese la estructura o colección a iterar, el ciclo *for* no ejecutará su bloque de acciones si la misma estuviese vacía. En el ejemplo anterior, si la variable *conjunto* se inicializa como una cadena vacía (*conjunto = ''*) o bien como un rango vacío (*conjunto = range(1, 0)*) el efecto será el mismo. En términos muy generales, el ciclo *for* entonces tiene una lógica esencial que puede graficarse de esta forma (que es claramente *[0, N]*), usando la diagramación clásica:

Figura 2: Diagrama de flujo de un ciclo *for* (diagramación clásica) como ciclo *[0, N]*.



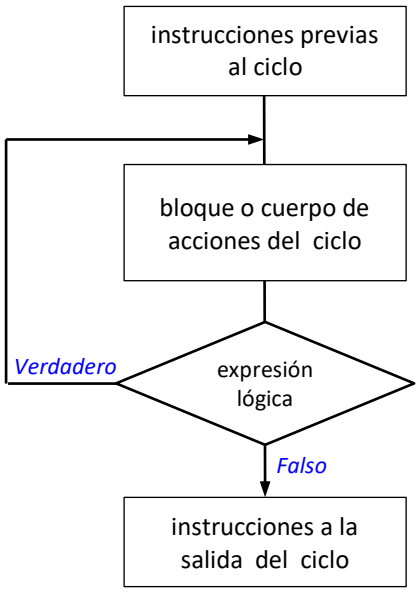
Como se puede ver, si el conjunto de datos a recorrer estuviese inicialmente vacío, la condición de control del ciclo entregaría un *falso* en el primer chequeo, ya que el conjunto no tendría en ese caso ningún elemento.

Por lo tanto, los dos tipos de instrucciones repetitivas provistas por Python son de la forma *[0, N]*. Sin embargo, existe al menos un segundo tipo de ciclo general (no provisto en forma directa por Python) que se conoce como ciclo *[1, N]*. En esencia, un ciclo *[1, N]* es aquel que *siempre ejecuta su bloque de acciones, **al menos una vez*** (y de allí la designación de *[1, N]*).

Para que al menos una vez el bloque de acciones del ciclo se ejecute siempre, los ciclos de este tipo se plantean en forma ligeramente diferente a los ciclos *[0, N]*: en los *[0, N]* el

chequeo de la expresión lógica de control se escribe ***antes*** que el bloque de acciones, mientras que en los ciclos *[1, N]* se escribe ***después*** del bloque. El siguiente diagrama en forma clásica muestra la lógica esencial de funcionamiento de un *ciclo [1, N]* [2]:

Figura 3: Diagrama de flujo de un ciclo *[1, N]* (diagramación clásica).



El hecho es que no existe en Python una forma directa de ciclo *[1-N]* (como sería el *do – while* de C, C++ y Java o el *repeat – until* de Pascal), aunque eso no representa un problema ya que *siempre se puede emular un ciclo [1-N]* desde un ciclo *[0-N] forzando a que la expresión lógica de control sea cierta la primera vez que se evalúa*, como en el ejemplo siguiente, que carga una sucesión de números positivos hasta que se ingrese un número impar, mostrando al final la suma de los valores pares:

```
sp = 0

hecho = False
while hecho == False:
   n = int(input('Cargue un número (con impar corta:') )
   if n%2 == 0:
         sp += n
   else:
         hecho = True

print('La suma de los valores pares es:', sp)
```

Si bien el ciclo usado en el script anterior es un *while*, que en Python es *[0, N]*, el esquema de repetición que se muestra se comporta como *[1, N]* en la forma en que está planteado: la variable *hecho* se inicializa en *False* inmediatamente antes de iniciar el ciclo, y en la expresión lógica de control del ciclo se pregunta si *hecho* es *False*. Así escrito, *no hay ninguna oportunidad de que la expresión lógica sea falsa en el primer chequeo*, por lo que el bloque de acciones del ciclo se ejecutará inexorablemente al menos una vez. Si en la primera vuelta del ciclo se cargase por teclado un valor impar en la variable *n*, el valor de la variable *hecho* cambiará a *True* y el ciclo cortará… pero al menos una vez el bloque ya habrá sido ejecutado.

La aplicación de un tipo de ciclo u otro en un programa depende de las necesidades del programador: sabemos que si se conoce la cantidad exacta de repeticiones a realizar, suele

ser conveniente el uso de un *for*, pero nada obliga a ello ya que hemos visto que un *while* puede usarse también. Si se desconoce la cantidad de repeticiones, el *while* suele ser el ciclo más cómodo y esto es efectivamente así en Python, pero en otros lenguajes cualquier ciclo puede usarse en lugar de cualquier otro. Ahora sabemos que también puede plantearse un ciclo *while* para que se comporte en forma *[1, N]*. Y de nuevo, la decisión de hacer una u otra cosa depende del programador.

Sabiendo lo anterior, nada impide que nos preguntemos en qué situaciones reales un programador podría preferir la aplicación de un ciclo *[1, N]* en lugar de un *[0, N]*. Digamos que hay por lo menos una: la carga de valores desde el teclado, *validando esa carga* para impedir que se ingresen valores incorrectos.

Es muy común que en ciertas ocasiones se requiera que un programa cargue datos por teclado, pero de tal forma que se garantice que esos datos sean correctos. Por ejemplo: si en un programa se pide cargar el sueldo de un empleado, o la nota obtenida por un alumno en una materia, es de esperar que el valor que se cargue por teclado no sea negativo (un sueldo negativo o una nota negativa son situaciones que están claramente fuera de las hipótesis válidas para los datos del programa) [2].

La salida más cómoda es suponer que el usuario *no cargará* valores inadecuados, pero la solución más profesional es que el programa de alguna forma haga un control de los valores que se cargan (lo que forma parte de una estrategia que suele designarse como *programación defensiva*). Si el usuario carga un valor negativo cuando se esperaba que fuera cero o positivo, el programa *debería rechazar dicho valor*, y *volver a pedir la carga* del mismo hasta que finalmente el usuario introduzca un valor correcto. Ese proceso de verificación de un dato cargado por teclado, suele llamarse *proceso de validación de datos* y es usual realizarlo con un ciclo *[1, N]*: esto es así porque se espera que el usuario cargará *al menos un valor*. Si el primer valor cargado es correcto, el ciclo no pedirá otro valor y continuará el programa normalmente. Pero si el primer valor cargado fuera incorrecto, el ciclo hará otra repetición y volverá a pedir un valor, y así seguirá hasta que el valor sea correcto.

En el siguiente programa, se aplican *dos procesos de validación* para cargar por teclado el *sueldo* de un empleado y la *edad* de ese empleado, asegurando que ninguno sea negativo ni inapropiado. Ambos procesos validan la carga sólo terminan cuando el valor cargado finalmente se ingresa en forma correcta (aunque la lógica de cada validación es diferente en cada esquema):

```
print('Validación de carga de datos')
nombre = input('Ingrese su nombre: ')

edad = -1
while edad < 0 or edad >= 120:
    edad = int(input('Edad (mayor o igual a 0 y menor a 120, por favor): '))
    if edad < 0 or edad >= 120:
        print('Incorrecto... se pidió >= 0 y < 120... cargue otra vez...')

sueldo = 0
while sueldo <= 0:
    sueldo = int(input('Ingrese su sueldo (mayor a 0, por favor): '))
    if sueldo <= 0:
        print('Incorrecto... se pidió mayor a 0... cargue otra vez...')

print('Los datos registrados son:')
print('Nombre:', nombre, '- Edad:', edad, '- Sueldo:', sueldo)
```

Ambos procesos de validación usan un ciclo *while* forzándolo a actuar en modo *[1, N]*. Para la carga de la *edad* se inicializa la variable *edad* con el valor -1 para que la condición de control del ciclo sea verdadera en el primer chequeo, mientras que en la carga del *sueldo* se inicializa la variable *sueldo* en 0 para lograr el mismo efecto. Dentro del bloque de acciones de cada ciclo se carga un valor por teclado para esas variables, los que serán controlados por cada ciclo, y si fuesen incorrectos se volverán a pedir. Para reforzar al usuario el hecho de haber cometido un error (y que no lo deje pasar en forma desapercibida), el bloque de acciones de cada ciclo contiene también una instrucción *if* que hace el mismo chequeo que el ciclo, pero en este caso sólo para activar un *print()* avisando del error e invitando al usuario a volver a ingresar. La repetición del proceso de carga es implementado por el ciclo, pero el mensaje de error surge de la instrucción condicional.

Bibliografía

[1] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: https://docs.python.org/3/.

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

[3] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.

[4] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.