Ficha 5

Estructuras Condicionales: Variantes

1.] Variantes de la instrucción condicional.

Hemos visto que la forma general típica de una instrucción condicional, consistente en una expresión lógica y dos ramas o salidas: la rama verdadera y la rama falsa. Planteada de esta forma, la instrucción condicional suele ser designada como una *condición doble*. En muchos casos esta forma típica es adecuada y suficiente para el planteo del programa que está desarrollando.

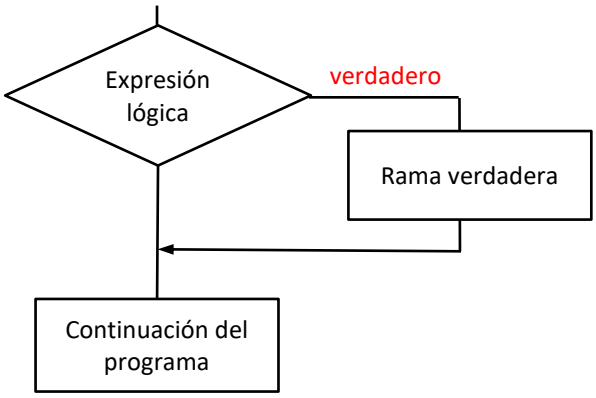
Sin embargo, puede ocurrir (y de hecho es muy común) que para una condición sólo se especifique la realización de una acción si la respuesta es verdadera y no se requiera hacer nada en caso de responder por falso. Para estos casos, en Python y otros lenguajes es perfectamente válido escribir una instrucción condicional que *sólo tenga la rama verdadera*, *omitiendo por completo la falsa*. Una instrucción condicional de ese tipo se suele designar como *condición simple*, y en ella no se especifica la rama *else*: la instrucción condicional termina cuando termina la rama verdadera. La forma general típica de una *instrucción condicional simple* en Python es la siguiente:

```
if expresión lógica:
      instrucciones de la rama verdadera

continuación del programa
```

Las instrucciones de la rama verdadera se encolumnan en un bloque hacia la derecha, del mismo modo que en una condición doble, pero al terminar este bloque se escriben directamente las instrucciones para continuar con el programa, en la misma columna del *if* inicial, sin escribir la rama *else*. El diagrama de flujo de la *Figura 1* aclara la forma de funcionamiento.

Figura 1: Diagrama general de una instrucción condicional *simple* típica.



Como se ve en la figura, si la expresión lógica es verdadera se ejecutará la rama verdadera y luego continuará el programa, igual que en las condiciones dobles. Pero si la expresión es

falsa, no se ejecutará ninguna rama especial y también se continuará con el programa. Vale decir: el bloque etiquetado como "*Continuación del programa*" en la figura, ***no es*** la rama o salida falsa de la condición, ya que ese mismo bloque se ejecutará también al responder por verdadero a la expresión lógica. De hecho, ese bloque está *fuera* de la instrucción condicional.

A modo de ejemplo, en el siguiente script el valor de la variable *n* empieza siendo cero. Si el valor de otra variable *x* que se carga por teclado es mayor a cero, se cambia el valor de *n* asignándole el cociente entre *a* y *x*, pero de otro modo no se hace nada y el valor final de la variable *n* queda en 0:

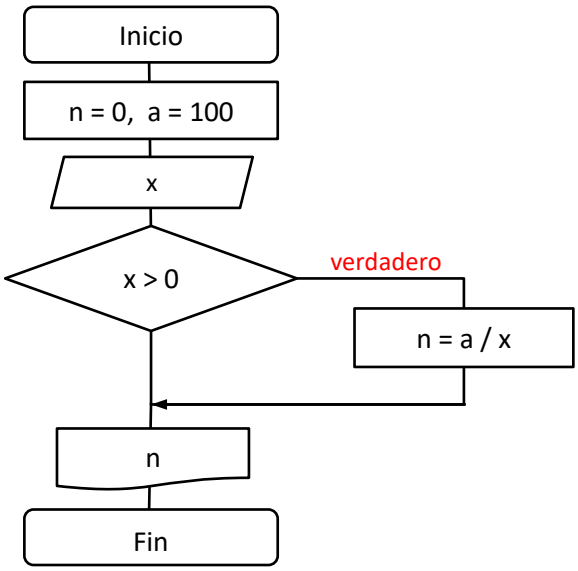
```
n = 0
a = 100
x = int(input('x: '))

if x > 0:
      n =  a / x

print('Valor final:', n)
```

El diagrama de flujo del script anterior, se ve en la figura siguiente:

Figura 2: Diagrama de flujo del script del cálculo condicional del cociente.



Por otra parte, y continuando con el estudio de variantes para la instrucción condicional, es posible que en la rama falsa y/o en la rama verdadera de una instrucción condicional se requiera plantear otra instrucción condicional. De hecho, es posible que a su vez cada nueva instrucción condicional incluya otras y así sucesivamente según lo vaya necesitando el programador, sin límites teóricos. Cuando esto ocurre, se tiene lo que se conoce como un *anidamiento de condiciones*. A modo de ejemplo, analicemos el siguiente problema:

Problema 11.) *Cargar por teclado tres números enteros y determinar y mostrar el mayor de ellos. No utilice para el proceso la función max() de la librería estándar de Python: diseñe el algoritmo suponiendo que tal función no existe en el lenguaje que usará para el desarrollo del programa.*

a.) Identificación de componentes:

- **Resultados:** El mayor entre tres números. (*may*: int)
- **Datos:** Tres números enteros. (*n1*, *n2*, *n3*: int)
- **Procesos:** Problema de *búsqueda del mayor de un conjunto*. El desafío básico es tratar de plantear un esquema de condiciones que lleve al mayor del conjunto, pero de forma que ese esquema incluya *la menor cantidad posible de condiciones*. De esta forma, el programa será más breve, más simple de comprender, y posiblemente más eficiente en cuanto a tiempo de ejecución.

  Existen muchas estrategias que podrían aplicarse, pero una muy intuitiva es la siguiente: comenzar con una de las tres variables (sea *n1*) y tratar de determinar si esa variable contiene al mayor, sin preocuparse por las otras dos. Evidentemente, el siguiente esquema de pseudocódigo condicional cumple con esa consigna, y deja el valor del mayor en la variable *may* si ese mayor estaba en *n1*:

```
si n1 > n2  y  n1 > n3:
    may = n1
sino:
    # el mayor no es n1!!!
```

  La condición planteada es simple y directa: se pregunta si *n1* es mayor que *n2* y si al mismo tiempo *n1* es mayor que *n3*, usando un conector *y* (conjunción lógica) (o sea, un *and* en Python). Si la respuesta es cierta, evidentemente el mayor está en *n1* y en la rama verdadera de esa condición se asigna *n1* en *may*.

  El problema surge si la condición fuese falsa, en cuyo caso no podemos afirmar en forma inmediata cuál es el mayor. Sin embargo, aun cuando en este caso el problema no está resuelto, tenemos una pequeña ganancia: si la condición fue falsa, no sabemos cuál es el mayor, pero sabemos de manera categórica que ese mayor ***no es*** *n1*: si la construcción *and* fue falsa, significa que o bien *n1* es menor que *n2*, o bien es menor que *n3*, o bien es menor que ambas… y en los tres casos implica que *n1* no es el mayor.

  Por lo tanto, si la condición entró en la rama falsa sólo nos queda comparar *n2* con *n3* para saber cuál de las dos contiene al mayor, lo cual puede hacerse con otra condición, *anidada en esa rama falsa*. La estructura de pseudocódigo podría verse así:

```
si n1 > n2  y  n1 > n3:
    may = n1
sino:
    si n2 > n3:
        may = n2
    sino:
        may = n3
```

  Note la forma en que se escribió el pseudocódigo del *anidamiento de condiciones*: la rama falsa de la primera condición incluye a la segunda condición, escrita (o indentada) en una nueva columna hacia la derecha. Toda la segunda condición está escrita a partir de esa columna, indicando que toda ella pertenece al bloque de la rama falsa de la primera condición. Y a su vez, las ramas verdadera y falsa de la segunda condición, se indentaron en una nueva columna hacia la derecha.

b.) Planteo del algoritmo: Mostramos tanto el *pseudocódigo* como el *diagrama de flujo* a continuación:

Figura 3: Pseudocódigo y diagrama de flujo del problema de cálculo del mayor.

Algoritmo:

1.) Cargar n1, n2 y n3: tres números enteros.

2.) Si n1 > n2 y n1 > n3:

2.1.) may = n1

3.) sino:

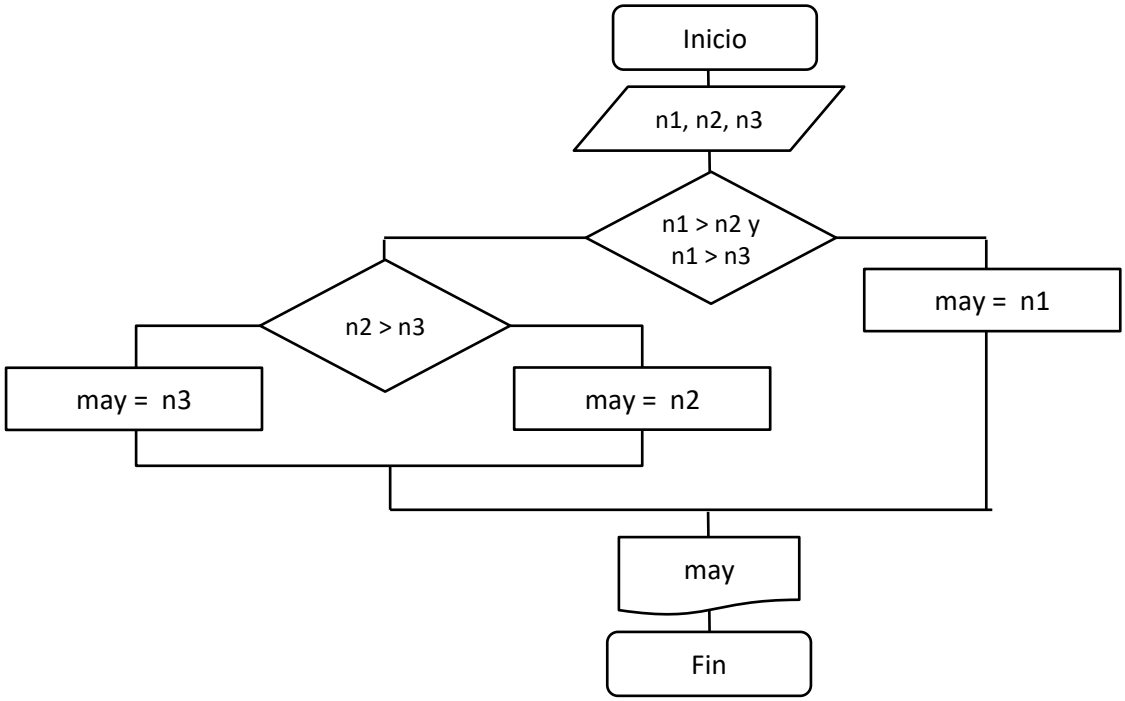
3.1.) si n2 > n3:

3.1.1.) may = n2

3.2.) sino:

3.2.1.) may = n3

4.) Mostrar may: el mayor de los tres números.



c.) Desarrollo del programa: En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```
__author__ = 'Cátedra de AED'


# Título general y carga de datos...
print('Problema del cálculo del mayor entre tres números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))
n3 = int(input('N3: '))

# Procesos...
if n1 > n2 and n1 > n3:
    may = n1
else:
```

```
    if n2 > n3:
        may = n2
    else:
        may = n3

# Visualización de resultados..
print('El mayor es:', may)
```

La instrucción condicional anidada que aparece en este script sigue fielmente la estructura del pseudocódigo que mostramos más arriba. La *segunda instrucción condicional está incluida en el bloque de la rama falsa de la primera*, y esto se indica en Python por medio de la correcta indentación de ese bloque.

Señalemos finalmente que en Python se puede plantear una variante para la instrucción condicional, *que permite evitar el anidamiento excesivo de condiciones*. Se trata de la variante *if – elif*, que puede usarse en forma combinada con el *if – else* normal [1] [2]. La parte *elif* de una condición, cuando está presente, equivale a un *else* que contenga a su vez a otra condición, sin tener que trabajar tanto en la indentación de la estructura anidada y simplificando el código fuente. A modo de ejemplo, una estructura anidada tal como:

```
if opcion == 1:
      print('Se eligió la opción 1')
else:
      if opcion == 2:
            print('Se eligió la opción 2')
      else:
            if opcion == 3:
                  print('Se eligió la opción 3')
            else:
                  print('Opción no válida')
```

podría reescribirse de la siguiente forma, más compacta, usando *elif*:

```
if opcion == 1:
      print('Se eligió la opción 1')
elif opcion == 2:
      print('Se eligió la opción ')
elif opcion == 3:
      print('Se eligió la opción 3')
else:
      print('Opción no válida')
```

Cada línea que contiene un *elif* continúa en forma directa con la *expresión lógica* que se quería evaluar en la salida falsa de la condición anterior, sin tener que volver a escribir la palabra *if* en esa línea. Observe que en este esquema, la última condición llevó un *else* y no un *elif*, ya que esa rama no incluye (en el esquema original anidado) otra condición, sino directamente un *print()*. Note también que al usar *elif* se simplificó de manera notable el esquema de indentación del script.

Un detalle interesante, es que Python *no dispone* de *instrucciones condicionales múltiples* especiales como las instrucciones *switch* o *case* de otros lenguajes, pero como se vio en el ejemplo anterior, el uso de *elif* permite plantear una estructura de condiciones anidadas que equivale por completo a una condición múltiple.

Por otra parte, observe que tanto el diagrama de flujo como el pseudocódigo no tienen por qué modificarse si el programador piensa hacer uso de *if – elif* en lugar de *if – else* anidados. La lógica es la misma, y en este caso Python provee una alternativa sintáctica más compacta para escribir el mismo algoritmo. El script para el cálculo del mayor que antes mostramos con condiciones *if – else* anidadas, puede replantearse así, mediante instrucciones *if – elif*, sin tocar en absoluto el diagrama de flujo ni el pseudocódigo de la *Figura 3*:

```
__author__ = 'Cátedra de AED'


# Título general y carga de datos...
print('Problema del cálculo del mayor entre tres números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))
n3 = int(input('N3: '))

# Procesos...
if n1 > n2 and n1 > n3:
    may = n1
elif n2 > n3:
    may = n2
else:
    may = n3

# Visualización de resultados..
print('El mayor es:', may)
```

Para cerrar esta sección, mostramos ahora un nuevo problema de aplicación de los temas vistos hasta aquí:

Problema 12.) *Una compañía de alquiler de automóviles necesita un programa que calcule lo que se debe cobrar a cada cliente, teniendo en cuenta el kilometraje recorrido por el cliente al devolver el automóvil* [1]*:*

i. *Si el cliente no superó los 300 km recorridos se deberá cobrar $500.*

ii. *Para recorridos desde más de 300 km y hasta no más de 1000 km se le cobrará $500 más el kilometraje excedente a los 300, a razón de $3 por kilómetro.*

iii. *Para recorridos mayores a 1000 km se le cobrará $500 más el kilometraje excedente a los 300, a razón de $1.5 por kilómetro.*

a.) Identificación de componentes:

- **Resultados:** El monto a cobrar. (*monto*: float)
- **Datos:** El kilometraje recorrido. (*kilometraje*: int)
- **Procesos:** Problema de *aritmética básica aplicado a un contexto comercial*. La base del algoritmo es determinar en qué rango está el valor de la variable *kilometraje*, lo cual puede hacerse con condiciones anidadas y prestando atención a los límites de esos rangos. Por otra parte, sabemos que si kilometraje es mayor que 300, debemos saber cuántos kilómetros de excedente hubo, para calcular con

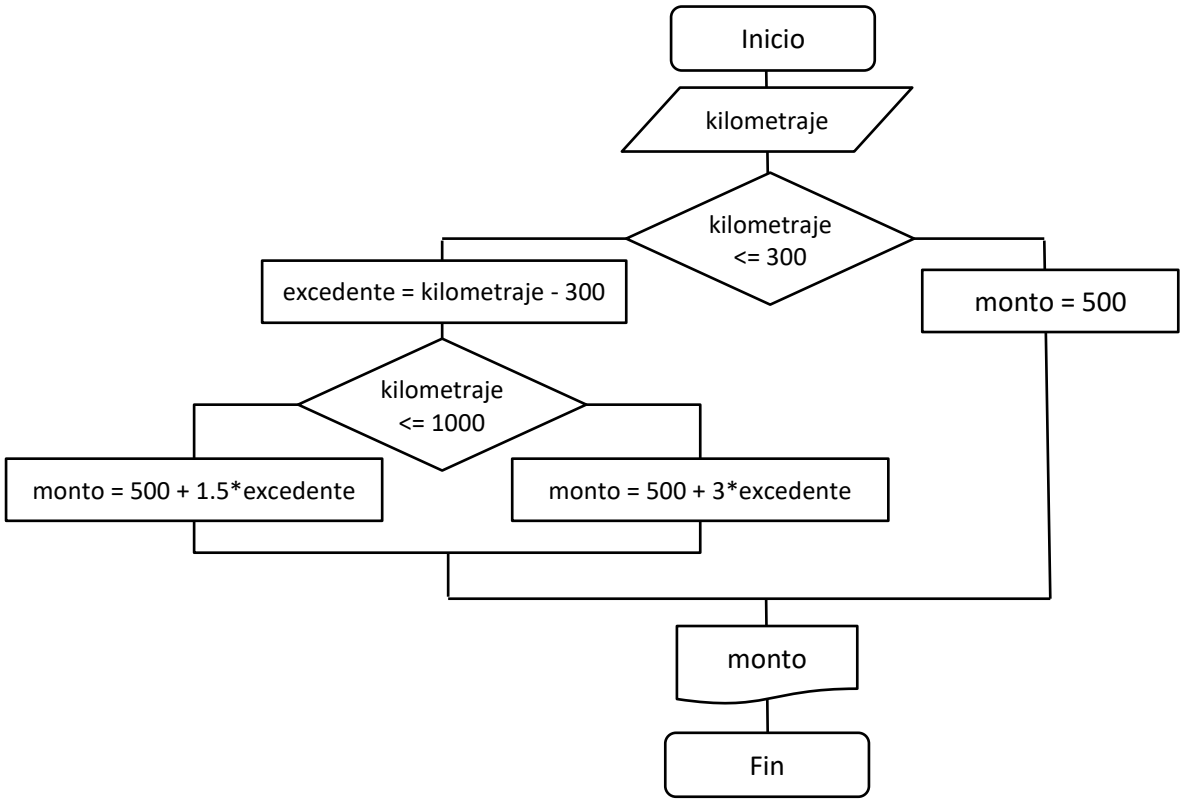
[1] Este problema fue sugerido por la *Ing. Marina Cardenas*, docente de esta Cátedra.

ello el monto final a pagar. Todo eso puede resumirse en el siguiente esquema de pseudocódigo general:

```
si kilometraje <= 300:
    monto = 500
sino:
    excedente = kilometraje - 300
    si kilometraje <= 1000:
        monto = 500 + 3*excedente
    sino:
        monto = 500 + 1.5*excedente
```

La idea es simple: si *kilometraje* es mayor a 300, se calcula por única vez el valor del *excedente*, y luego se determina si el kilometraje superó o no los 1000 kilómetros. Si lo hizo, se calcula el monto final sumando 1.5**excedente* al valor base de 500. Y si no superó los 1000 kilómetros, se suma 3**excedente* al monto base de 500.

b.) Planteo del algoritmo: Mostramos ahora el *diagrama de flujo* de la solución:



c.) Desarrollo del programa: Como siempre, en base al diagrama el script se deduce en forma inmediata:

```
__author__ = 'Cátedra de AED'

# Título general y carga de datos...
kilometraje = int(input("Ingrese la cantidad de kilómetros: "))

# Proceso: cálculo del valor a cobrar
if kilometraje <= 300:
    monto = 500
```

```
else:
    excedente = kilometraje - 300
    if kilometraje <= 1000:
        monto = 500 + 3*excedente
    else:
        monto = 500 + 1.5*excedente

# Visualización de resultados
print("El importe a pagar es:", monto)
```

2.] Expresiones de conteo y acumulación.

A medida que se avanza en el estudio y planteo de algoritmos y programas para problemas cada vez más complejos, se verá que en la mayoría de esos programas será necesario eventualmente llevar a cabo procesos de *conteo* (por ejemplo, determinar cuántas veces apareció un número negativo), o de *sumarización* (por ejemplo, determinar cuánto vale la suma de todos los valores que tomó la variable *x* a lo largo de la ejecución de programa, suponiendo que *x* cambia de valor durante esa ejecución). El primer caso se resuelve incorporando una *variable de conteo* (o simplemente un *contador*), y el segundo, incorporando una *variable de acumulación* (o simplemente un *acumulador*) [3].

En ambas situaciones se trata de variables que en una expresión de asignación aparecen en *ambos miembros*: la misma variable se usa para hacer un cálculo y para recibir la asignación del resultado de ese cálculo. Los siguientes son dos ejemplos de *expresiones de conteo* o *acumulación* (en el primero, la variable *a* se usa como un *contador*, y en el segundo la variable *b* se usa como un *acumulador*):

```
a = a + 1
b = b + x
```

En general, un *contador* es una variable que sirve para *contar* ciertos eventos que ocurren durante la ejecución de un programa. Intuitivamente, se trata de una variable a la cual se le suma el valor *1 (uno)* cada vez que se ejecuta la expresión. Esto es así porque *contar* normalmente significa *sumar 1*, pero en la práctica puede sumarse cualquier valor, o incluso no hacer una suma sino cualquier otra operación que involucre a cualquier *constante*.

Entonces, técnicamente, un *contador* es una variable que actualiza su valor en términos de *su propio valor anterior* y de *una constante*. A modo de ejemplo: la variable *a* del siguiente esquema actúa a modo de *contador* para determinar cuántos números son negativos entre tres que se cargan por teclado:

Figura 4: Uso básico de un contador.

```
__author__ = 'Catedra de AED'


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

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

num = int(input('Ingrese otro: '))
```

```
if num < 0:
    a = a + 1

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

Cada vez que alguno de los números cargados en la variable *num* sea negativo, se ejecuta la instrucción indicada en color azul: *a = a + 1* la cual funciona así:

- ✓ **En primer lugar** se ejecuta la parte *derecha* de la asignación, con el valor actual de ***a***. Si ***a*** comenzó valiendo cero, entonces la primera vez que se ejecute la expresión ***a + 1*** se obtiene un uno.
- ✓ **En segundo lugar** se asigna el valor así obtenido en la misma variable ***a***, con lo cual se cambia el valor original. En este caso, la primera vez que se cargue un negativo en *num* la variable ***a*** quedará valiendo **1**. Si se carga un segundo negativo en *num*, ***a*** quedará valiendo **2**, y así cada vez que se ingrese un negativo, asignando a la variable ***a*** su valor anterior sumado en uno.

En nuestro ejemplo, el valor final de la variable *a* indicará cuántos números negativos se ingresaron, mostrándolo con un mensaje. Observar que la variable que se usa como contador *debe* ser inicializada con un valor adecuado antes de comenzar a usarla, para eliminar cualquier valor residual y garantizar que el conteo comience desde el valor correcto (normalmente el *cero*, como en el ejemplo, aunque esto depende de lo pedido en el problema).

Del mismo modo que en la expresión *a = a + 1* la variable *a* funciona como *contador*, en forma similar la variable *c* en la expresión *c = c - 1* funciona como *decrementador*: va restando de a uno a partir del valor original de la variable *c*. Pero también notemos que cualquier *constante* y cualquier *operador* sirven para formar la expresión general. En los siguientes ejemplos mostramos *contadores* de formas diversas (asegúrese de entender lo que cada instrucción hace cada vez que se ejecuta):

```
a = a + 1
b = b - 1
c = c + 2
d = d * 4
e = e / 3
```

Por otra parte, un *acumulador* o *variable de acumulación* es básicamente una variable que permite *sumar* los valores que va asumiendo *otra variable* o bien *otra expresión* en un proceso cualquiera. Técnicamente, y en general, un *acumulador* es una variable que actualiza su valor en términos de su propio valor anterior y el valor de *otra variable* u *otra expresión*. Por ejemplo, el siguiente esquema permite ir *sumando* los valores que asume la variable *x*, usando una *variable de acumulación s*:

Figura 5: Uso básico de un acumulador.

```
__author__ = 'Catedra de AED'


s = 0
x = int(input('Ingrese un número: '))
s = s + x

x = int(input('Ingrese otro: '))
s = s + x

x = int(input('Ingrese otro: '))
s = s + x

print('La suma de los valores cargados es:', s)
```

En este esquema, se cargan por teclado distintos valores en la variable *x* a lo largo de tres instrucciones de carga. La instrucción *s = s + x* funciona en forma similar a como trabajaba un contador, pero ahora no se va sumando de a uno sino que se va *sumando el valor que en cada carga tenga la variable x*. Si dicha variable asume sucesivamente los valores *3*, *4* y *7* entonces la variable *s* quedará valiendo *14*. Observar que al usar un *acumulador* también es obligación del programador asegurar un valor inicial a la variable de acumulación antes de empezar a usarla, para que efectivamente quede definida o para eliminar cualquier valor residual si ya estaba definida y se usó en otro lugar del mismo programa. En el ejemplo anterior, la variable *s* se inicializó en cero, y luego comenzó el proceso de carga de los valores de la variable *x*.

Según la definición dada de un acumulador, las siguientes expresiones también son expresiones de acumulación:

```
s = s + x
b = b * z
a = a - y
p = p / t
c = c + 2*x
```

Es interesante notar que en Python (como en otros lenguajes) cualquier *expresión de conteo* o *de acumulación* responde a la *forma general* siguiente [1]:

```
variable = variable  operador  expresión
```

donde ***variable*** es la variable cuyo valor se actualiza (y aparece en ambos miembros de la expresión de asignación) y ***expresión*** es una constante, una variable o una expresión propiamente dicha (formada a su vez por constantes, variables y operadores). Y el hecho es que en el lenguaje Python cualquier expresión que venga escrita en la *forma general* anterior, se puede escribir también en la ***forma resumida*** siguiente:

```
variable  operador= expresión
```

A modo de ejemplo, veamos las siguientes equivalencias:

Forma general	Forma resumida
a = a + 1	a += 1
b = b – 1	b –= 1
c = c + 2	c += 2
d = d * 3	d *= 3
e = e / 4	e /= 4
s = s + x	s += x
b = b * z	b *= z
a = a – x	a –= x
p = p / t	p /= t
c = c + 2*x	c += 2*x

3.] Variables centinela (o *banderas*).

En muchas ocasiones los programadores necesitan *marcar* de alguna forma *un suceso que hubiese ocurrido* a lo largo de la ejecución de un programa, para luego poder chequear en distintos puntos del programa y en forma simple si ese suceso realmente tuvo lugar o no, para ejecutar en consecuencia ciertas acciones.

Las situaciones prácticas que darían lugar a esta clase de necesidad son muy diversas y dependen del contexto de cada problema: por ejemplo, se podría querer saber si algunos de los números que se cargaron en un programa eran negativos para que al finalizar la carga se muestre un único mensaje de advertencia, o si todos los datos cargados en otro programa correspondían a mujeres para lanzar al final un único aviso de posibles beneficios por maternidad, o si en algún momento el valor de una variable cambió para que al terminar el proceso se avise al administrador del sistema de un posible error.

En estos casos los programadores pueden recurrir a diferentes técnicas para marcar el suceso. Una técnica muy común se designa como uso de *variables centinela*, también conocida como uso de *variables bandera*, o uso de *señales* o simplemente uso de *banderas* (o uso de *flags* por su traducción al inglés) [3].

Una *bandera* es una variable (típicamente *booleana*, aunque podría ser de cualquier otro tipo) cuyo valor se controla en forma metódica a lo largo de la ejecución de un programa, de forma que cada valor posible se asocia a la ocurrencia o no de un evento (por ejemplo, si el evento ocurrió la *bandera* se asigna en *True*, y mientras el evento no ocurra el *flag* de mantiene en *False*). En cualquier momento el programador puede chequear el estado de la *bandera* y saber si el evento marcado ocurrió o no.

La mejor forma de terminar de incorporar la idea del uso de una *bandera*, es analizando su aplicación efectiva en un programa, para lo cual proponemos el siguiente problema (que además servirá como ejemplo de aplicación del uso de *variables de conteo*):

Problema 13.) *Se cargan por teclado las notas obtenidas por un estudiante en tres parciales realizados durante el cursado de una materia universitaria. Además, se carga la nota final que ese estudiante obtuvo en el desarrollo de los trabajos prácticos en esa misma materia. Se sabe que al terminar el cursado de la materia, todo alumno puede quedar en uno de los siguientes estados académicos:*

a. *Libre: si no llegó a cumplir con las condiciones para ser Regular.*
b. *Regular: si aprobó al menos dos de los tres parciales con nota de 4 o más y además obtuvo nota de 4 o más en la nota final de trabajos prácticos.*
c. *Promocionado: si aprobó los tres parciales con nota de 7 o más pero con promedio entre ellos de **8(ocho)** o más, y además obtuvo nota de 8 o más en la nota final del práctico.*
d. *Aprobado: si aprobó los tres parciales con nota de 7 o más pero con promedio entre ellos de **9(nueve)** o más, y además obtuvo nota de 8 o más en la nota final del práctico.*

El programa debe determinar y mostrar por pantalla el estado en que finalmente quedó el estudiante.

Discusión y solución: Asumiremos que las notas de los tres parciales serán cargadas por teclado en tres variables *n1*, *n2* y *n3* y que la nota final del práctico se cargará a su vez en la variable *np*. El programa determinará la condición final del estudiante, y almacenará en la variable *condicion* una cadena de caracteres que podrá ser *'Libre'*, *'Regular'*, *'Promocionado'* o *'Aprobado'*.

En primera instancia, notemos que la diferencia entre el estado *Promocionado* y el estado *Aprobado* es el valor del promedio de las notas de los tres parciales (que llamaremos *pp*): en el primer caso se pide un promedio *pp* de 8 o más y en el segundo un promedio *pp* de 9 o más. Pero en ambos estados el resto de las exigencias es el mismo: aprobar los tres parciales con nota de 7 o más, y obtener nota de 8 o más en la nota final del práctico *np*.

Lo anterior implica que tanto para verificar si el estudiante está *Promocionado* como para saber si está *Aprobado* el programa deberá chequear si los tres parciales están aprobados y con notas de 7 o más en cada caso y también si la nota final de práctico es 8 o más. La condición para hacer eso tendría la forma que sigue:

```
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8:
```

lo cual incluye cuatro proposiciones encadenadas con operadores tipo *and*. Si esta misma condición debe plantearse en dos o más lugares diferentes del programa, y además hay que agregar alguna otra proposición (por ejemplo: el chequeo del promedio entre los tres parciales) entonces cada condición sería bastante extensa de escribir. Por caso, estas serían las dos condiciones a plantear para saber si el estudiantes está *Aprobado* o *Promocionado*:

```
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8 and pp >= 9:
    condicion = 'Aprobado'

if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8 and pp >= 8:
    condicion = 'Promocionado'
```

El esquema anterior ilustra una situación simple pero válida en la cual podría usarse una *bandera* para simplificar el control. Claramente, el programador necesita *dejar marcado* el evento consistente en que los tres parciales estén aprobados con nota de 7 o más, al mismo tiempo que la nota final del práctico sea de 8 o más.

Procederemos para eso de la siguiente manera: sea la variable booleana *notas_ok* que comenzaremos asignando con el valor *False* antes de hacer el chequeo de las notas. La idea es que el valor de esa variable indique lo que hasta ese momento se sabe respecto de los parciales y la nota del práctico: usamos el valor *False* para registrar que hasta ese momento, no hay nada que nos indique que esas notas cumplen con los requisitos pedidos. Si luego del chequeo condicional se descubre que efectivamente esas notas son correctas, en ese momento el valor de *notas_ok* se cambia a *True* y se deja marcado con eso que las notas han sido verificadas con éxito.

En cualquier lugar del programa en que el programador necesite saber si las notas de los parciales y la nota del práctico cumplían con los requisitos exigidos, se podrá preguntar por el valor final de *notas_ok*: el valor *True* indicará que todo estaba en orden (*notas_ok = True*) o no (*notas_ok = False*). El esquema podría quedar modificado así:

```
notas_ok = False
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8:
    notas_ok = True

if notas_ok and pp >= 9:
    condicion = 'Aprobado'

if notas_ok and pp >= 8:
    condicion = 'Promocionado'
```

Recordemos que si la variable *notas_ok* es booleana, entonces no es necesario usar explícitamente el operador == para verificar su valor. Las dos últimas condiciones del esquema anterior son completamente equivalentes a las que se muestran a continuación:

```
if notas_ok == True and pp >= 9:
    condicion = 'Aprobado'

if notas_ok == True and pp >= 8:
    condicion = 'Promocionado'
```

Queda claro que el programador podría haber planteado el esquema de alguna forma alternativa para evitar tener que chequear dos veces las notas de los parciales y el práctico (invitamos a los estudiantes a que piensen por sí mismos en qué forma se podría haber hecho…) pero la idea aquí era simplemente mostrar la manera de aplicar en forma efectiva una *variable bandera* para marcar un evento. El programa completo que resuelve el problema se muestra a continuación:

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


# Carga de datos...
n1 = int(input('Nota en el parcial 1 (cargue 0 si no hizo el parcial): '))
n2 = int(input('Nota en el parcial 2 (cargue 0 si no hizo el parcial): '))
n3 = int(input('Nota en el parcial 3 (cargue 0 si no hizo el parcial): '))
np = int(input('Nota final del practico: '))

# Procesos...
# ...cuantos parciales aprobó?
cpa = 0
if n1 >= 4:
    cpa += 1
if n2 >= 4:
    cpa += 1
if n3 >= 4:
    cpa += 1

# ...aprobó los tres con nota >= 7 y el practico con nota >= 8?
notas_ok = False
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8:
    notas_ok = True

# ...promedio entre los tres parciales? (con coma y sin redondear)...
pp = (n1 + n2 + n3) / 3

# ...empezamos asumiendo que el estudiante es Regular...
condicion = 'Regular'

# ...pero si no lo es, cambiamos su estado...
if cpa < 2 or np < 4:
    condicion = 'Libre'

elif notas_ok and pp >= 9:
    condicion = 'Aprobado'

elif notas_ok and pp >= 8:
    condicion = 'Promocionado'

# Visualización de resultados...
print('Condicion final del estudiante:', condicion)
```

Adicionalmente, puede verse que el programa usa una *variable de conteo* llamada *cpa* para llevar la cuenta de la cantidad de parciales que el estudiante aprobó, lo cual es requerido para determinar si ese estudiante alcanzó o no la condición de Regular. El valor inicial de ese contador se ajusta en 0, y luego simplemente se incrementa en 1 por cada parcial con nota mayor o igual a 4 que se detecte.

Anexo: Temas Avanzados

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

a.) Caso de análisis: un algoritmo para ordenar tres números contenidos en tres variables.

Nos proponemos analizar con cierta profundidad un problema que se presentará en reiteradas ocasiones a todo lo largo de este curso: el ordenamiento de un conjunto de valores. En estas primeras etapas de la asignatura el problema aparecerá simplificado: dado un conjunto de unos pocos números, mostrar esos números en forma ordenada de menor a mayor.

En este caso, nuestro objetivo *será ordenar un conjunto de sólo tres números, contenidos en tres variables*. Podemos hacer esto con las herramientas que hemos visto hasta ahora, en forma muy elemental. Pero a medida que el curso avance será cada vez mayor el volumen de datos a ordenar, y para esos momentos deberán estudiarse algoritmos algo más sofisticados y emplear otras herramientas disponibles en un lenguaje de programación.

Para formalizar el trabajo, enunciamos el problema a modo de ejercicio:

Problema 14.) *Se cargan por teclado tres números. Se pide mostrarlos en pantalla, ordenados de menor a mayor.*

Discusión y solución: Comenzaremos suponiendo que tenemos que ordenar un conjunto de sólo dos números, que están contenidos en sendas variables *a* y *b* que se cargarán por teclado. Este caso es trivial, y puede resolverse con sólo una condición (como ya vimos en la *Ficha 4*, *problema 8*). Se puede plantear un esquema que proceda a ordenar dos números almacenados originalmente en las variables *a* y *b* guardándolos ordenados en otras dos variables que llamaremos *may* y *men*. La idea final es que si queremos ver esos dos números ordenados, apliquemos este proceso y luego mostremos los valores de *men* y *may* (en ese orden). En pseudocódigo el proceso podría quedar así (sin numeración de líneas para abreviar):

```
ordenar_dos_numeros:
     si a > b:
          may = a
          men = b
     sino:
          may = b
          men = a
```

Si ahora queremos pasar a ordenar los *tres* números que nos dieron, almacenados en tres variables *a*, *b* y *c*, podemos mantener la idea que ya hemos usado: intentaremos reasignar esos tres números en otras tres variables llamadas *men*, *med* y *may*, de forma que la primera (*men*) siempre termine conteniendo al menor de los tres originales, *med* al valor mediano, y *may* al mayor.

Una primera idea que siempre suele surgir, consiste en usar tantas condiciones como combinaciones posibles existan entre los números originales. En nuestro caso, siendo tres las variables de entrada, las combinaciones posibles son 6 y eso nos llevaría a tener que plantear 6 condiciones (si usamos condiciones simples), como se ve en el esquema siguiente (en el que para simplificar, asumimos que las tres variables *a*, *b* y *c* tienen *valores diferentes*):

```
# Pseudocódigo para el ordenamiento de tres números: versión 1...
ordenar_tres_numeros:
    si a > b > c:
          may, med, men = a, b, c

    si a > c > b:
          may, med, men = a, c, b

    si b > a > c:
          may, med, men = b, a, c

    si b > c > a:
          may, med, men = b, c, a

    si c > a > b:
          may, med, men = c, a, b

    si c > b > a:
          may, med, men = c, b, a
```

Aun cuando la solución obtenida parece buena, un programador debería acostumbrarse a desconfiar de su "primer impulso" cuando busca un algoritmo, pues ese primer impulso suele terminar en una solución intuitivamente obvia, pero ineficiente o poco práctica en términos de demora en el tiempo de ejecución o en la cantidad de líneas de código que deberá escribir para implementarla. Estas soluciones en las que el programador explora todas las posibles variantes y combinaciones posibles de resultados, se suelen designar como estrategias de "*fuerza bruta*".

Veamos el caso: hemos encontrado una solución simple, pero nos ha llevado a escribir *seis* condiciones para plantearla. Podemos intuir que si seguimos este camino, nos irá cada vez peor a medida que el número de datos aumente. Por caso, si queremos ordenar 4 números dispuestos en 4 variables de entrada, y seguimos la misma idea, requeriremos 24 condiciones (que es igual al número de combinaciones posibles entre 4 valores...) y eso ya parece demasiado trabajo...

Nuestro objetivo es encontrar un algoritmo que haga el ordenamiento de tres números, pero con la *menor cantidad posible de instrucciones condicionales*. Además, queremos que este algoritmo nos abra la mente, para repetir la misma la idea básica cuando el número de datos suba a 4 (por ejemplo) y también queramos la mínima cantidad de condiciones.

Una forma de hacerlo consiste en aplicar una variante simplificada de un algoritmo más amplio llamado *ordenamiento por inserción*. La idea es comenzar con una *reducción del volumen de datos del problema*: en vez de considerar los tres números *a*, *b* y *c* originales, suponemos que nos han dado sólo dos (*a* y *b*) e intentamos ordenarlos llevándolos a las variables *men* y *may* (por ahora, nos olvidamos de *c* y de la variable *med*):

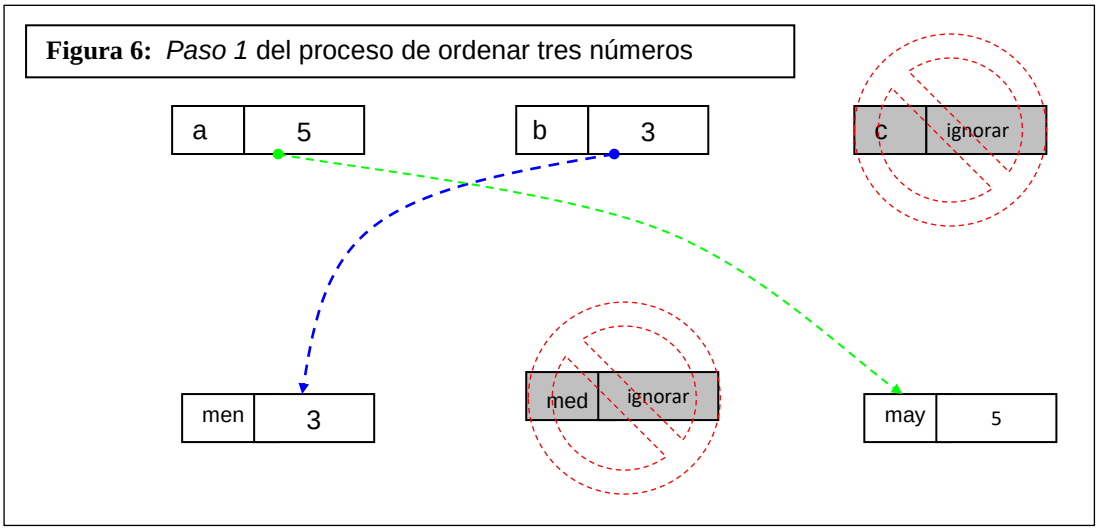


Figura 6: *Paso 1* del proceso de ordenar tres números

Está claro que este primer ordenamiento de dos números puede hacerse con nuestro ya conocido proceso *ordenar_dos_numeros* (que ordena los dos valores *a* y *b* y usa un sola condición...) No importa si el menor estaba en *a* o en *b*, finalmente ese proceso lo asignará en *men*. Y algo similar ocurrirá con el mayor y la variable *may*. Una vez ordenados esos dos primeros números, de aquí en más podemos olvidarnos de las variables *a* y *b*, y seguir trabajando con *men* y *may* (que contienen los mismos valores que *a* y *b*, pero ahora tenemos más información: sabemos cuál de las dos contiene al menor y cuál al mayor.)

El siguiente paso (ahora sí) es abrir el juego y considerar la variable *c* y la variable *med*. Es obvio que tenemos aquí tres posibilidades:

- ✓ El valor de *c* podría ser *mayor que el que tenemos guardado en may*. Sería el caso de la *Figura 6* si *c* valiese 8. Esto significaría que el valor que teníamos como el mayor, es en realidad el valor mediano. Reasignamos primero el valor de *may* en la variable *med*, y luego guardamos *c* en *may*. No tocamos el valor que teníamos en *men* (ya que definitivamente era el menor) (ver *Figura 7*).

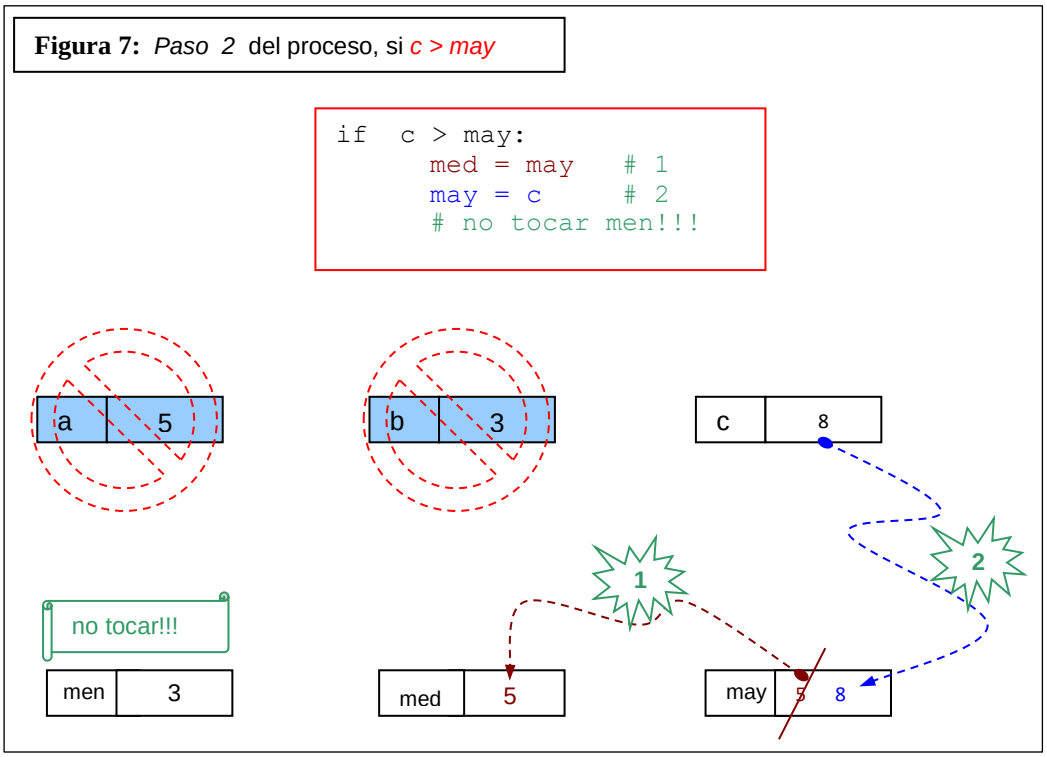
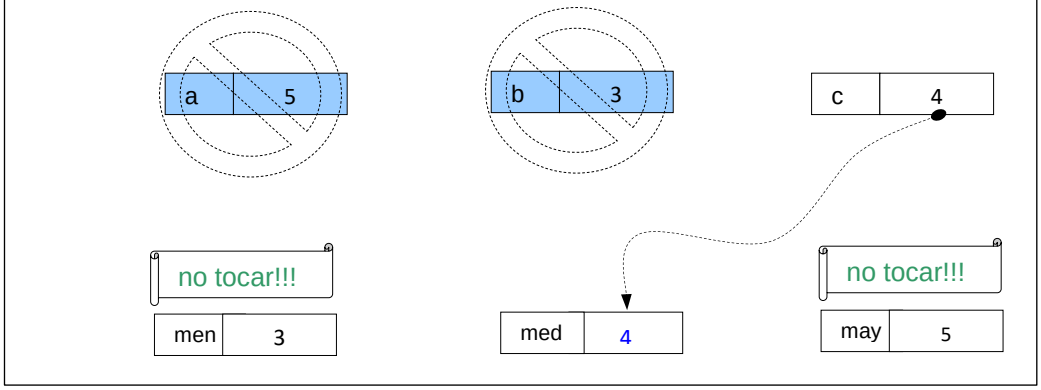


Figura 7: *Paso 2* del proceso, si *c > may*

✓ El valor de *c* podría ser menor que el que tenemos guardado en *may*, pero mayor al que tenemos en *men*. Sería el caso del gráfico de la *Figura 6* si *c* valiese 4. Y esto significaría que el valor que tenemos en *c* es directamente el mediano: sólo tendríamos que asignar *c* en *med*, y dejar *men* y *may* como estaban (ver *Figura 8*).

Figura 8: Tercer paso, si *men < c < may*

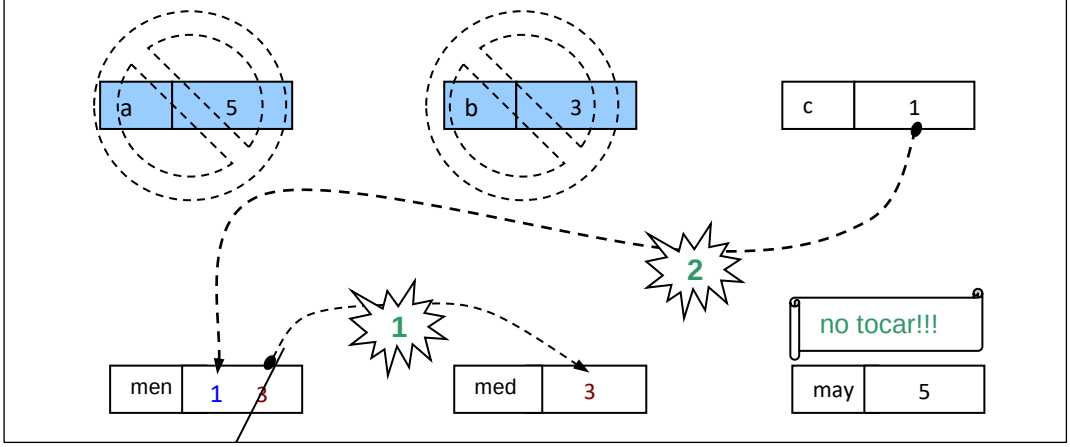
```
if  c > may:
      med = may
      may = c
else:
      if c > men:
            med = c
```



✓ Finalmente, el valor de *c* podría ser menor que el que tenemos guardado en *men*. Sería el caso del gráfico de *la Figura 6* si *c* valiese 1. Esto significaría que el valor que teníamos como el menor, es en realidad el valor mediano. Reasignamos primero el valor de *men* en la variable *med*, y luego guardamos *c* en *men*. No tocamos el valor que teníamos en *may* (ya que definitivamente era el mayor) (ver *Figura 9*).

Figura 9: Cuarto paso, si *c < men*

```
if c > may:
      med = may
      may = c
else:
      if c > men:
            med = c
      else:
            med = men   # 1
            men = c     # 2
            // no tocar may!
```



El algoritmo descripto se conoce como *ordenamiento por inserción*, porque una vez ordenados los dos primeros números en *men* y *may*, se toma el valor *c* y se lo *inserta* en el lugar correcto (delante de *may*, entre *may* y *men* o detrás de *men*, según sea el caso) siguiendo los mecanismos sugeridos.

Después de todo este análisis podemos plantear un programa completo que cargue por teclado los tres números *a*, *b*, *c* y los ordene por inserción. Primero se procede a aplicar el proceso *ordenar_dos_numeros* para el primer paso de ordenar *a* y *b* entre *men* y *may*; y luego se aplican los pasos que hemos visto más arriba para terminar el proceso insertando *c* entre *men* y *may*. Es fácil ver que la cantidad total de condiciones empleadas en todo el proceso *es de sólo tres*: una en el proceso *ordenar_dos_numeros* y otras dos en el proceso que ordenar los tres, con lo que logramos una sustancial mejora respecto de las seis condiciones de nuestro primer intento...[2]

```
__author__ = 'Cátedra de AED'


# Título general y carga de datos...
print('Problema del ordenamiento de tres numeros')
a = int(input('a: '))
b = int(input('b: '))
c = int(input('c: '))

# Primero: ordenar los dos primeros numeros...
if a > b:
    men, may = b, a
else:
    men, may = a, b

# Segundo: ordenar los tres numeros...
if c > may:
    med, may = may, c
else:
    if c > men:
        med = c
    else:
        med, men = men, c

# Visualización de los resultados ordenados...
print('Menor:', men)
print('Medio:', med)
print('Mayor:', may)
```

[2] Y si se trata de resolver problemas ahorrando recursos tanto como sea posible, no podemos dejar de recomendar la película *El Marciano* (*The Martian*) estrenada en 2015 y candidata al Oscar. Fue dirigida por *Ridley Scott* (el mismo que dirigió *Blade Runner*) y protagonizada por *Matt Damon*. Está basada en el reciente best seller del escritor *Andy Weir*. Es la historia de un integrante de una hipotética misión a Marte, que por accidente es dado por muerto y abandonado en el Planeta Rojo. El pobre astronauta deberá resolver a partir de allí una enorme cantidad de problemas para sobrevivir hasta su eventual rescate, con un buen humor infalible y atacando "de a un problema por vez". Imperdible la película. Pero mucho mejor el libro.

Créditos

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

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

Bibliografía

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

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

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