Ficha 4

Estructuras Condicionales

1.] Fundamentos.

Todos los problemas y casos de análisis que hasta aquí hemos presentado tenían en común el hecho de ser problemas simples de lógica lineal: el algoritmo para resolver a cualquiera de esos problemas consistía en una estructura secuencial de instrucciones o pasos (es decir, un conjunto de instrucciones simples como asignaciones, visualizaciones y/o cargas por teclado, una debajo de la otra). Sin embargo, como pronto veremos, esos casos son poco frecuentes en la práctica real de la programación.

En problemas que no sean absolutamente triviales, es muy común que en algún punto se requiera comprobar el valor de alguna condición, y en función de ello proceder a dividir la lógica del algoritmo en dos o más ramas o caminos de ejecución. Por ejemplo, en un programa de control de acceso a un lugar seguro se debe pedir a cada usuario que cargue su clave de identificación. El programa entonces debería controlar si la clave cargada es correcta y sólo en ese caso habilitar el paso a esa persona. Pero si la clave fuese incorrecta, el programa debería tomar alguna medida alternativa, como sacar un mensaje de alerta por la consola de salida, bloquear una puerta, dar aviso a un supervisor, etc. Pero el hecho es que si sólo se emplean estructuras secuenciales de instrucciones, la situación anterior no podría resolverse.

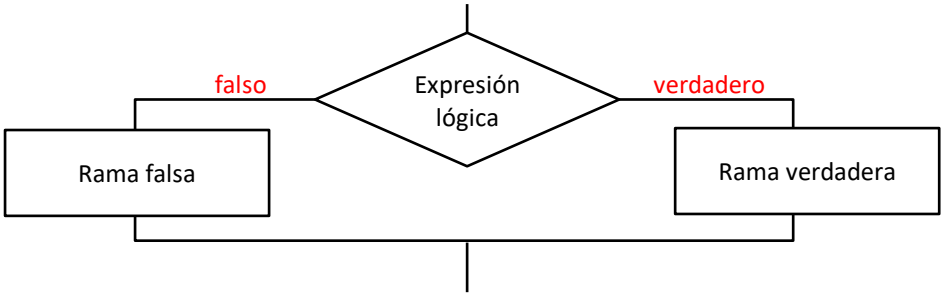
Para casos así, los lenguajes de programación proveen instrucciones específicamente diseñadas para el chequeo de una o más condiciones, permitiendo que el programador indique con sencillez lo que debe hacer el programa en cada caso. Esas instrucciones se denominan *estructuras condicionales*, o bien, *instrucciones condicionales*[1].

En general, una *instrucción condicional* contiene una *expresión lógica* que puede ser evaluada por verdadera o por falsa, y dos bloques de instrucciones adicionales designados en general como *la salida o rama verdadera* y *la salida o rama falsa*. Si un programa alcanza una instrucción condicional y en ese momento la expresión lógica es verdadera, el programa ejecutará las instrucciones de la rama verdadera (y sólo esas). Pero si la expresión es falsa, el programa ejecutará las instrucciones de la rama falsa (y sólo esas). La forma de funcionamiento típica de una instrucción condicional se muestra en forma clara en su representación de diagrama de flujo (ver *Figura 1*):

[1] Programar un computador para que sea capaz de responder preguntas hace un poco más cercana la tarea de imitar el pensamiento humano con ese computador, lo cual fue el pilar del trabajo y del sueño de *Ada Byron*. Y si se trata de sueños combinados con preguntas, en 1968 el escritor *Philip Dick* publicó una novela corta de ciencia ficción llamada (en español): *¿Pueden los androides soñar con ovejas eléctricas?* en la cual se aborda el tema de la línea que separa a lo natural de lo artificial. Esta novela inspiró el guion de la película *Blade Runner* (de 1982, dirigida por *Ridley Scott* y protagonizada por *Harrison Ford* y *Rutger Hauer*)… en opinión de este profesor, la mejor película de ciencia ficción de todos los tiempos.

En el esquema de la *Figura 1*, se escribieron las palabras *verdadero* y *falso* para etiquetar cada rama, pero a lo largo del desarrollo de nuestro curso adoptaremos la convención de asumir que *siempre la rama verdadera estará a la derecha*, y *la falsa a la izquierda*. Por lo tanto, en nuestros diagramas de flujo *no etiquetaremos las ramas*.

Figura 1: Diagrama general de una instrucción condicional típica.



En el lenguaje Python, una instrucción condicional típica como la que mostramos en la figura anterior, se escribe (esquemáticamente) así [1]:

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

La palabra reservada *if* da inicio a la estructura condicional. La *expresión lógica* que se quiere evaluar por verdadero o falso se escribe a continuación, y se cierra la línea con el símbolo *dos puntos* (:). Recordemos en este contexto, que una expresión *lógica* es una fórmula cuyo resultado es un *valor lógico* (o *valor de verdad*) (que en Python son los valores *True* y *False*).

La secuencia de instrucciones que conforma la rama verdadera se escribe luego a renglón seguido, y *respetando la indentación o encolumnado*: no olvide que Python identifica a las instrucciones que pertenecen a un mismo bloque de acuerdo a su encolumnado o nivel de indentación. Por eso, las instrucciones que pertenecen al bloque de la rama verdadera deben escribirse encolumnadas hacia la derecha de la palabra *if*, y todas en la misma columna (de lo contrario, Python interpretará en forma incorrecta la estructura).

Una vez terminado de escribir el bloque de la rama verdadera, a renglón seguido se escribe la palabra reservada *else* (volviendo a la *misma* columna de la palabra *if*) seguida a su vez de *dos puntos* (:) y a partir de una nueva línea se escribe la rama falsa *respetando también el encolumnado*.

El siguiente ejemplo (más concreto) es un script que carga por teclado dos números *a* y *b* y muestra el mayor de ambos en la consola estándar:

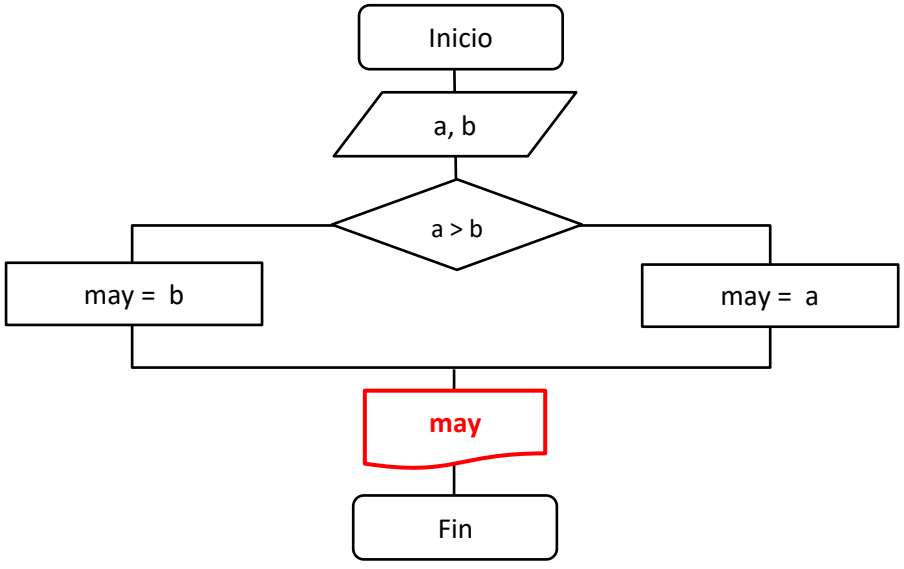
```
a = int(input('A: '))
b = int(input('B: '))

if a > b:
    may = a
else:
    may = b

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

Para determinar cuál es el mayor entre ambos números, se usa una *instrucción condicional* cuya estructura está remarcada en rojo en el ejemplo. En este caso, la *expresión lógica* que se evalúa es *a > b*, la cual puede ser verdadera o falsa dependiendo de los valores cargados en *a* y en *b*. Si efectivamente el valor de *a* fuese mayor que el de *b*, entonces se ejecutará la instrucción *may = a* (y sólo esa instrucción). Pero si *a* no fuese mayor que *b*, entonces el script *salteará la rama verdadera* y ejecutará la instrucción que figura en la rama *else* (la rama falsa): *may = b*.

Note que ***nunca*** se ejecutan ambas ramas de una instrucción condicional para el mismo lote de datos: si la expresión lógica es verdadera, se ejecuta la rama verdadera ***y sólo la rama verdadera***, ignorando la rama falsa. Y si la expresión lógica fuese falsa, se ejecutará la rama falsa (*else*) ***y sólo la rama falsa***, ignorando la rama verdadera.

Observe también la forma en que se escribió la instrucción *print()* de la última línea: se encolumnó *a la misma altura* de la palabra *if* con la que abrió la instrucción condicional, ***y no*** en la misma columna de las instrucciones de la rama falsa. De esta forma, el programador está indicando que ese llamado a *print()* ***no pertenece*** a la rama falsa de la condición, sino que está *completamente fuera* de la rama falsa y de la propia condición. De hecho, en el script anterior, el *print()* de la última línea será ejecutado ***siempre***, sin importar si la condición entró por la rama verdadera o por la falsa (se dice por eso que esa instrucción es de *ejecución incondicional*). El diagrama de flujo del script anterior (ver *Figura 2*) permite aclarar la idea:

Figura 2: Diagrama de flujo del script del cálculo del mayor



El diagrama de la figura anterior muestra en *forma exacta* el funcionamiento lógico del script del cálculo del mayor: si la expresión *a > b* es cierta, se ejecutará la instrucción *may = a* y luego se mostrará el valor de *may*; pero si la expresión es falsa, se ejecutará la instrucción *may = b* y luego también se mostrará el valor de *may*.

Recuerde que un diagrama de flujo se construye *de arriba hacia abajo*, tratando de incluir sólo un símbolo debajo de otro, y ahora que aparece la posibilidad de incluir alguna condición, es bueno incorporar nuevas convenciones de trabajo:

- ✓ Por lo pronto, un diagrama de flujo bien planteado *debería tener un único punto de comienzo y un único punto de final* (como el que se ve en la *Figura 2*). Sólo debería haber un único símbolo de *Inicio* al comienzo y arriba, y un único símbolo de *Fin* al final y abajo.
- ✓ Los distintos símbolos se conectan entre ellos con líneas rectas verticales. Si aparece una condición que abra la lógica en dos o más ramas, se traza una línea horizontal a cada lado, pero apenas se disponga de espacio el trazo debe volver a ser descendente.
- ✓ En cada rama el programador debe especificar los procesos a realizar, y apenas le sea posible debe volver a unir esas ramas en un único camino descendente, para llegar en algún momento a un único final.
- ✓ Y mantendremos la convención de que al aparecer una condición, haremos que la salida por verdadero esté siempre a la derecha, y la salida por falso a la izquierda.

Es muy importante que se comprenda el impacto en Python de escribir cada instrucción con la indentación o encolumnado correcto [2]. En otros lenguajes, como C, C++, Pascal o Java, un bloque de instrucciones se delimita con símbolos o palabras reservadas especiales. Pero en Python, la pertenencia a un bloque se determina por su indentación. A modo de ejemplo, supongamos que el script del cálculo del mayor hubiese sido escrito (incorrectamente…) así:

```
a = int(input('A: '))
b = int(input('B: '))

if a > b:
    may = a
else:
    may = b
    print('El mayor es:', may)
```

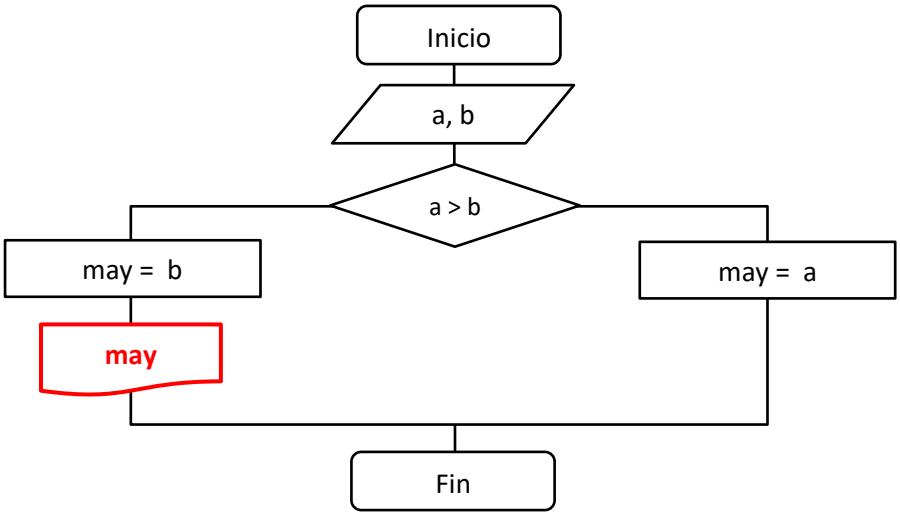
En la forma en que está escrito ahora el script, Python asumirá que la instrucción *print()* para visualizar el valor de *may* pertenece al bloque de la *rama falsa* de la instrucción condicional, y por lo tanto, *sólo será mostrado el valor de may en caso de ser falsa la condición*. Si la condición fuese verdadera, el script terminará sin mostrar nada en la consola de salida. Para terminar de entenderlo, vea el diagrama de flujo que correspondería a este script incorrecto (ver *Figura 3*).

Como se puede ver en esa figura, la lógica es diferente: en la forma en que fue indentada la instrucción *print()* en la segunda versión del script, será entendida como parte del bloque de la rama falsa. Si la condición fuese verdadera, se ejecutará la instrucción *may = a*, pero luego el script terminará y no veremos el valor del mayor.

2.] Expresiones lógicas. Operadores relacionales y conectores lógicos.

En una ficha anterior hemos visto que en general, una *expresión* es una fórmula compuesta por variables y constantes (llamados *operandos*) y por símbolos que indican la aplicación de una acción (llamados *operadores*). Hemos analizado también el uso de los llamados *operadores aritméticos* básicos de Python (suma, resta, producto, etc.) y sabemos que en función de esto, una *expresión aritmética* es una expresión cuyo resultado es un número. Todos los problemas y ejemplos que se analizaron en las tres primeras fichas de estudio, incluían algún tipo de expresión aritmética.

Figura 3: Diagrama de flujo del script *incorrecto* del cálculo del mayor.



Ahora bien: el hecho de que una expresión entregue como resultado un número, se debe a que los operadores que aparecen en ella son operadores aritméticos y por lo tanto llevan a la realización de alguna operación cuyo resultado será numérico. Sin embargo, en todo lenguaje existen operadores cuya acción no implica la obtención de un número como resultado, sino, por ejemplo, *valores lógicos* de la forma *verdadero* o *falso* (*True* o *False* en Python) y algunos otros operadores entregarán resultados de otros tipos (cadenas de caracteres, por ejemplo). En ese sentido, como ya hemos indicado, una *expresión lógica* es una expresión cuyo resultado esperado es un valor de verdad (*True* o *False*).

Como vimos, las *instrucciones condicionales* (y otros tipos de instrucciones que veremos, como las *instrucciones repetitivas*) se basan *típicamente* en chequear el valor de una *expresión lógica* para determinar el camino que seguirá el programa en su ejecución. Pero veremos oportunamente que en Python es posible que la expresión a evaluar no sea necesariamente lógica y sería válida una expresión de cualquier tipo. Por razones de simplificación del análisis inicial, supondremos por ahora que la expresión a evaluar en una instrucción condicional será de tipo lógico.

Para el planteo de *expresiones lógicas*, todo lenguaje de programación provee operadores que implican la obtención de un valor de verdad como resultado. Los más elementales son los llamados *operadores relacionales* u *operadores de comparación*, que en Python son los siguientes [1]:

Figura 4: Tabla de operadores relacionales (o de comparación) en Python.

Operador	Significado	Ejemplo	Observaciones
==	igual que	a == b	retorna *True* si *a* es igual que *b*, o *False* en caso contrario
!=	distinto de	a != b	retorna *True* si *a* es distinto de *b*, o *False* en caso contrario
>	mayor que	a > b	retorna *True* si *a* es mayor que *b*, o *False* en caso contrario
<	menor que	a < b	retorna *True* si *a* es menor que *b*, o *False* en caso contrario
>=	mayor o igual que	a >= b	retorna *True* si *a* es mayor o igual que *b*, o *False* en caso contrario
<=	menor o igual que	a <= b	retorna *True* si *a* es menor o igual que *b*, o *False* en caso contrario

Los operadores de la tabla anterior permiten plantear instrucciones condicionales en Python para comparar de distintas formas dos valores. Hemos visto un ejemplo de aplicación en el

script para calcular el mayor de dos números. Pero note que en Python, los operadores relacionales también permiten *comparar en forma directa dos cadenas de caracteres*: los operadores == (igual que) y != (distinto de) harán lo que se espera: determinar si dos cadenas son iguales o distintas [1] [2]:

```
cad1 = 'Hola'
cad2 = 'Mundo'

if cad1 == cad2:
    print('Son iguales')
else:
    print('No son iguales')

if cad1 != 'Hello':
   print('No es la palabra Hello...')
else:
   print('Es la palabra Hello...')
```

Ahora bien: si se usan los operadores <, <=, >, o >= para comparar cadenas, Python hará lo que conoce como la *comparación lexicográfica*, que no es otra cosa que la forma de comparación de palabras que normalmente aplican las personas cuando intentan ordenar alfabéticamente un conjunto de palabras, o cuando buscan una palabra en un diccionario. En ese sentido, y para simplificar, si se tienen dos cadenas de caracteres *cad1* y *cad2* y se comparan en forma lexicográfica, entonces *cad1* será considerada *menor* que *cad2* si *cad1 estuviese antes* que *cad2* en un diccionario. Considere a modo de ejemplo el siguiente script sencillo:

```
cad1 = 'mesa'
cad2 = 'silla'

if cad1 < cad2:
    print('Orden alfabético:', cad1, cad2)
else:
    print('Orden alfabético:', cad2, cad1)
```

En el ejemplo, la condición chequea si la cadena en *cad1* es menor que la cadena en *cad2* y lo hace automáticamente aplicando el criterio lexicográfico. Estrictamente hablando, Python comparará el primer caracter de cada cadena (los valores *cad1[0]* y *cad2[0]*, que son 'm' y 's' respectivamente). Si fuesen diferentes, tomará como *lexicográficamente menor a la cadena que comience con el caracter que aparezca primero en la tabla Unicode*: en este caso, es el caracter 'm', y por lo tanto, *cad1* será considerada menor que *cad2,* haciendo que la instrucción condicional entre por la rama verdadera.

Según en el criterio lexicográfico, si ambas cadenas tuviesen el mismo primer caracter, Python tomará el segundo de ambas (*cad1[1]* y *cad2[1]*) y aplicará la misma regla. Y si esos caracteres fuesen otra vez iguales, Python continuará con el par que sigue hasta encontrar dos diferentes. Obviamente, en el caso en que todos los caracteres de ambas cadenas fuesen iguales, las cadenas mismas serían iguales y la comparación del ejemplo daría un resultado de *False* (si son iguales, *cad1* ***no es menor*** que *cad2*), activando la rama falsa.

En muchas ocasiones el programador necesitará hacer varias comprobaciones al mismo tiempo (por ejemplo, podría querer comprobar si $a > b$ y al mismo tiempo $a > c$). La forma

más práctica de hacer esas comprobaciones múltiples consiste en aplicar los llamados *conectores lógicos* (en algunos contextos llamados también *operadores lógicos* o incluso *operadores booleanos*), que en Python son los que siguen [1]:

Figura 5: Tabla de conectores lógicos en Python.

Operador	Significado	Ejemplo	Observaciones
and	conjunción lógica (y)	a == b and y != x	ver "Tablas de Verdad" en esta misma sección
or	disyunción lógica (o)	n == 1 or n == 2	ver "Tablas de Verdad" en esta misma sección
not	negación lógica (no)	not x > 7	ver "Tablas de Verdad" en esta misma sección

Un conector lógico u operador booleano es un operador que permite encadenar la comprobación de dos o más expresiones lógicas y obtener un resultado único. En general, cada una de las expresiones lógicas encadenadas por un conector lógico se designa como una proposición lógica. En la columna *Ejemplo* de la tabla anterior, las expresiones a == b, y != x, n == 1, n == 2 y x > 7 son proposiciones lógicas.

Si llamamos en general *p* y *q* a dos proposiciones lógicas cualesquiera, podemos mostrar la forma en que operan los conectores lógicos *and* y *or* mediante tablas que muestran qué resultado se obtiene para cada posible combinación de valores de *p* y de *q*. Esas tablas se suelen designar como *tablas de verdad* [3]. Mostramos aquí las tablas de verdad para los conectores *and* y *or*:

Figura 6: Tablas de verdad de los conectores *and* y *or*.

Tabla de verdad del conector *and*		
p	**q**	**p and q**
True	True	True
True	False	False
False	True	False
False	False	False

Tabla de verdad del conector *or*		
p	**q**	**p or q**
True	True	True
True	False	True
False	True	True
False	False	False

Podemos ver que en el caso del conector ***and*** la salida o respuesta obtenida sólo será verdadera si las proposiciones conectadas son verdaderas al mismo tiempo, y en todo otro caso, la salida será falsa. Por lo tanto, un *and* es muy útil cuando se quiere estar seguro que todas las condiciones impuestas sean ciertas en un proceso. Por caso, suponga que se cargaron por teclado dos variables *sueldo* y *antiguedad* con los datos de un empleado, y se quiere saber si ese empleado gana más de 15000 pesos y tiene al mismo tiempo una

antigüedad de por lo menos 10 años, para decidir si se le otorga o no un crédito. Como *ambas* condiciones son exigibles a la vez, la pregunta requiere un conector *and*, y en Python puede plantearse así:

```
if sueldo > 15000 and antiguedad >= 10:
    print('Crédito concedido')
else:
    print('Crédito rechazado')
```

Note que si alguna de las dos proposiciones fuese falsa, la condición completa sería falsa y se activaría la rama *else*, rechazando el crédito. Por supuesto, lo mismo ocurriría si ambas proposiciones fuesen falsas al mismo tiempo. *Sólo si ambas fuesen ciertas*, la condición sería cierta ella misma, y se activaría la rama verdadera.

El conector ***or*** opera de forma diferente: la salida será verdadera si al menos una de las proposiciones es verdadera. Sólo si *todas* las proposiciones al mismo tiempo son falsas, se obtendrá un falso como respuesta. El uso de un ***or*** es valioso (por ejemplo) cuando se quiere determinar si una variable que se acaba de cargar por teclado vale uno de varios posibles valores que se consideran correctos. Por ejemplo, supongamos que queremos saber si la variable *opcion* fue cargada con un 1, un 3, o un 5 (cualquiera de los tres valores es correcto en este ejemplo). En Python, podemos hacerlo así:

```
if opcion == 1 or opcion == 3 or opcion == 5:
    print('Opción correcta')
else:
    print('Opción incorrecta')
```

En este caso, la instrucción condicional se usa para comprobar si el valor de la variable *opcion* coincide con alguno de los números 1, 3 o 5. Cualquiera de las tres proposiciones que fuese cierta, haría cierta también la condición completa y se activaría la rama verdadera. Sólo si todas las proposiciones fuesen falsas, se activaría la rama *else* (por ejemplo, si el valor contenido en la variable *opcion* fuese el 7).

Como vimos, los conectores *and* y *or* se aplican sobre dos o más proposiciones (en general, si un operador aplica sobre dos operandos, se lo suele designar como un *operador binario*). Pero el operador *not* (negación lógica) aplica sobre una sola proposición (y por lo tanto se lo suele designar como un *operador unario*) y su efecto es obtener el *valor opuesto* al de la proposición negada. Por lo tanto, la tabla de verdad del *negador lógico* es trivial:

Figura 7: Tabla de verdad del conector *not*.

Tabla de verdad del conector *not*	
p	**not p**
True	False
False	True

En general, el uso indiscriminado del negador lógico suele llevar a condiciones difíciles de leer y entender por parte de otros programadores, por lo que sugerimos se aplique con cuidado y sentido común. A modo de ejemplo, supongamos que se tiene una variable *edad* con la edad de una persona, y se quiere saber si esa persona tiene al menos 18 años para

decidir si puede acceder al permiso de conducir. Una forma de hacerlo en Python, sería preguntar si la edad ***no es*** menor que 18:

```
if not edad < 18:
    print('Puede acceder al permiso')
else:
    print('No puede acceder al permiso')
```

Si bien lo anterior es correcto y cumple efectivamente con el requerimiento, queda claro que la pregunta podría reformularse de forma de eliminar el negador, dejándola más clara. Sólo debemos notar que preguntar si *edad* no es menor que 18, es exactamente lo mismo que preguntar si *edad* es mayor o igual que 18. El script podría replantearse así:

```
if edad >= 18:
    print('Puede acceder al permiso')
else:
    print('No puede acceder al permiso')
```

3.] Precedencia de ejecución de los operadores relacionales y de los conectores lógicos.

En cuanto a la precedencia de ejecución, en Python todos los *operadores relacionales* (o *de comparación*) tienen la *misma precedencia*, y a su vez esta es *mayor que la de los conectores lógicos*, pero *menor* que la de los *operadores matemáticos* [1]. Obviamente, el uso de paréntesis permite al programador cambiar esas precedencias según sus necesidades. A modo de ayuda mnemotécnica, podemos usar la siguiente relación general:

precedencia(conectores) < precedencia(relacionales) < precedencia(matemáticos)

De esta forma, en cualquier expresión lógica sencilla o compleja el lenguaje agrupará los términos y resolverá *primero las operaciones aritméticas*. Luego, aplicará los *operadores de comparación*, y finalmente ejecutará los *conectores lógicos*.

Sabemos que los operadores aritméticos tienen distinta precedencia (por ejemplo, las sumas y las restas son de menor precedencia que los productos, los cocientes y los restos) y serán aplicados de acuerdo a ella, de izquierda a derecha (salvo el operador de exponenciación (**) que tiene precedencia mayor).

Los operadores relacionales o de comparación tienen la misma prioridad todos ellos, y serán aplicados de izquierda a derecha. En particular, en Python es especialmente notable el hecho de que estos operadores pueden aplicarse en forma idéntica a como se hace en la notación algebraica normal: una instrucción condicional como:

```
if a > b > c:
```

en Python ejecutará sin problemas [1], significando exactamente lo que se quiere hacer: preguntar si $a > b$ y al mismo tiempo preguntar si $b > c$. La misma instrucción podría haber sido escrita así, equivalentemente:

```
if a > b and b > c:
```

Además, es oportuno indicar que tanto el conector *and* como el *or* en Python actúan en forma *cortocircuitada*, lo cual quiere decir que dependiendo del valor de la primera

proposición evaluada, la segunda (o las restantes a partir de ella) podrían no llegar a ser evaluadas (y esto se deduce y se justifica a partir de las *tablas de verdad* de ambos conectores). La idea es la siguiente [1]:

Conector	Ejemplo	Efecto del cortocircuito
and	if a > b **and** a > c:	Si la primera proposición es *False* (*a > b* en este caso) la segunda (*a > c*) *no se chequea* y la salida también es *False*.
or	if n < 0 **or** n > 9:	Si la primera proposición es *True* (*n < 0* en este caso), la segunda (*n > 9*) *no se chequea* y la salida también es *True*.

De acuerdo a las *tablas de verdad*, es fácil ver que si se tiene un *and* y la primera proposición ya es *False*, no importará el valor de la segunda: la salida será *False* sea cual sea el valor de la segunda, ya que en un *and* es suficiente un *False* en una de ellas para obtener una salida *False*. En forma similar, si se tiene un *or* y la primera proposición es *True*, la salida será *True* sin importar el valor de verdad de la segunda. En Python, estos hechos están ya incorporados en la forma en que el intérprete trabaja cuando evalúa una expresión lógica. Incluso cuando la expresión esté planteada en forma algebraica lineal:

```
if a > b > c:
```

el intérprete Python chequeará la proposición *a > b*, y si esta fuese *False*, no evaluará el valor de *b > c* y retornará directamente un *False* como salida.

Con respecto a los conectores lógicos, el *not* es de mayor precedencia que el *and*, y este a su vez es de mayor precedencia que el *or*. Si aparecen dos conectores de la misma precedencia, se aplicarán de izquierda a derecha. Las precedencias pueden esquematizarse así:

precedencia(**or**) < precedencia(**and**) < precedencia(**not**)

Vale decir: si en una misma expresión lógica aparecen varios conectores *and*, *or* o *not* entremezclados, sin paréntesis que cambien las prioridades, los *not* se aplicarán primero, luego los *and*, y finalmente los *or*. Mnemotécnicamente, puede decirse que el conector *not* equivale a un *signo menos* cambiando el signo en una expresión matemática, el *and* equivale a un *producto*, y el *or* a una *suma*[2].

Para el ejemplo que sigue, recuerde que cualquier expresión de cualquier tipo (aritmética, lógica, etc.) entrega un resultado y ese resultado puede ser asignado en una variable o usado en el contexto que se requiera (como una instrucción condicional). En este caso, mostramos una expresión lógica cuyo valor final (un *True* o un *False*) se asigna en la variable *r1* (que en consecuencia será de tipo *boolean*) (En el desarrollo del ejemplo mostraremos todas las evaluaciones de todas las proposiciones, para dejar en claro el orden en que proceden, pero recuerde que los conectores *and* y *or* son *cortocircuitados*, por lo que algunos de los chequeos de proposiciones que siguen, podrían no llegar a efectuarse realmente):

```
# Considere estas variables con estos valores:
a, b, c, d, e = 3, 5, 7, 2, 3
```

[2] De hecho, el operador *and* se designa también como el *producto lógico*, y el operador *or* como la *suma lógica*.

```
# Ejemplo: valor final asignado en r1: True
r1 = a > b and c == 3*d or not e != a

# Desarrollo paso a paso: reemplazo de variables por sus valores
# r1 = 3 > 5 and  7 == 3*2 or not 3 != 3

# Desarrollo paso a paso: primero los operadores aritméticos
# r1 = 3 > 5 and  7 == 3*2 or not 3 != 3
# r1 = 3 > 5 and  7 == 6 or not 3 != 3

# Desarrollo paso a paso: segundo los operadores relacionales
# r1 = 3 > 5 and  7 == 6 or not 3 != 3
# r1 = False and False or not False

# Desarrollo paso a paso: tercero los not, and y or en ese orden...
# r1 = False and False or not False
# r1 = False and False or True
# r1 = False or True
# r1 = True
```

Veamos ahora el ***mismo*** ejemplo, pero cambiando las precedencias con ***paréntesis***:

```
# Considere estas variables con estos valores:
a, b, c, d, e = 3, 5, 7, 2, 3

# Ejemplo: valor final asignado en r1: False
r1 = a > b and (c == 3*d or not e != a)

# Desarrollo paso a paso: reemplazo de variables por sus valores
# r1 = 3 > 5 and (7 == 3*2 or not 3 != 3)

# Desarrollo paso a paso: primero los operadores aritméticos
# r1 = 3 > 5 and  (7 == 3*2 or not 3 != 3)
# r1 = 3 > 5 and  (7 == 6 or not 3 != 3)

# Desarrollo paso a paso: segundo los operadores relacionales
# r1 = 3 > 5 and  (7 == 6 or not 3 != 3)
# r1 = False and (False or not False)

# Desarrollo paso a paso: tercero los not, and y or en ese orden...
# ...pero resolviendo los paréntesis antes de aplicar and y or...
# r1 = False and (False or not False)
# r1 = False and (False or True)
# r1 = False and True
# r1 = False
```

Como cierre de esta sección, considere que una variable de tipo *boolean* constituye en sí misma una *expresión lógica* trivial, por lo que si el programador necesita comprobar su valor en una instrucción condicional no es necesario hacer *explícitamente la comparación mediante operadores relacionales*. La siguiente secuencia de instrucciones:

```
r1 = a > b and (c == 3*d or not e != a)
if r1 == True:
        print('Verdadero...')
else:
        print('Falso...')
```

podría haber sido escrita, equivalentemente, *replanteando la condición de esta forma implícita*:

```
r1 = a > b and (c == 3*d or not e != a)
if r1:
        print('Verdadero...')
else:
        print('Falso...')
```

Como se ve, la sola presencia de la variable es suficiente: si su valor es *True*, la instrucción condicional activará la rama verdadera, y si es *False*, entrará por la falsa. En forma similar, si lo que se desea es *invertir la lógica* de *la comprobación* (esto es, activar la rama verdadera si la variable *r1* vale *False*, y viceversa) está claro que puede hacerse con una *condición explícita* en la forma siguiente:

```
r1 = a > b and (c == 3*d or not e != a)
if r1 == False:
        print('Falso...')
else:
        print('Verdadero...')
```

Pero se puede hacer exactamente lo mismo con una *condición implícita*, simplemente negando el valor de la variable *r1* con el operador *not* (cuya función es justamente esa: invertir la lógica de una comprobación):

```
r1 = a > b and (c == 3*d or not e != a)
if not r1:
        print('Falso...')
else:
        print('Verdadero...')
```

Tómese un par de minutos para analizar este último script y comprender que es *estrictamente equivalente al anterior*: si la variable *r1* vale *False*, el operador *not* obtendrá un *True* y la instrucción condicional entrará por la rama verdadera. Y si vale *True*, el *not* obtendrá un *False* y el camino seguirá por la salida falsa.

4.] Aplicaciones prácticas básicas.

Presentamos en esta sección algunos problemas resueltos, de naturaleza sencilla, que incluyen el uso de condiciones.

Problema 8.) *Cargar por teclado dos números enteros. Mostrarlos ordenados de menor a mayor.*

a.) Identificación de componentes:

- **Resultados:** Un listado ordenado de dos números. (*men, may*: int)
- **Datos:** Dos números enteros. (*n1*, *n2*: int)
- **Procesos:** Problema de *ordenamiento de un conjunto*. La idea es que dados los dos números originales *n1* y *n2*, se obtengan otras dos variables *men* y *may* que contengan respectivamente al menor y al mayor de los valores *n1* y *n2*, y luego se muestren los valores finales de *men* y *may*, siempre en ese orden para lograr la visualización ordenada de los datos originales [4].

Esto puede lograrse en forma simple con una instrucción condicional como la que sugerimos a continuación:

```
si n1 > n2:
    may = n1
    men = n2
sino:
    may = n2
    men = n1
```

Si la expresión lógica *n1 > n2* es cierta, el orden entre *n1* y *n2* queda totalmente definido (sin necesidad de hacer otra pregunta): *n1* es el mayor (y ese asigna en *may*) y *n2* es el menor (y se asigna en *men*). Y si la expresión *n1 > n2* fuese falsa, entonces el orden *también queda definido* pero a la inversa: se asigna *n2* en *may*, y luego *n1* en *men*.

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

Figura 8: Pseudocódigo y diagrama de flujo del problema del ordenamiento de dos números.

Algoritmo:

1.) Cargar n1 y n2: dos números enteros.

2.) Si n1 > n2:

   2.1.) may = n1

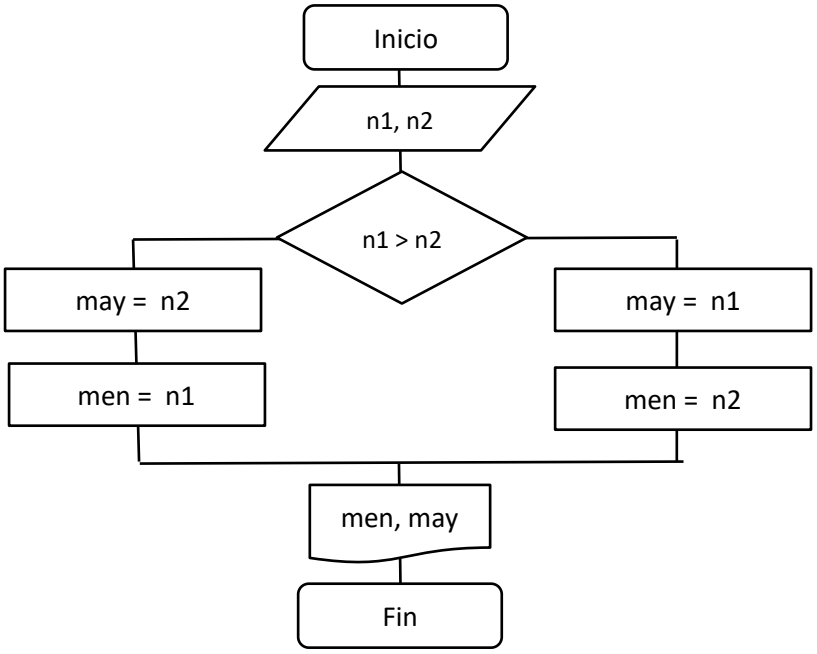
   2.2.) men = n2

3.) sino:

   3.1.) may = n2

   3.2.) men = n1

4.) Mostrar men y may: el menor y el mayor, *siempre* en ese orden.



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 ordenamiento de dos números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))

# Procesos...
if n1 > n2:
    may = n1
    men = n2
else:
    may = n2
    men = n1

# Visualización de resultados..
print('Números ordenados:', men, ' ', may)
```

La instrucción condicional que aparece en este script sigue fielmente la estructura del pseudocódigo que mostramos más arriba. Insistimos en hacer notar la importancia de la *correcta indentación de las instrucciones*, sobre todo las que aparecen en las ramas verdadera y falsa de la condición.

Problema 9.) *Cargar por teclado tres números enteros. Determinar si el primero que se cargó es el mayor de los tres (informe en pantalla con un mensaje tal como: **Es el mayor** o **No es el mayor**).*

a.) Identificación de componentes:

- **Resultados:** Un mensaje. (*mensaje: cadena de caracteres*)
- **Datos:** Tres números enteros. (*n1*, *n2, n3*: int)
- **Procesos:** Problema de *determinación del mayor de un conjunto*. Si el primero de los números se carga en la variable *n1*, entonces lo que se debe hacer es comprobar si el valor cargado en *n1* resultó mayor al que se hubiese cargado en *n2* y *n3*.

  Esto puede lograrse en con una instrucción condicional como la que sugerimos a continuación, empleando un conector tipo *and* (o sea, un *y lógico*):

```
si n1 > n2 y n1 > n3:
    mensaje = 'El primero es el mayor'
sino:
    mensaje = 'El primero no es el mayor'
```

  Si la expresión lógica *n1 > n2 y n1 > n3* es cierta, entonces el valor almacenado en *n1* es el mayor de los tres (recuerde que un *y lógico* sólo es verdadero si todas las proposiciones evaluadas son ciertas). En ese caso se asigna en la variable *mensaje* la cadena de caracteres *'El primero es el mayor'*. Y si la expresión *n1 > n2 y n1 > n3* fuese falsa, entonces *n1* no contiene al mayor de los tres: al menos uno de los otros dos números (*n2*, *n3 o ambos*) es mayor que *n1* (pues el *y lógico* salió por falso). En este caso entonces, la variable *mensaje* es asignada con la cadena *'El primero no es el mayor'*.

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

Figura 9: Pseudocódigo y diagrama de flujo del problema del ordenamiento de dos números.

Algoritmo:

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

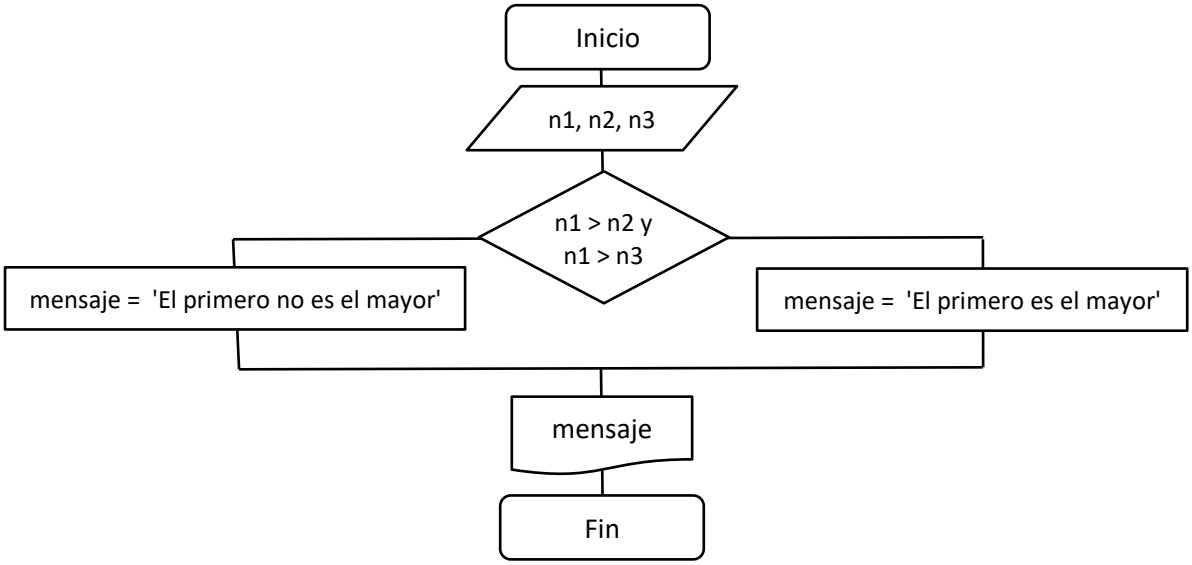
2.) Si n1 > n2 y n1 > n3:

    2.1.) mensaje = 'El primero es el mayor'

3.) sino:

    3.1.) mensaje = 'El primero no es el mayor'

4.) Mostrar mensaje.



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 de determinar si el primero es el mayor')
n1 = int(input('N1: '))
n2 = int(input('N2: '))
n3 = int(input('N3: '))

# Procesos...
if n1 > n2 and n1 > n3:
    mensaje = 'El primero es el mayor'
else:
    mensaje = 'El primero no es el mayor'

# Visualización de resultados..
print('Conclusión:', mensaje)
```

Note que la instrucción *print()* que muestra en pantalla el mensaje pedido, está ubicada *luego de la instrucción condicional*, cuando la misma ha terminado. Sea cual sea el valor de la expresión lógica (*True* o *False*) la instrucción *print()* será ejecutada y el valor de la variable *mensaje* se mostrará correctamente.

Problema 10.) *Cargar por teclado tres números enteros que se supone representan las edades de tres personas. Determinar si alguno de los valores cargados era negativo, en cuyo caso informe en pantalla con un mensaje tal como:* ***Alguna es incorrecta: negativa****. Si todos los valores eran positivos o cero, informe que todas eran correctas.*

a.) Identificación de componentes:

- **Resultados:** Un mensaje. (*mensaje: cadena de caracteres*)
- **Datos:** Tres números enteros. (*n1*, *n2, n3*: int)
- **Procesos:** Problema de *validación de valores de un conjunto*. Una vez cargados los tres valores se puede usar una condición que chequee cada una de las variables controlando si su valor es negativo, y usar un conector *o lógico* para armar la expresión completa.

  Esto puede lograrse en con una instrucción condicional como la que sugerimos a continuación:

```
si n1 < 0 o n2 < 0 o n3 < 0:
    mensaje = 'Alguna es incorrecta: negativa'
sino:
    mensaje = 'Todas son correctas'
```

  Si la expresión lógica *n1 < 0 o n2 < 0 o n3 < 0* es cierta, entonces alguno de los tres valores es negativo (al menos uno lo es, aunque podrían serlo dos de ellos o incluso los tres). Recuerde que un *o lógico* es verdadero si al menos una de las proposiciones evaluadas es cierta. En ese caso se asigna en la variable *mensaje* la cadena de caracteres *'Alguna es incorrecta: negativa'*. Y si la expresión fuese falsa, entonces *ninguno de los valores era negativo*. En este caso la variable *mensaje* es asignada con la cadena *'todas son correctas'*.

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

Figura 10: Pseudocódigo y diagrama de flujo del problema del ordenamiento de dos números.

Algoritmo:

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

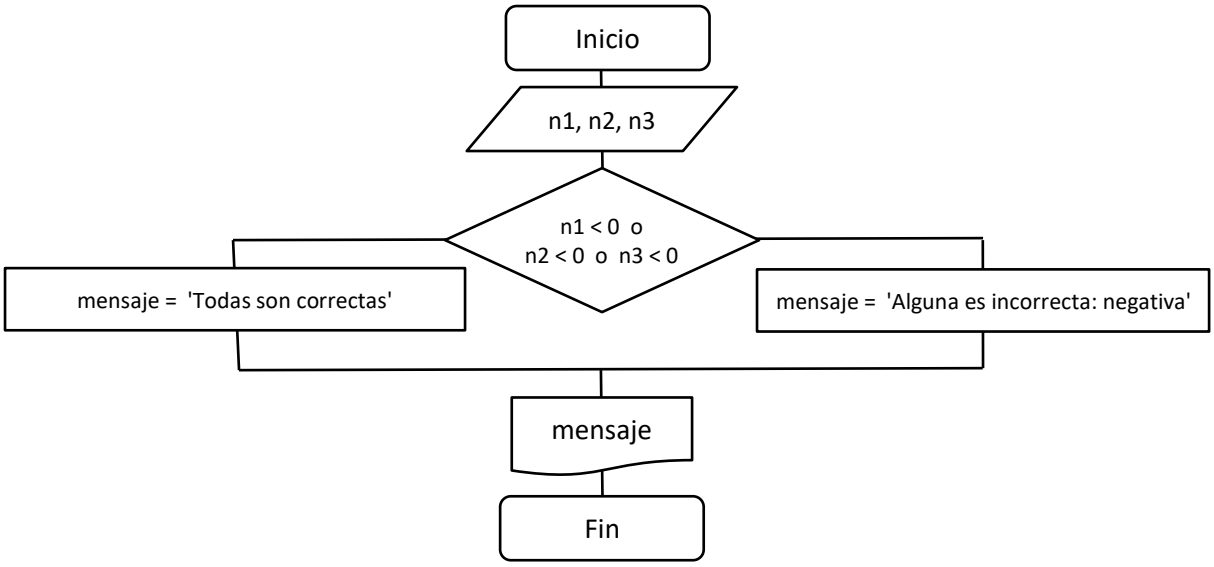
2.) Si n1 < 0 o n2 < 0 o n3 < 0:

  2.1.) mensaje = ' Alguna es incorrecta: negativa '

3.) sino:

  3.1.) mensaje = 'Todas son correctas'

4.) Mostrar mensaje.



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 de determinar si alguna edad es negativa')
n1 = int(input('Edad 1: '))
n2 = int(input('Edad 2: '))
n3 = int(input('Edad 3: '))

# Procesos...
if n1 < 0 or n2 < 0 or n3 < 0:
    mensaje = ' Alguna es incorrecta: negativa '
else:
    mensaje = 'Todas son correctas'

# Visualización de resultados..
print('Resultado del control:', mensaje)
```

Otra vez, la instrucción *print()* que muestra en pantalla el mensaje pedido está ubicada *luego de la instrucción condicional*, y sea cual sea el valor de la expresión lógica (*True* o *False*) la instrucción *print()* será ejecutada y el valor de la variable *mensaje* se mostrará correctamente.

5.] Generación de valores aleatorios[3].

Para cerrar los temas generales de esta Ficha, introduciremos brevemente la forma práctica de hacer que un programa en Python genere números al azar (o números aleatorios), lo cual es de mucha utilidad en aplicaciones que se basan en desarrollos probabilísticos (como por ejemplo, el desarrollo de juegos de computadora).

En muchos problemas, se requiere poder pedirle al programa que almacene en una o más variables algún número seleccionado al azar (o al menos, en una forma que no sea

[3] Parte del contenido de esta sección fue aportado por la ing. *Romina Teicher* en ocasión del planteo del enunciado general del Trabajo Práctico 1.

predecible por el usuario ni por el mismo programador). Esto se conoce como *generación de números aleatorios* (o *números random*) y en general, todo lenguaje de programación provee un cierto número de funciones predefinidas que permiten hacer esto en forma aceptable.

Conceptualmente hablando, estas funciones no generan *realmente* números en forma completamente aleatoria, sino que se basan en algún algoritmo que partiendo de un valor inicial (llamado el *valor semilla* o *seed* del generador) es capaz de generar una secuencia de aspecto aleatorio para cualquiera que no conozca el valor inicial (lo cual suele ser suficiente para aplicaciones generales que requieran algún nivel de aleatoriedad, como los video juegos). Esto se conoce como *generación de números pseudo-aleatorios* y en general los lenguajes de programación suelen tomar ese valor semilla en forma automática desde el reloj interno del sistema (pero como veremos, el programador puede cambiar eso e indicar al lenguaje el uso de una semilla específica, si lo prefiere).

Dejando de lado estos elementos (que por ahora tomaremos como tecnicismos), en Python existe lo que se conoce como un *módulo* (o una *librería*) llamado *random*, que contiene las definiciones y funciones necesarias para poder acceder a la gestión de números *pseudo-aleatorios* [1]. Veremos en una ficha posterior la forma de gestionar módulos en Python, pero por ahora baste con saber que para poder usar el contenido del módulo *random* en un programa, debe usarse la instrucción *import random* al inicio del script en donde se requiera. Luego de esto, el programa podrá acceder a las funciones contenidas en ese módulo invocando a esas funciones pero precediendo la invocación con el prefijo "*random.*" A modo de ejemplo, el siguiente script sencillo obtiene un valor aleatorio entre 0 y 1 con la función *random.random()*, lo almacena en una variable *x*, y finalmente lo muestra:

```
__author__ = 'Cátedra de AED'

import random

x = random.random()
print(x)
```

Concretamente, al escribir *random.random()* se está indicando al intérprete que debe buscar la función *random()* dentro del módulo *random* que fue habilitado con la instrucción *import* anterior. La función *random.random()* calcula y retorna lo que se conoce como un *valor de probabilidad*: un número en coma flotante pseudo-aleatorio dentro del intervalo [0, 1) (incluye al cero como posible salida, pero no al 1).

Si lo que necesita el programador es obtener un valor aleatorio de tipo *int* (y no un *float*) en un rango o intervalo entero específico, puede usar la función *random.randrange(a, b)* que obtiene y retorna un número entero pseudo-aleatorio pero comprendido en el intervalo entero *[a, b-1]* (incluye al valor *a* como posible salida, pero no al *b*). En el ejemplo siguiente, la variable *y* quedará asignada con un número al azar tomado del intervalo [2, 9] (recuerde: en general en Python, las funciones que trabajan sobre un rango o intervalo de valores, *incluyen al límite izquierdo* de ese intervalo, pero *excluyen al límite derecho*) [1]:

```
__author__ = 'Cátedra de AED'

import random
```

```
# número aleatorio entero y tal que 2 <= y < 10
y = random.randrange(2, 10)
print(y)
```

Una alternativa es la función *random.randint(a, b)* que permite hacer lo mismo que *random.randrange(a, b)*, pero de forma tal que el número retornado estará en el intervalo *[a, b]* (incluirá a *b* como posible salida):

```
__author__ = 'Cátedra de AED'

import random

# número aleatorio entero y tal que 2 <= y <= 10
y = random.randint(2, 10)
print(y)
```

El módulo *random* provee muchas otras funciones para manejo de números pseudo-aleatorios, que en este momento escapan al alcance de lo visto hasta ahora en el curso. Sin embargo, queda al menos una que puede ser de interés inmediato: La función *random.choice(sec)* acepta como parámetro una *secuencia* (que aquí llamamos *sec*) (o sea, una tupla, una cadena, una lista, etc.) y retorna un elemento cualquiera de esa secuencia, elegido al azar. El siguiente script usa dos veces esta función: la primera vez para obtener al azar un número cualquiera de la tupla *sec1 = 2, 10, 7, 9, 3, 4* y la segunda vez para obtener al azar una letra cualquiera de la cadena *sec2 = 'ABCDEFGHI'*:

```
__author__ = 'Cátedra de AED'

import random

sec1 = 2, 10, 7, 9, 3, 4
r1 = random.choice(sec1)
print(r1)

sec2 = 'ABCDEFGHI'
r2 = random.choice(sec2)
print(r2)
```

Para introducir el tema que sigue, mostramos nuevamente el programa que ordena dos números (problema 8 en esta misma Ficha), pero modificado levemente para que los dos números de entrada sean generados en forma aleatoria (tomando números entre 1 y 10), en lugar de ser cargados por teclado:

```
__author__ = 'Cátedra de AED'


import random

# Título general y generación de datos...
print('Problema del ordenamiento de dos números')
print('(los numeros de entrada son generados aleatoriamente...')
n1 = random.randint(1,10)
n2 = random.randint(1,10)

# Procesos...
if n1 > n2:
    may = n1
```

```
    men = n2
else:
    may = n2
    men = n1
# Visualización de resultados..
print('Números ordenados:', men, ' ', may)
```

Como se ve, se usa dos veces la función *random.randint(a, b)* para generar los dos números enteros en forma aleatoria. Así como está planteado, usted puede ejecutar el programa tantas veces quiera y puede estar razonablemente seguro que obtendrá números diferentes en cada ejecución (o por lo menos, puede estar seguro que no podrá predecir en forma simple cuáles serán los números generados). Como dijimos, esto es así porque el algoritmo usado por las funciones (de cualquier lenguaje de programación) que generan números supuestamente aleatorios aplican una fórmula que parte de un valor numérico inicial (o *semilla (seed) del generador*), que normalmente se toma del reloj interno del sistema. Como el estado del reloj interno cambia continuamente, es de esperar que en consecuencia la semilla del generador cambie en cada ejecución del programa y eso produce secuencias de números que a primera vista son impredecibles.

Sin embargo, en algunas ocasiones un programador podría querer configurar el uso de un valor conocido y específico para la semilla del generador, sin que esa semilla se tome en forma automática. Si la semilla provista es siempre la misma cada vez que un programa se ejecuta, *entonces inevitablemente la secuencia de valores generados será también la misma en cada ejecución*. Un ejemplo común se da en situaciones de evaluación en cursos de programación, o en situaciones de prueba en equipos de trabajo, en las que se pide a los programadores desarrollar un programa que procese una secuencia extensa y específica de datos, para luego comprobar si el programa obtiene los resultados correctos.

En Python el programador puede asignar el valor que desee a la semilla del generador usando la función *random.seed(a)*, donde ***a*** es el valor que se quiere usar como semilla desde allí en adelante. Y eso es todo. Solo se debe llamar a esta función *antes* de comenzar a usar las funciones de generación de valores aleatorios en el programa, y esas funciones harán sus cálculos tomando el valor indicado como semilla. Pero como dijimos, si ese valor es siempre el mismo entonces el programa generará siempre la misma secuencia. Mostramos el mismo programa que genera y ordena dos números aleatorios, pero aplicando ahora la función *random.seed(a)* para fijar la semilla:

```
__author__ = 'Cátedra de AED'


import random

# Título general y generación de datos...
print('Problema del ordenamiento de dos números')
print('(los numeros de entrada son generados aleatoriamente...')
random.seed(75)
n1 = random.randint(1,10)
n2 = random.randint(1,10)

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

```
        men = n1
    # Visualización de resultados..
    print('Números ordenados:', men, ' ', may)
```

Si se observa, el único cambio en este ejemplo es la línea resaltada en rojo con la instrucción *random.seed(75)*. Si se ejecuta el programa una vez, los números generados serán el 8 y el 10. Y en cada ejecución que se realice los números obtenidos serán siempre el 8 y el 10. Cambie el valor de la semilla, o elimine la llamada a la función *random.seed()*, si desea cambiar la secuencia.

6.] Expresiones lógicas: forma general de evaluación en Python.

Hemos visto a lo largo de las secciones de esta Ficha la forma de plantear una instrucción condicional en Python y hemos indicado que en general, la forma típica de una instrucción condicional incluye una *expresión lógica* cuyo valor *True* o *False* determina el camino que seguirá la ejecución del programa. Hemos indicado además, que una *expresión lógica* es una fórmula cuyo resultado es un *valor de verdad*, y este hecho surge de la utilización de *operadores relacionales* (o *de comparación*) combinados eventualmente con *conectores lógicos*.

Esto es: si se aplican operadores relacionales y conectores lógicos, la expresión entregará un resultado *True* o *False* y será por ello una expresión lógica. Sin embargo, debemos hacer notar que en Python pueden combinarse estos operadores en formas que podrían resultar extrañas para los programadores poco experimentados. Esta sección está dedicada a estudiar esas aplicaciones.

Específicamente, en Python, la aplicación de *conectores lógicos* (u *operadores booleanos*) como *and*, *or* o *not* hace que *todo operando (constantes o variables) sea considerado como booleano*, incluso si su valor inicial no es de tipo *boolean*. En ese sentido, en una expresión que contenga operadores booleanos, Python asumirá que los siguientes valores equivalen a un *False* [1]:

- *False*
- *None*
- El valor numérico *0(cero)* de todos los tipos
- Las *cadenas de caracteres vacías*, y en general, *todos los objetos contenedores (o estructuras de datos) vacíos* (incluyendo cadenas, tuplas, listas, diccionarios y conjuntos de distintos tipos)

Y en correspondencia con esto, en una expresión con operadores booleanos Python interpretará como un *True* a cualquier otro valor: números diferentes de cero, cadenas o contenedores no vacíos, y el propio valor *True*.

Lo anterior también es aplicable *en cualquier contexto en el cual Python espere encontrar una expresión lógica* (por ejemplo, en una *instrucción condicional* o en una *instrucción repetitiva*), haya o no *operadores booleanos and*, *or* y *not* en la expresión analizada.

En lo inmediato, esto implica que si se quiere simplemente chequear si el valor de una variable (posiblemente numérica) es *diferente de cero*, no necesita el programador hacer la *comparación explícita* mediante operadores relacionales, sino que puede simplemente escribir la variable y *comprobarla en forma implícita como si fuese booleana*: el valor *cero*

será interpretado como *False* y cualquier otro valor será tomado como un *True*. La siguiente comprobación explícita:

```
n = int(input('Ingrese un número entero: '))

if n != 0:
    print('Es diferente de cero')
else:
    print('Es igual a cero')
```

puede hacerse también de *manera implícita*, aun cuando la variable *n* contiene un número, en la forma siguiente:

```
n = int(input('Ingrese un número entero: '))

if n:
    print('Es diferente de cero')
else:
    print('Es igual a cero')
```

Insistimos: en este último ejemplo la variable *n* está siendo usada directamente en el lugar donde Python *esperaría una expresión lógica*, y por lo tanto, el valor de *n* es interpretado *según ese contexto en forma booleana*: si su valor fuese cero, Python lo tomará como un *False* y activará la rama falsa de la instrucción condicional; pero si *n* valiese cualquier otro número, Python lo interpretará como un *True*, activando la rama verdadera.

En forma similar, si la variable analizada fuese una cadena de caracteres, una tupla, una lista u otro tipo de contenedor o variable estructurada se puede chequear en forma implícita si esa variable está vacía o no. En el script siguiente, la variable *cad* representa una *cadena de caracteres* (que asignamos en forma fija para hacer más claro el ejemplo) y *se chequea en forma explícita* si *cad* realmente contiene una cadena o está vacía (la cadena vacía se representa en Python con la constante '' (dos comillas simples sin ningún caracter entre ellas) o bien con la constante "" (dos comillas dobles sin ningún caracter entre ellas)):

```
cad = 'Hola'

if cad != '':
    print('La cadena es:', cad)
else:
    print('La cadena está vacía')
```

Pero como queda dicho, la cadena vacía será interpretada por Python como un *False* en un contexto lógico, por lo cual el script podría replantearse así, mediante una *condición implícita*:

```
cad = 'Hola'

if cad:
    print('La cadena es:', cad)
else:
    print('La cadena está vacía')
```

Como dijimos, el uso de conectores lógicos *and*, *or* y *not* convierte la expresión analizada en una *expresión lógica*, sean cuales sean los tipos de las variables sobre las que se aplican

estos conectores, y sin importar el contexto. Un caso trivial pero muy ilustrativo ayuda a comenzar a entender la idea:

```
a = 10
b = not a
print('b: ', b)
```

En el pequeño script anterior la variable *a* contiene un valor numérico (el 10) y por lo tanto, *no es booleana*. Sin embargo, en la segunda línea se aplica sobre ella el operador booleano *not*, y se asigna en la variable *b* el *resultado de la negación lógica* de *a*. Podría parecer que esto no tiene sentido, pero en Python lo tiene: la aparición del *operador booleano not* hace que la expresión *not a* sea considerada como lógica, y por lo tanto el valor de la variable *a* será interpretado en forma *booleana* y ***no*** en forma *numérica*. Dado que el valor de *a es 10*, en un contexto booleano como es este, Python lo interpretará como un *True* (ya que es diferente de cero) y al negar ese valor con *not*, *el resultado asignado en b será un False*. La ejecución de este script producirá la siguiente salida en consola estándar:

```
b:  False
```

Por cierto, note que ***no es lo mismo*** hacer *b = not a* que hacer b = -a. La primera asignación, como vimos, toma el valor de *a* en forma booleana y asigna en *b* un *False* si *a* valía 10 o cualquier otro número diferente de cero. Pero la segunda asignación toma el valor de *a* y cambia su signo, por lo que si *a* valía 10, el valor asignado en *b* será un -10.

Y a partir de todo lo expuesto, prácticamente cualquier expresión que incluya operadores booleanos será válida en una instrucción condicional o en cualquier otra que espere una expresión lógica: los operadores booleanos *and*, *or* y *not* convertirán esa expresión en una expresión lógica aplicando los criterios de transformación de valores no booleanos en booleanos según vimos al inicio de esta sección. A modo de ejemplo de cierre, analicemos este modelo en el cual se asigna en la variable *r* el resultado (*False*) de una expresión lógica basada en las ideas que hemos expuesto.

```
a, b, c, d, e = 10, 4, 5, 5, 3

# Asignación en r del valor False...
r = a and b > d or c != e and not (d or b)
# Desarrollo paso a paso: reemplazo de las variables por sus valores...
r = 10 and 4 > 5 or 5 != 3 and not (5 or 4)

# Desarrollo paso a paso: se aplican los operadores relacionales...
r = 10 and 4 > 5 or 5 != 3 and not (5 or 4)
r = 10 and False or True and not (5 or 4)

# Desarrollo paso a paso: se aplican los conectores lógicos...
# ...primero los que vengan entre paréntesis
r = 10 and False or True and not (5 or 4)
r = 10 and False or True and not True

# Desarrollo paso a paso: se aplican los conectores lógicos...
# ...luego los not...
r = 10 and False or True and not True
r = 10 and False or True and False

# Desarrollo paso a paso: se aplican los conectores lógicos...
# ...luego los and...
r = 10 and False or True and False
```

```
r = False or False

# Desarrollo paso a paso: se aplican los conectores lógicos...
# ...y finalmente los or...
r = False or False
r = False
```

Puede verse que el manejo de expresiones en contextos booleanos requiere que el programador preste atención a lo que hace y comprenda el efecto de las trasformaciones de valores que efectúa Python para la conversión lógica. Esta forma de trabajo del lenguaje puede parecer oscura y hasta contraria al espíritu de Python de favorecer la claridad y la legibilidad del código fuente, pero el hecho es que las expresiones booleanas han sido implementadas así siguiendo el concepto de manejo de valores lógicos del lenguaje C y del C++. Aquellos programadores que migran a Python desde C o C++ asumen estos manejos como naturales, mientras que aquellos que llegan desde otros lenguajes más estructurados en la gestión de tipos (como Java o Pascal) deberán hacer un esfuerzo para incorporar estos elementos. Y por cierto, aquellos que son programadores absolutamente novatos, deberán hacer el esfuerzo en cualquier caso y de todos modos… ☺.

7.] Breve discusión: el Álgebra de *Boole* y las Leyes de *De Morgan*.

El estudio y sistematización de las operaciones lógicas como *and*, *or* y *not*, forma parte de la rama de la Matemática denominada *Álgebra de Boole*, la que toma su nombre del matemático inglés *George Boole* que fue quien sentó sus bases en 1847. El planteo inicial fue simplemente el de intentar generalizar las técnicas del álgebra para el tratamiento de la lógica proposicional, pero a lo largo del tiempo evolucionó y hoy se utiliza ampliamente en el campo de la electrónica, específicamente en el diseño de circuitos.

Como hemos visto en toda esta Ficha, los elementos fundamentales del *Álgebra de Boole* se utilizan también en el área de la programación de computadoras, ya que está contenida en el planteo de las expresiones lógicas que se usan para el control de instrucciones tales como las condicionales y las repetitivas, entre otras. El planteo y estudio de las ya citadas *tablas de verdad*, y la demostración de teoremas y propiedades relativas a la aplicación de los operadores booleanos, forman parte de esta disciplina.

Hemos visto también que el planteo de expresiones lógicas que combinen operadores booleanos, operadores relacionales, operadores aritméticos y variables y constantes de distintos tipos, puede dar lugar a fórmulas de aspecto realmente intimidante que podrían poner a cualquier programador (experto o inexperto) en serios problemas para interpretarlas correctamente. Es común encontrarse con casos de programas en fase de desarrollo que muestran fallas más o menos graves debidas al planteo incorrecto de una o más expresiones lógicas. Por lo tanto, uno de los tantos trabajos que un buen programador debe saber llevar a cabo, es el de simplificar expresiones lógicas, de forma que la expresión simplificada sea más clara, más breve, menos extensa y más simple de entender (lo cual lleva a que finalmente sea más simple de modificar o ajustar si fuese el caso).

En esa dirección apuntan algunas reglas y teoremas del Álgebra de Boole. Dos de esas reglas son muy conocidas y aplicadas en la simplificación de expresiones lógicas y en el diseño de circuitos, y se conocen como *Leyes de De Morgan* [3], en honor al matemático británico (nacido en la India) *Augustus De Morgan* que era contemporáneo de *George Boole*, al punto de que los trabajos de ambos se complementaron entre sí.

Las dos Leyes de *De Morgan* están orientadas a la forma de simplificar la *negación de una conjunción* (o sea, la negación de un *and*) y a la forma de simplificar la *negación de una disyunción* (o sea, la negación de un *or*). Si suponemos que *p* y *q* son dos proposiciones lógicas, usando sintaxis de Python el planteo de ambas reglas es el que sigue:

1.] Negación de un and: `not(p and q) ⇔ not p or not q`
2.] Negación de un or: `not(p or q) ⇔ not p and not q`

En lenguaje informal, la primera regla se expresa como: *la negación de un and es igual al or de las negaciones de ambas proposiciones*. Y la segunda se expresa como: *la negación de un or es igual al and de las negaciones de ambas proposiciones*.

Ambas reglas pueden demostrarse por diversas vías. Una forma de hacerlo, es simplemente exponer las tablas de verdad y comprobar que para cada regla, las tablas de las expresiones a la izquierda y a la derecha son equivalentes, lo cual hicimos en la figura que sigue:

Figura 11: Tablas de verdad de las Leyes de De Morgan.

Tablas de verdad de las *Leyes de De Morgan*					
p	**q**	**not(p and q)**	**not p or not q**	**not(p or q)**	**not p and not q**
True	True	False	False	False	False
True	False	True	True	False	False
False	True	True	True	False	False
False	False	True	True	True	True

1.] Negación de un and	**2.]** Negación de un or
not(p and q) ⇔ not p or not q	not(p or q) ⇔ not p and not q

Como se ve, las columnas 3 y 4 son equivalentes entre sí y justamente esas columnas corresponden a las expresiones de la regla 1 (*negación de un and*). Y a su vez también las columnas 5 y 6 son equivalentes entre sí, correspondiendo ambas a las expresiones de la regla 2 (*negación de un or*).

La utilidad práctica de estas dos reglas en programación se hace evidente por medio de un ejemplo: Sin importar los valores que asuman las variables, supongamos una expresión lógica como la que sigue:

```
r = not(a < b and c != b and d >= e and c <= 0)
```

En principio, no hay mayor problema en dejarla como está, pero el planteo en base a un *not* inicial suele producir confusión, ya que el programador debe pensar la expresión completa sin el negador, y finalmente negar la salida para terminar de verla. Dado que esta expresión es la *negación de un encadenamiento de proposiciones con el conector and*, se puede aplicar la regla 1 de *De Morgan*, e intentar eliminar el negador inicial. En el consabido proceso "paso a paso" que sigue, mostramos como se va reduciendo la expresión, hasta llegar a la versión final, más simple (y siempre recuerde que tanto el *and* como el *or* son cortocircuitados, por lo que en realidad, alguna de las proposiciones siguientes podría no llegar a chequearse cuando sea ejecutada):

```
# Expresión inicial...
r = not(a < b and c != b and d >= e and c <= 0)

# Aplicar De Morgan - regla 1... (no parece que ganemos nada...)
r = not a < b or not c != b or not d >= e or not c <= 0

# Reducir los not individuales... y entonces sí... ☺
# Expresión final, equivalente a la inicial, pero más simple...
r = a >= b or c == b or d < e or c > 0
```

La expresión finalmente asignada en *r* está compuesta sólo por proposiciones conectadas por el operador *or*: todas las proposiciones individualmente negadas fueron reemplazadas por sus equivalentes en lógica directa (sin los negadores) y los *and* se reemplazaron por los *or*. El truco para lograr la simplificación final, consiste en el reemplazo de cada proposición individual negada por su equivalente directa, de forma de eliminar el negador. Por ejemplo, la expresión *not a < b* es equivalente a la expresión *a >= b* (tienen la misma tabla de verdad), lo cual es simple de deducir: preguntar si *no es cierto que a es menor que b*, es lo mismo que preguntar si *a es mayor o igual que b*. Asegúrese de entender por su propia deducción el resto de los reemplazos que hemos indicado en el ejemplo anterior.

Bibliografía

[1] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: https://docs.python.org/3/.

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

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

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