



Ficha 7

Estructuras Repetitivas: El Ciclo *for*

1.] El ciclo *for* en Python.

El *ciclo for* suele ser el más práctico cuando se conoce previamente la cantidad de repeticiones a realizar. El ejemplo que vimos en la *Ficha 6* para cargar tres números y contar cuántos eran negativos, es un caso ideal (como veremos) para resolver mediante un *ciclo for* porque conocemos la cantidad de repeticiones (tres, o las que sea que se indiquen).

Sin embargo, el *ciclo for* en Python está formulado para hacer mucho más que simplemente ejecutar en forma repetida un bloque de acciones durante un número determinado de vueltas. En realidad es un ciclo especialmente diseñado para *recorrer secuencias* como *tuplas*, *cadenas de caracteres*, *rangos*, *listas*, etc. (presentadas en la *Ficha 3*) recuperando de a un elemento por vez (es decir, un elemento en cada vuelta) [1].

Por ejemplo, si definimos una *tupla* con tres nombres de esta forma:

```
nombres = ('Juan', 'Pedro', 'Maria')
```

es posible plantear un *ciclo for* de manera que use una variable para recuperar de a un nombre por vez en cada vuelta. El siguiente script de código Python define la *tupla* y muestra de a un nombre por vez en la consola de salida:

```
nombres = ('Juan', 'Pedro', 'Maria')
for nom in nombres:
    print(nom)
```

Analicemos el script línea por línea. Primero se define una *tupla* llamada *nombres* con los valores de las cadenas *'Juan'*, *'Pedro'* y *'Maria'*. Luego se lanza un *ciclo for* que define la variable *nom* y está planteado para ejecutar en forma repetida el bloque de acciones asociado a él. En este caso, el *ciclo for* repetirá la ejecución de la única instrucción que tiene en su bloque: *print(nom)*.

En cada repetición el propio ciclo se encarga de colocar en la variable *nom* un elemento de la *tupla* en el orden en que han sido alojados en ella y finaliza cuando ya no tiene elementos que tomar de la *tupla*. La cantidad de *vueltas* (o *iteraciones* o *repeticiones*) que hace este ciclo en el ejemplo es tres, porque la *tupla* que está recorriendo tiene exactamente tres elementos. La salida en consola del script anterior será algo como:

```
Juan
Pedro
Maria
```

La primera línea de la instrucción *for* (en el ejemplo: *for nom in nombres*) se conoce como la *cabecera del ciclo*. Típicamente la cabecera incluye la definición de una variable (*nom* en este caso) que será usada para ir almacenando uno a uno los valores que se recuperen automáticamente desde la estructura de datos que se pretende recorrer (la *tupla nombres* en este caso).

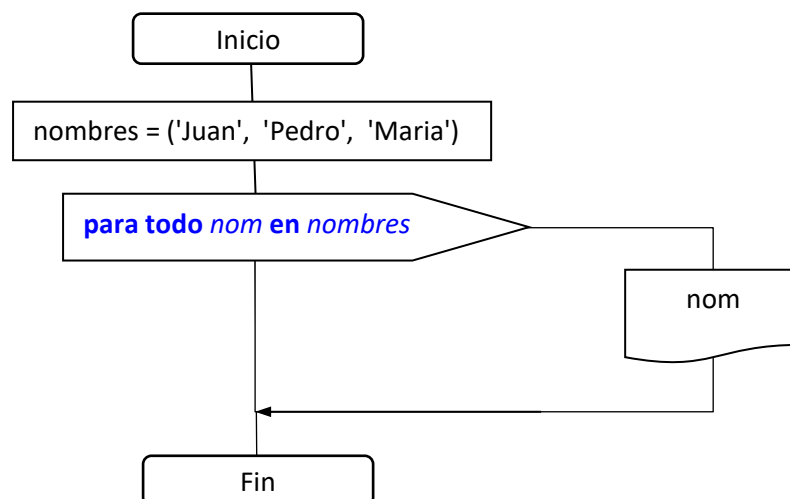
La secuencia de instrucciones que el ciclo debe ejecutar en forma repetida, se conoce como el *bloque de acciones* del ciclo, y obviamente debe escribirse indentado hacia la derecha de la cabecera. En el ejemplo el bloque sólo contiene una instrucción de visualización (`print(nom)`), pero podría contener tantas instrucciones como el programador disponga.

En cuanto a su representación en un diagrama de flujo, el *ciclo for* típicamente se ha representado siempre con el símbolo no estándar que hemos mostrado para el *ciclo while*, con diversas variantes, escribiendo en su interior los elementos que definen la *cabecera del ciclo*, aunque sin necesidad de usar sintaxis estricta de Python o de otro lenguaje: puede escribir los elementos de la cabecera en castellano, en forma resumida, o de la forma que prefiera, siempre que deje en claro lo que necesita que haga el ciclo.

La *Figura 1* muestra el diagrama de flujo que representa al sencillo script anterior. Como se ve en el diagrama, el símbolo que representa al ciclo contiene en su interior la definición informal de la cabecera del mismo, y el bloque de acciones se dibujó *indentada hacia la derecha* (en forma similar a lo que se hace con la rama verdadera de una condición en un diagrama, y respetando el hecho que luego ese bloque será efectivamente indentado hacia la derecha en el código fuente del programa).

El gráfico busca expresar que las instrucciones del bloque se ejecutarán en forma repetida una y otra vez hasta que se hayan procesado todos los elementos de la tupla *nombres*. Para indicar el final del ciclo se dibuja una línea recta que sale desde abajo del símbolo del ciclo y continúa con la línea descendente del diagrama original (de nuevo, en forma similar a lo que se hace al graficar una *instrucción condicional simple*, sin rama *else*).

Figura 1: Diagrama de flujo del script para mostrar tres nombres con un ciclo *for*.



Cuando se tiene una estructura de datos cualquiera (por ejemplo, una *secuencia* de cualquier tipo) la operación de recorrer esa estructura y procesar de a uno sus elementos se designa en forma general como *iterar* la estructura. En forma similar, la variable que se use para ir tomando uno a uno los valores de la estructura a medida que se itera sobre ella, se conoce como *variable iteradora* (en el ejemplo, el *for* está *iterando* sobre la tupla *nombres*, y está usando a la variable *nom* como *variable iteradora*).

Por lo tanto, se puede decir que el ciclo *for* en Python está esencialmente previsto para la *iteración de estructuras de datos*. El siguiente ejemplo muestra un *ciclo for* iterando sobre



todos los caracteres de una cadena contenida en la variable *ciudad*. El *for* usa la *variable iteradora* *c* para ir almacenando una copia de los caracteres de la cadena, a razón de uno por vuelta del ciclo y en el orden en que aparecen, y también va mostrando cada caracter por separado en la consola de salida:

```
ciudad = 'Cordoba'
for c in ciudad:
    print(c)
```

Note que la variable *ciudad* es de tipo *cadena de caracteres*, pero la variable iteradora *c* va tomando de a un caracter por vez. El ciclo finaliza su ejecución cuando se terminan de procesar todos los caracteres de la cadena.

Nos interesa ahora definir cómo se puede definir un *ciclo for* que nos permita realizar una *determinada cantidad de repeticiones* cuando nuestra necesidad no sea procesar uno a uno los elementos de una secuencia. En otras palabras, queremos plantear un *ciclo for* para resolver problemas como el de *cargar tres números y contar cuántos de esos números eran negativos* (ver Ficha 06).

La forma básica de hacerlo consiste en crear un tipo de secuencia numérica designada como *rango* (o *range*), de forma que el ciclo itere sobre ella. Una secuencia de tipo *range* es una sucesión *inmutable* (sus componentes no pueden ser cambiados una vez creada la secuencia) de valores numéricos en un intervalo definido, que puede ser creada fácilmente con la función *range()* [1]. Por ejemplo, el siguiente script usa un *ciclo for* para mostrar una secuencia numérica que contiene todos los números enteros entre el 1 y el 5:

```
for i in range(1, 6):
    print(i)
```

El ciclo del ejemplo anterior realiza cinco repeticiones y en cada una ejecuta la instrucción *print(i)* para mostrar el valor que tenga en esa vuelta la variable iteradora *i*. La variable *i* comienza con el valor 1 en la primera vuelta, y luego en cada una de las iteraciones del ciclo cambia automáticamente su valor para tomar el siguiente del *range*. Cuando la variable *i* llega a tomar el valor 5 (o sea, el último contenido en el *range*), el ciclo procesa ese valor y se detiene. Observe que la variable *i* nunca llega a tomar el valor 6 con el cual se invocó a la función *range()* para crear la secuencia.

Note entonces que la secuencia generada con *range(a, b)* incluye al valor *a* pero no incluye al valor *b*. Por supuesto, los valores inicial y final de la secuencia pueden venir almacenados en variables, y los valores inicial y final pueden ser cualesquiera. Si se invoca a *range()* con un solo parámetro, se asume que ese es el valor final (sin incluirlo) de la secuencia a generar, y se toma el valor inicial como 0(cero):

```
# muestra los números del 0 al 5...
for i in range(6):
    print(i)
```

La función *range()* acepta si es necesario un tercer parámetro que indica el *incremento* a usar para pasar de un valor a otro de la secuencia generada (este incremento puede ser positivo o negativo). Por defecto se asume que ese incremento es 1(unos). El mismo script que usamos para mostrar los números del 1 al 5, podría haber sido planteado así:

```
# muestra los números del 1 al 5...
for i in range(1, 6, 1):
    print(i)
```



En general, un range r creado con la instrucción $r = \text{range}(\text{start}, \text{stop}, \text{step})$ representa una secuencia numérica inmutable que contiene sólo a los números dados por las siguientes dos expresiones [1]:

1. Si $\text{step} > 0$ entonces el contenido de cada elemento $r[i]$ se determina como:

$$r[i] = \text{start} + \text{step} * i \quad \text{donde} \quad i \geq 0 \quad \text{y} \quad r[i] < \text{stop}.$$

2. Si $\text{step} < 0$ entonces el contenido de cada elemento $r[i]$ se determina como:

$$r[i] = \text{start} + \text{step} * i \quad \text{donde} \quad i \geq 0 \quad \text{y} \quad r[i] > \text{stop}.$$

Por lo tanto, si $r = \text{range}(1, 10, 2)$ entonces r representará una secuencia numérica de la forma (1, 3, 5, 7, 9) ya que según las fórmulas anteriores:

step = 2 es mayor a 0 por lo tanto:

```
r[0] = start + step*0 = 1 + 2*0 = 1    (y como 1 < stop (=10), se acepta)
r[1] = start + step*1 = 1 + 2*1 = 3    (y como 3 < stop (=10), se acepta)
r[2] = start + step*2 = 1 + 2*2 = 5    (y como 5 < stop (=10), se acepta)
r[3] = start + step*3 = 1 + 2*3 = 7    (y como 7 < stop (=10), se acepta)
r[4] = start + step*4 = 1 + 2*4 = 9    (y como 9 < stop (=10), se acepta)
```

Este *range* no contiene a todos los números entre 1 y 10: sólo los que cumplen con la primera de las fórmulas de cálculo, y su tamaño es 5 (y no 10).

En el siguiente ejemplo, entonces, el *ciclo for* muestra los primeros n números pares, comenzando desde el cero:

```
n = int(input('Cuántos pares quiere mostrar?: '))
for par in range(0, 2*n, 2):
    print(par)
```

Si el valor cargado en n fuese, por ejemplo, el 4, entonces el *ciclo for* del script anterior generará el *range* de 4 números (0, 2, 4, 6) y comenzará haciendo que la variable *par* valga 0, mostrando el 0 en la primera vuelta. Si el tercer parámetro fuese 1 en lugar de 2, el ciclo mostraría ocho números, ya que en ese caso, el *range* efectivamente contendría a los ocho números del 0 al 7. Recuerde que si efectivamente desea un paso de avance igual a 1, entonces no es necesario incluir el tercer parámetro: Python asumirá que su valor es 1.

Como el incremento o paso de avance puede ser negativo, entonces el siguiente script simplemente muestra los números del 5 al 1 en forma descendente:

```
for i in range(5, 0, -1):
    print(i)
```

Aquí la invocación *range(5, 0, -1)* crea la secuencia de números (5, 4, 3, 2, 1) (en base a la fórmula 2 de cálculo de contenidos de un *range*) y luego esa secuencia se itera usando la variable i del *ciclo for*, que comienza valiendo 5. Cuando i haya tomado todos los valores del *range*, el ciclo se detendrá.

Es interesante notar que el mismo script anterior, pero sin el incremento de -1 indicado en la función *range()* en su tercer parámetro, no mostrará nada en la consola de salida ya que el ciclo no hará ninguna repetición:

```
for i in range(5, 0):
    print(i)
```

Esto se debe a que en este caso el incremento o paso de avance es igual a 1, que es positivo. Como en este caso se aplica la fórmula 1, los valores generados serían de la forma $(5 + 1*0, 5 + 1*1, 5 + 1*2, \dots) = (5, 6, 7, \dots)$ y como ninguno de ellos es menor a 0 (que es el valor *stop*) y



la fórmula 1 exige que cada valor $r[i]$ calculado sea menor a *stop*, entonces el *range* queda vacío y el ciclo no hace repetición alguna.

El caso es que la función *range()* nos permite definir muy simplemente una secuencia de números enteros sucesivos lo que, a su vez, es muy útil a la hora de plantear un *ciclo for* que permita una determinada cantidad de iteraciones. Si volvemos al problema de *cargar tres números y contar los negativos*, es fácil ver que ahora podemos usar un *ciclo for* ajustado para hacer tres repeticiones y ahorrar muchas líneas de código. El siguiente programa aplica la idea (y es un replanteo del programa de la Ficha 06 que se basaba en un *while*):

```
a = 0
for i in range(1, 4):
    num = int(input('Ingrese un número: '))
    if num < 0:
        a += 1

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

Al ejecutarlo vemos que en tres ocasiones nos pide ingresar un número y al finalizar nos indica cuántos de esos números eran negativos. La cabecera del *ciclo for* crea un *range* con los números (1, 2, 3) (recuerde que en este caso el 4 no queda incluido) y define la *variable iteradora i* para recorrer ese *range*. El *bloque de acciones* del ciclo consta de una instrucción de carga por teclado para la variable *num*, el chequeo del valor *num* para saber si es negativo, y el incremento de un contador en caso de serlo. Hay dos detalles interesantes a rescatar:

1. Las *instrucciones del bloque de acciones* del ciclo de este programa, estaban escritas tres veces en el programa original sin ciclos de la *Ficha 06*. Ahora, esas instrucciones se escribieron una sola vez, y el ciclo se encarga de repetir su ejecución tantas veces como se haya dispuesto.
2. La variable iteradora *i* que va tomando los valores del *range* no se utiliza dentro del bloque de acciones del ciclo en este caso. Tenemos aquí un buen ejemplo de cómo definir un *ciclo for* que lance una cantidad predefinida de repeticiones, aunque en realidad no nos resulte relevante trabajar con los valores del *range* usado para controlarlo.

Note que *si la variable iteradora no va a usarse explícitamente dentro del bloque de acciones*, sino que será usada sólo para controlar la cantidad de repeticiones, entonces puede usarse *cualquier range en ese ciclo*, siempre que tenga *la misma cantidad de elementos que el original*. El ciclo mostrado en el programa anterior ejecuta tres iteraciones, pero podría ser reemplazado por el siguiente (*que también ejecuta tres iteraciones*), sin cambio alguno en el resultado del programa:

```
for i in range(11, 14):
    num = int(input('Ingrese un número: '))
    if num < 0:
        a += 1
```

Por otra parte, nada impide que el programa que se acaba de mostrar sea replanteado para que en lugar de sólo tres números, nos pida tantos como el usuario necesite. El único cambio a realizar, es incluir la carga por teclado de una variable *n* antes del ciclo, en la cuál se almacene la cantidad de números que se desee ingresar, y luego cambiar la llamada a *range()* para que genere una secuencia entre 0 y *n-1* (lo cual sería algo como: *range(n)*):

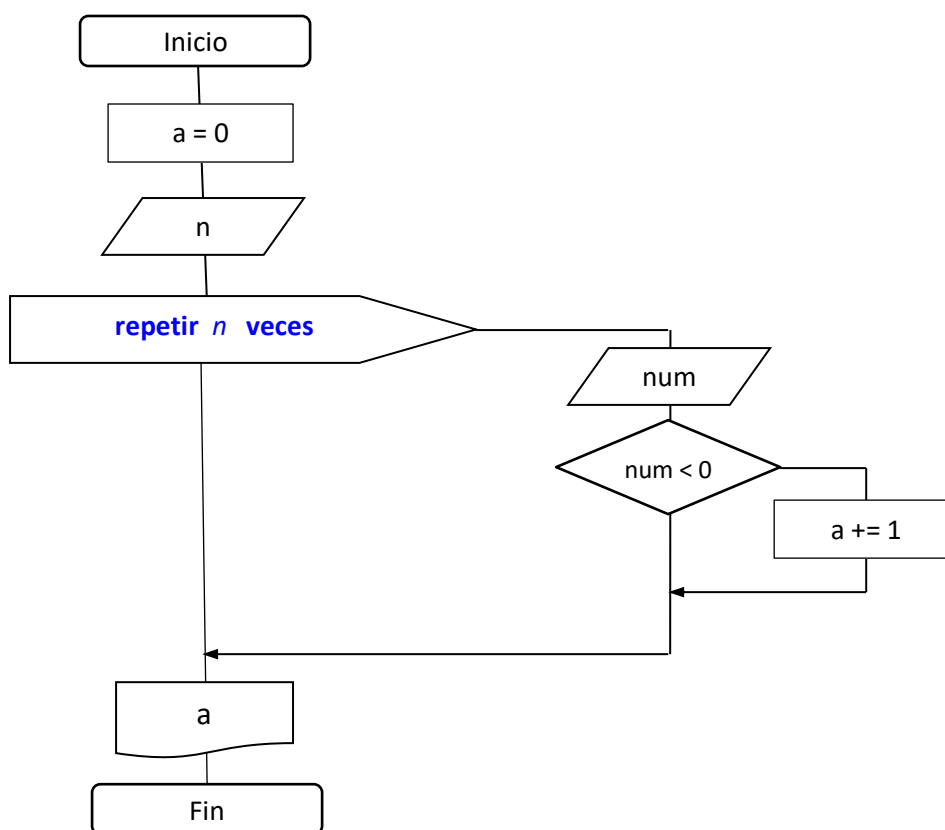
```
a = 0
n = int(input('Cuántos numeros quiere procesar?: '))
```

```
for i in range(n):  
    num = int(input('Ingrese un numero: '))  
    if num < 0:  
        a += 1  
  
print('Cantidad de negativos cargados:', a)
```

Si se quisiera procesar cinco números, el valor cargado en n sería 5 y la invocación $\text{range}(n)$ sería equivalente a $\text{range}(5)$. Esto crea un *range* de la forma (0, 1, 2, 3, 4) que comienza en 0 y termina en 4, pero contiene exactamente 5 valores... que es la cantidad de iteraciones que necesitábamos para el *ciclo for*. No es obligatorio que el *range* comience en el valor 1 y termine en n ($=5$): sólo necesitamos que haya n ($=5$) elementos en el *range* a iterar.

El siguiente diagrama de flujo representa la solución final para el problema, usando un *ciclo for* con los detalles explicados:

Figura 2: Diagrama de flujo (problema del conteo de negativos) con un ciclo *for* de n iteraciones



La versión final del programa deja clara la ventaja de usar un ciclo en lugar de una estructura secuencial: no importa cuántos números se necesite procesar (no importa el valor de n): el programa es siempre el mismo y desde el punto de vista de la escritura del código fuente da igual si n es 3, o 5, o 100, o 100000... El *bloque de acciones del ciclo* contiene lo que debe hacerse con *uno* de los números, y el ciclo repite la ejecución de ese bloque tantas veces como números haya.

No ocurre lo mismo con una *estructura secuencial* en la que la cantidad de instrucciones a escribir en el programa guarda una relación directa con la cantidad de datos n a procesar: si n es 5, el programador escribirá cinco veces el bloque de acciones. Si n es 100, el programador escribirá cien veces ese bloque... y si todavía no comprende la diferencia,



intente escribir el programa anterior mediante una estructura secuencial y sin el ciclo, pero suponiendo que serán $n = 10000$ los números a procesar. Habrá que armarse de paciencia... y además esperar a que luego no cambie el requerimiento y se pida procesar $n = 15000$ números en lugar de 10000.

A modo de cierre de esta introducción al uso del *ciclo for*, analicemos el siguiente problema:

Problema 18.) Sea n un número entero mayor o igual a 0. El **factorial** del número n (denotado algebraicamente como $n!$) es el producto de todos los números enteros en el intervalo $[1, n]$ si $n > 0$. En caso que $n = 0$, entonces se define $0! = 1$. Más formalmente:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1) * (n-2) * \dots * 3 * 2 * 1 & \text{si } n > 0 \end{cases}$$

A modo de ejemplo:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

$$7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040$$

Se pide desarrollar un programa en Python que cargue por teclado un número n y calcule y muestre su factorial.

Discusión y solución: Este es un problema clásico para plantear en base a un ciclo que ejecute un número predeterminado de iteraciones. El producto de todos los números enteros entre 1 y n requiere un ciclo que sea capaz de tomar todos esos números, uno a uno, y multiplicarlos en forma acumulativa, esto es, multiplicar dos de ellos, y al resultado volver a multiplicarlo por el que sigue, y este nuevo resultado multiplicarlo a su vez por el siguiente, hasta terminar con los n números del intervalo¹.

Está claro que el intervalo de todos los números enteros entre 1 y n puede obtenerse con `range(1, n+1)`: así invocada, la función `range()` nos permitirá disponer de una secuencia de números entre 1 y n (recuerde que $n+1$ no está incluido en el `range` generado). Por lo tanto, un *ciclo for* con la siguiente cabecera:

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

nos permitirá tomar uno por uno los números del `range(1, n+1)` asignándolos a razón de uno por vuelta en la *variable iteradora* i . Esto nos dará en la variable i todos los factores 1, 2, 3, ... $(n-2)$, $(n-1)$, n que necesitamos para el cálculo.

El producto acumulativo puede hacerse con sencillez. Si la variable f contiene un valor cualquiera, entonces la expresión de acumulación $f = f * i$ (que como sabemos es equivalente en Python a la expresión $f *= i$) hará que el valor actual de f se multiplique por el valor de i , y el resultado se vuelva a asignar en f .

¹ El cálculo de un factorial sería un simple pasatiempo para una superdotada en matemáticas, como la niña sobre la cual trata la película *Gifted* (traducida en español como *Un don excepcional*) del año 2017, dirigida por Marc Webb y protagonizada por McKenna Grace (la niña en cuestión) y Chris Evans (el padre). La película trata sobre la pequeña Mary, que demuestra tener una genial facilidad para los cálculos y la comprensión de las matemáticas (posiblemente por herencia de su madre, ya fallecida), y se produce un esperable conflicto entre su padre (que quiere darle una vida tan normal como sea posible) y la abuela de la niña (que busca fortalecer su don y pretende enviarla a toda costa a algún instituto para niños con habilidades extraordinarias). Una película entretenida, humana y directa.



En base a las ideas generales anteriores, el programa que se muestra a continuación permite calcular y mostrar el valor del factorial del número n :

```
n = int(input('Valor de n (>=0 por favor): '))

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

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

Una vez cargado por teclado el valor de n , el proceso comienza asignando el valor 1 en la variable f . Como f será usada como *variable de acumulación multiplicativa*, su valor inicial es 1: no puede ser 0 ya que cualquier producto que tenga a f como factor, resultaría indefectiblemente en un resultado de 0. En general, si una variable será usada como *acumulador sumativo*, se esperaría que su valor inicial sea cero (el neutro de la suma); pero si la variable será usada como *acumulador multiplicativo* entonces se esperaría que su valor inicial fuese 1 (el neutro de la multiplicación).

El *ciclo for* del programa anterior crea un *range()* con los números del 1 al n , y luego itera sobre ese *range* con la variable i . El bloque de acciones del ciclo, simplemente ejecuta la expresión $f *= i$ (que es lo mismo que $f = f*i$) almacenando el resultado en la misma f . Está claro entonces que en la primera repetición del ciclo, hará $f = f*i = 1*1 = 1$ sin cambiar el valor de f . Pero en la segunda vuelta, el producto será $f = f*i = 1*2 = 2$, cambiando f al valor 2. La siguiente iteración hará $f = f*i = 2*3 = 6$, y así sucesivamente hasta la última vuelta, en la que finalmente f quedará asignada con el valor del factorial de n (el resultado de la última multiplicación) que será mostrado en pantalla.

Un detalle: sabemos que si $n = 0$, entonces $n!$ debe ser 1. La función *factorial()* está planteada de forma que ese caso ya está previsto: si n es cero, el *ciclo for* debe iterar en el $range(1, n+1) = range(1, 0+1) = range(1, 1)$ que es un *range* vacío al no incluir el límite derecho: incluye los elementos del intervalo $[1, 0]$, que no puede iterarse con un paso de avance igual a 1 (como en el *for* del script). Por lo tanto, en ese caso el *ciclo for* no hará ninguna repetición, y el proceso pasará de largo a la función *print*, mostrando el valor 1 con el que estaba inicializada f . Y este sería el resultado correcto ya que supusimos que n valía 0.

2.] Búsqueda del mayor (o el menor) de una secuencia cargada por teclado.

Habiendo llegado a un punto en desarrollo de temas en el que se han introducido herramientas diversas del lenguaje Python como las *estructuras condicionales y cíclicas* y el uso elemental de ciertas *estructuras de datos como las secuencias (hasta aquí, tuplas, cadenas y rangos)* podemos entonces abordar con profundidad una serie de temas, problemas, algoritmos y técnicas de la programación básica que más temprano que tarde deben ser incorporados al conjunto de conocimientos y habilidades de todo buen programador.

Parte de esta Ficha está orientada a ese objetivo. En general los temas nuevos que podrían aparecer desde este punto y hasta el final de la Ficha, lo harán en el contexto del análisis de los problemas y casos de estudio que hemos seleccionado aquí. Tómese su tiempo para comprender, aplicar y eventualmente modificar las ideas que serán expuestas, o pensar en variantes que pudieran servirle para la resolución de problemas similares. Siempre recuerde: **a programar, se aprende programando.**



El primero de los problemas que veremos constituye todo un caso de análisis clásico de la introducción a la programación: en la práctica es común enfrentar situaciones en las cuales se tiene un conjunto grande de valores que serán introducidos uno a uno en el programa (posiblemente a razón de uno por cada vuelta de un ciclo) y se *desea saber cuál es el mayor (o el menor) valor del conjunto*, o alguna característica asociada a ese valor.

Existen varias formas de hacer esto, pero la estrategia básica es la misma [2]: se trata de tomar al primer valor de la sucesión y guardarlo en una variable *may*. Luego tomar el que sigue, y compararlo contra el guardado antes en *may*. Si el nuevo valor es mayor, reemplazar el valor de *may* por el nuevo, y en caso contrario, mantener el valor de *may*. Todo el proceso es controlado por un ciclo que se detiene cuando se terminan los números de entrada. Se plantean a continuación tres variantes de resolución del problema (de acuerdo a distintas situaciones que podrían presentarse) cuyo enunciado formal sería el que sigue:

Problema 19.) *Determinar el mayor valor de una sucesión de valores que se cargan por teclado. Asuma en primera instancia que la cantidad n de números a procesar se conoce de antemano. Y luego de resolverlo con esa suposición, replantee su solución asumiendo que la cantidad n de números se desconoce y que la carga de datos terminará cuando se ingrese un 0.*

Discusión y solución: Tenemos distintas variantes interesantes para el análisis:

Primera variante: Como aquí se supone que se conoce de antemano la cantidad n de valores que serán cargados, puede usarse un ciclo *for* para la carga y procesamiento de los mismos. Según se indicó al exponer la estrategia básica para resolver el problema, la idea era comenzar tomando el *primer valor* de la sucesión y guardarlo en la variable *may*.

En esta primera variante, para determinar si el valor que se está procesando en una vuelta del ciclo es o no el primero, se usa un truco simple, pero directo: se pregunta si el valor de la variable i (que es la que controla al ciclo) es igual a 1 (*uno*). Si lo es, entonces el ciclo está en su primera repetición, y por lo tanto el valor leído en *num* (que es la variable que se usará para la carga de los valores) debe ser el primero, por lo cual se lo asigna en *may*. Si la variable i no es 1, entonces el dato cargado no es el primero, y por lo tanto se procede a compararlo con *may* para determinar si es el nuevo mayor o no. Mostramos el *pseudocódigo general* para esta solución propuesta:

```
# El valor  $n$  es la cantidad de datos a procesar.
Variante 1:
1. Repetir para  $i$  que va entre 1 y  $n$ :
    1.1 Cargar num (el número a procesar en esta vuelta)
    1.2 Si  $i == 1$  (¿es la primera vuelta del ciclo?):
        1.2.1 Sea may = num (iniciar may con ese número)
    sino:
        1.2.2 Si num > may (¿debo reemplazar may?):
            1.2.2.1 Sea may = num
2. Mostrar may
```

El programa completo en Python sigue a continuación:

```
print('Determinacion del mayor de una sucesion (variante 1)...')
n = int(input('Cantidad de numeros a procesar: '))

for i in range(1, n+1):
    num = int(input('Numero: '))
    if i == 1:
```



```
    may = num
else:
    if num > may:
        may = num

print('El mayor es:', may)
```

Segunda variante: Si se observa bien la solución planteada en la primera variante, puede concluirse que la pregunta por el valor de i en cada vuelta del ciclo en realidad provoca una pérdida de tiempo: sabemos exactamente cuándo esa condición será cierta y cuándo falsa, y sabemos que sólo resulta realmente útil en la primera vuelta del ciclo.

¿Cómo evitar esa condición? Otra vez, la solución es un truco simple y directo: el primer valor de la secuencia se puede cargar *antes* de comenzar el ciclo, y asignarse directamente en la variable *may*, con lo cual la misma queda ya inicializada. El ciclo se reserva para procesar al resto de los números de la secuencia. Como el primero se procesó antes del ciclo, entonces el ciclo debe procesar todavía $n - 1$ valores (es decir, debe dar una vuelta menos que en la primera variante). Se puede ajustar el ciclo para ir desde 1 hasta $n - 1$, o bien para ir desde 2 hasta n , pues en ambos casos el ciclo realizará $n - 1$ vueltas. Dado que ahora el primer dato se carga y se procesa *antes del ciclo*, entonces ya no es necesario que en el bloque del ciclo se pregunte en cada vuelta si el dato cargado es el primero: el ciclo está procesando los datos desde el **segundo** en adelante.

El pseudocódigo general del algoritmo adaptado a la segunda variante, se muestra a continuación:

```
# El valor n es la cantidad de datos a procesar.
Variante 2:
1. Cargar num (con el primer número de la secuencia)
2. Sea may = num (iniciar may con ese primer número)
3. Repetir para i que va entre 2 y n (una vuelta menos...):
    3.1 Cargar num (el siguiente número a procesar en esta vuelta)
    3.2 Si num > may (¿debo reemplazar may?):
        3.2.1 Sea may = num
4. Mostrar may
```

Mostramos ahora el programa completo. Note que sólo ha cambiado el planteo del algoritmo de búsqueda del mayor, para eliminar la pregunta por la primera vuelta dentro del ciclo y ajustarla a la segunda variante:

```
print('Determinacion del mayor de una sucesion (variante 2)...')
n = int(input('Cantidad de numeros a procesar: '))

num = int(input('Ingrese el primer numero: '))
may = num

for i in range(2, n+1):
    num = int(input('Siguiente numero: '))
    if num > may:
        may = num

print('El mayor es:', may)
```

Tercera variante: En las dos variantes anteriores se usó un ciclo *for* porque la primera parte del enunciado indicaba que se conocía de antemano la cantidad n de valores a procesar. Pero el mismo enunciado, en su segunda parte, sugiere que se plantee una solución suponiendo ahora que se desconoce la cantidad de datos n a cargar. En la primera variante



el ciclo *for* ayudó mucho porque se usó la variable *i* de control del mismo para determinar si se estaba ejecutando o no la primera vuelta.

¿Qué pasaría si se desconociera la cantidad *n* de valores a procesar? Supongamos que el enunciado fuese el siguiente:

Determinar el mayor de una secuencia de valores que ingresan de a uno. Se desconoce cuantos valores serán procesados, pero se indicará el final de la secuencia cargando el número cero.

En este caso, como sabemos, la solución es usar un ciclo *while* que controle si el número cargado en cada vuelta es distinto de cero, y aplicar un esquema de carga por doble lectura (como se explicó en la *Ficha 6*). Si efectivamente el número cargado es distinto de cero, el ciclo continúa y se aplica básicamente la misma estrategia ya vista: se almacena el primer número la variable *may*, y a los restantes se los compara contra el valor de *may* para saber si se debe actualizar o no el valor de *may*.

Puede aplicarse sin mayores problemas la *segunda variante* ya vista. Sin embargo, ¿cómo aplicaríamos la *primera variante* si tuviésemos que hacerlo? ¿Cómo podríamos preguntar por el *primer valor* sin contar con la variable *i* de control del ciclo?

Si se analiza el papel que juega la variable *i* en la *primera variante*, puede deducirse que la idea central es asignar a dicha variable un *valor inicial conocido*, y luego en cada vuelta del ciclo, *preguntar por ese valor inicial*. En nuestro caso, el valor inicial de *i* es *1 (uno)*. En cada vuelta, se pregunta si *i* sigue valiendo *1 (uno)*. Si se responde que sí, se tiene el primer valor, por estar el ciclo en la primera vuelta. Pero el hecho importante es que en las siguientes vueltas del ciclo, el valor de *i* no es *1*, ya que el propio ciclo *for* cambia su valor en cada repetición. Por lo tanto, la pregunta por el valor inicial de *i* sólo será verdadera en la primera vuelta.

Para emular esta situación cuando no hay un ciclo *for* ni un contador disponible, puede usarse el concepto de *variable centinela* o *bandera* que se introdujo también en la *Ficha 6*. Los principios son los mismos: se usa una variable cualquiera *b*, a la cual se le asigna antes de comenzar el ciclo un valor inicial conocido arbitrario (por ejemplo, el valor *true*). En cada vuelta del ciclo, se pregunta por ese valor, pero de tal forma que si la respuesta es verdadera se asigna el número leído en *may* y se *cambia el valor de la bandera* para provocar un *false* en la vueltas posteriores.

Así de directo. La única diferencia con la *primera variante* es que ahora el cambio de valor en la variable centinela no es efectuado por la cabecera del ciclo, sino por una instrucción de asignación ubicada en la rama verdadera de la condición. Y como sabemos, es muy común que la variable centinela sea definida de tipo *boolean*, pues sólo se necesitan para ella dos valores. El programa Python se muestra a continuación:

```
print('Determinacion del mayor de una sucesion (variante 3)...')

may = None
b = False

num = int(input('Ingrese un numero (con 0 finaliza): '))
while num != 0:
    if b == False:
        may = num
```



```
b = True
else:
    if num > may:
        may = num

num = int(input('Ingrese otro (con 0 finaliza): '))

print('El mayor es:', may)
```

En el proceso se comienza definiendo la variable *may* con el valor *None*. Esto se hace por una razón preventiva: si al pedir la carga del primer número (antes del ciclo) el usuario llegase a ingresar directamente un 0, el ciclo no ejecutaría su bloque de acciones y la variable *may* no llegaría a definirse, por lo que la instrucción *print(may)* lanzaría un error [1]. Al iniciar *may* en *None* antes del ciclo, la variable queda definida aunque con un valor "ficticio", y ese será el valor que se mostrará en caso que el usuario no haya cargado dato alguno.

3.] Procesamiento de datos divididos en categorías distintas.

El siguiente problema contiene varias de las técnicas y elementos vistos hasta aquí, incluyendo la forma de determinar el mayor de una secuencia de valores cuando se desconoce la cantidad de datos a procesar y la *secuencia de valores de entrada se divide en categorías distintas* [2]. El enunciado es el que se muestra aquí:

Problema 20.) *Un pequeño comercio de papelería cuenta con dos vendedores. Cada vendedor está codificado con los números 1 y 2. Considere que la carga de datos se realizará desde teclado, de forma que una entrada consta de 3 variables que representan una venta realizada: por cada venta, cargar el código del vendedor (1 o 2) que hizo la venta, cantidad de artículos vendida en esa operación, e importe de la venta. El fin de datos se indicará con código de vendedor igual a 0 (cero). El dueño del comercio desea cierta información estadística y para ello solicita un programa que obtenga lo siguiente:*

- a.) *La cantidad de productos vendida por cada vendedor (dos totales).*
- b.) *El importe total vendido por cada vendedor (otros dos totales).*
- c.) *El importe de la menor venta realizada por el vendedor 2.*
- d.) *El importe promedio de ventas por vendedor (importe total acumulado / 2).*

Discusión y solución: Este problema muestra un típico caso en el que los datos de entrada se presentan de alguna forma divididos en *categorías diferentes*: en este caso, se tienen ciertos datos que pertenecen a las ventas realizadas por el *vendedor 1*, y ciertos otros datos que describen ventas del *vendedor 2*. Para cada vendedor se pide calcular resultados diferenciados, y eso lleva a tener que dividir el proceso a realizar en dos ramas (una por cada vendedor). El programa completo es algo extenso, pero no demasiado complejo en sus detalles.

La carga de datos, como es costumbre, se realiza con un ciclo, que en este caso será un *while* implementando un *esquema de carga por doble lectura*, ya que se desconoce la cantidad de ventas a procesar. Por cada venta, se tienen tres datos: el código del vendedor que la hizo (variable *codigo*), la cantidad de productos vendidos (variable *cantidad*), y el importe de esa venta (variable *importe*). El ciclo debe permitir que en cada vuelta se carguen esos tres



valores, y sabemos que debe detenerse si aparece un 0 en el código de vendedor. El programa completo podría quedar así:

```
# pasó la primera venta del vendedor 2?
aviso = False

# si no se cargan ventas del vendedor 2, menor_importe queda en None...
menor_importe = None

# acumuladores de cantidades...
c1 = c2 = 0

# acumuladores de importes...
i1 = i2 = 0

print('Ventas de un Comercio... ingrese los datos de cada venta...')

# ingresar (y validar) el primer codigo...
codigo = -1
while codigo < 0 or codigo > 2:
    codigo = int(input('Codigo de vendedor (1 o 2) (0 para cortar): '))
    if codigo > 2 or codigo < 0:
        print('Error... se pidio 1 o 2 o 0 para cortar...')

while codigo != 0:
    cantidad = int(input('Cantidad vendida: '))
    importe = float(input('Importe: '))

    if codigo == 1:
        c1 += cantidad
        i1 += importe

    elif codigo == 2:
        c2 += cantidad
        i2 += importe

    # Aplicar mecanismo de cálculo del menor...
    if not aviso:
        aviso = True
        menor_importe = importe

    elif importe < menor_importe:
        menor_importe = importe

# ingresar el siguiente codigo y volver al ciclo...
codigo = -1
while codigo < 0 or codigo > 2:
    codigo = int(input('Codigo de vendedor (1 o 2) (0 para cortar): '))
    if codigo > 2 or codigo < 0:
        print('Error... se pidio 1 o 2 o 0 para cortar...')

# Calcular el importe promedio...
promedio = (i1 + i2) / 2

print('Cantidad de productos vendida por el vendedor 1:', c1)
print('Cantidad de productos vendida por el vendedor 2:', c2)
print('Importe total facturado por el vendedor 1:', i1)
print('Importe total facturado por el vendedor 2:', i2)
print('Importe de la menor venta del vendedor 2:', menor_importe)
print('Importe promedio entre los dos vendedores:', promedio)
```



El programa lanza un proceso de carga por doble lectura para el *código del vendedor*, que sólo se detendrá cuando ese *código de vendedor* sea cero. Dentro del ciclo y al inicio del bloque de acciones, se cargan los demás datos de esa venta (*cantidad* e *importe*). Note que estas dos variables se cargan dentro del ciclo, *una vez que nos hemos asegurado que el código del vendedor no es cero*: no tendría sentido cargar los tres datos juntos (antes del ciclo o al final de su bloque) si el código del vendedor a cargar en ese momento fuese cero... por no decir que en ese caso la carga inútil de la *cantidad* y el *importe* molestaría mucho al usuario...

Note que para la carga del *código de vendedor* se está aplicando un *proceso de validación* (que se explica en detalle en una ficha posterior): básicamente, se pide el código del vendedor por teclado y se acepta ese código si se cargó correctamente, pero se vuelve a pedir si el número ingresado no es un 0, un 1 o un 2: el cero es un valor válido para el *código de vendedor*, ya que ese es justamente el valor a ingresar para detener el ciclo de carga en el programa principal.

También dentro del ciclo de carga se usa una *instrucción condicional anidada* para saber cuál de los vendedores hizo la venta, y en función de ello se acumulan las *cantidades* y los *importes* en *distintas variables de acumulación* (*c1* y *c2* para las cantidades, además de *i1* e *i2* para los importes). Esos cuatro acumuladores se definen antes del ciclo, inicializados en cero.

Si el *código de vendedor* fue el 2 (segunda rama de las instrucciones condicionales anidadas) se aplica el ya conocido *proceso de determinación del mayor/menor* que hemos analizado en esta misma Ficha, para ir quedándose con el menor de los *importes* que bajen por esa rama. Para capturar al *primero* de esos importes, se usa una *variable booleana a modo de bandera* llamada *aviso*, que empieza con el valor *False* y quedará con ese valor *hasta que se detecte la primera venta que baje por la rama 2* (y en ese momento cambia a *True*).

Al terminar el ciclo, cuando los importes individuales ya se han cargado y acumulado completamente, se calcula el *importe promedio por vendedor*. Un análisis más detallado (que recomendamos...) se deja para el estudiante.

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.