### 4.1.7. Una estrategia de diseño: refinamientos sucesivos

Es lógico que cuando estés aprendiendo a programar te cueste gran esfuerzo construir mentalmente un programa tan complicado, pero posiblemente sea porque sigues una aproximación equivocada: no debes intentar construir mentalmente *todo* el programa de una vez. Es recomendable que sigas una estrategia similar a la que hemos usado al desarrollar los programas de ejemplo:

1. Primero haz una versión sobre papel que resuelva el problema de forma directa y, posiblemente, un tanto tosca. Una buena estrategia es plantearse uno mismo el problema con unos datos concretos, resolverlo a mano con lápiz y papel y hacer un esquema con el orden de las operaciones realizadas y las decisiones tomadas. Tu primer programa puede pedir los datos de entrada (con *raw_input*), hacer los cálculos del mismo modo que tú los hiciste sobre el papel (utilizando variables para los resultados intermedios, si fuera menester) y mostrar finalmente el resultado del cálculo (con **print**).

2. Analiza tu programa y considera si realmente resuelve el problema planteado: ¿es posible que se cometan errores en tiempo de ejecución?, ¿hay configuraciones de los datos que son especiales y, para ellas, el cálculo debe ser diferente?

3. Cada vez que te plantees una de estas preguntas y tengas una respuesta, modifica el programa en consecuencia. No hagas más de un cambio cada vez.

4. Si el programa ya funciona correctamente *para todas las entradas posibles* y eres capaz de anticiparte a los posibles errores de ejecución, ¡enhorabuena!, ya casi has terminado. En caso contrario, vuelve al paso 2.

5. Ahora que ya estás «seguro» de que todo funciona correctamente, teclea el programa en el ordenador y efectúa el mayor número de pruebas posibles, comprobando cuidadosamente que el resultado calculado es correcto. Presta especial atención a configuraciones extremas o singulares de los datos (los que pueden provocar divisiones por cero o valores muy grandes, o muy pequeños, o negativos, etc.). Si el programa calcula algo diferente de lo esperado o si se aborta la ejecución del programa por los errores detectados, vuelve al paso 2.

Nadie es capaz de hacer un programa suficientemente largo de una sentada, empezando a escribir por la primera línea y acabando por la última, una tras otra, del mismo

modo que nadie escribe una novela o una sinfonía de una sentada[1]. Lo normal es empezar con un borrador e ir refinándolo, mejorándolo poco a poco.

Un error frecuente es tratar de diseñar el programa directamente sobre el ordenador, escribiéndolo a bote pronto. Es más, hay estudiantes que se atreven a empezar con la escritura de un programa sin haber entendido bien el enunciado del problema que se pretende resolver. Es fácil pillarlos en falta: no saben resolver a mano un caso particular del problema. Una buena práctica, pues, es solucionar manualmente unos pocos ejemplos concretos para estar seguros de que conocemos bien lo que se nos pide y cómo calcularlo. Una vez superada esta fase, estarás en condiciones de elaborar un borrador con los pasos que has de seguir. Créenos: es mejor que pienses un rato y diseñes un borrador del algoritmo sobre papel. Cuando estés muy seguro de la validez del algoritmo, impleméntalo en Python y pruébalo sobre el ordenador. Las pruebas con el ordenador te ayudarán a encontrar errores.

Ciertamente es posible utilizar el ordenador directamente, como si fuera el papel. Nada impide que el primer borrador lo hagas ya en pantalla, pero, si lo haces, verás que:

- Los detalles del lenguaje de programación interferirán en el diseño del algoritmo («¿he de poner dos puntos al final de la línea?», «¿uso marcas de formato para imprimir los resultados?», etc.): cuando piensas en el método de resolución del problema es mejor hacerlo con cierto grado de abstracción, sin tener en cuenta todas las particularidades de la notación.

- Si ya has tecleado un programa y sigue una aproximación incorrecta, te resultará más molesto prescindir de él que si no lo has tecleado aún. Esta molestia conduce a la tentación de ir poniendo parches a tu deficiente programa para ver si se puede arreglar algo. El resultado será, muy probablemente, un programa ilegible, pésimamente organizado... y erróneo. Te costará la mitad de tiempo empezar de cero, pero esta vez haciendo bien las cosas: pensando antes de escribir nada.

### 4.1.8. Un nuevo refinamiento del programa de ejemplo

Parece que nuestro programa ya funciona correctamente. Probemos a resolver esta ecuación:

$$x^2 + 2x + 3 = 0$$

```
Valor de a: 1
Valor de b: 2
Valor de c: 3
Traceback (innermost last):
  File 'segundo_grado.py', line 8, in ?
    x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
ValueError: math domain error
```

¡Nuevamente un error! El mensaje de error es diferente de los anteriores y es un «error de dominio matemático».

El problema es que estamos intentando calcular la raíz cuadrada de un número negativo en la línea 8. El resultado es un número complejo, pero el módulo *math* no «sabe» de números complejos, así que *sqrt* falla y se produce un error. También en la línea 9 se tiene que calcular la raíz cuadrada de un número negativo, pero como la línea 8 se ejecuta en primer lugar, es ahí donde se produce el error y se aborta la ejecución. La línea 9 no llega a ejecutarse.

[1] Aunque hay excepciones: cuenta la leyenda que Mozart escribía sus obras de principio a fin, sin volver atrás para efectuar correcciones.

### *El síndrome «a mí nunca se me hubiera ocurrido esto»*

Programar es una actividad que requiere un gran esfuerzo intelectual, no cabe duda, pero sobre todo, ahora que empiezas, es una actividad radicalmente diferente de cualquier otra para la que te vienes preparando desde la enseñanza primaria. Llevas muchos años aprendiendo lengua, matemáticas, física, etc., pero nunca antes habías programado. Los programas que hemos visto en este capítulo te deben parecer muy complicados, cuando no lo son tanto.

La reacción de muchos estudiantes al ver la solución que da el profesor o el libro de texto a un problema de programación es decirse «a mí nunca se me hubiera ocurrido esto». Debes tener en cuenta dos factores:

- La solución final muchas veces esconde la línea de razonamiento que permitió llegar a ese programa concreto. Nadie construye los programas de golpe: por regla general se hacen siguiendo refinamientos sucesivos a partir de una primera versión bastante tosca.
- La solución que se te presenta sigue la línea de razonamiento de una persona concreta: el profesor. Puede que tu línea de razonamiento sea diferente y, sin embargo, igualmente válida (¡o incluso mejor!), así que tu programa puede no parecerse en nada al suyo y, a la vez, ser correcto. No obstante, te conviene estudiar la solución que te propone el profesor: la lectura de programas escritos por otras personas es un buen método de aprendizaje y, probablemente, la solución que te ofrece resuelva cuestiones en las que no habías reparado (aunque sólo sea porque el profesor lleva más años que tú en esto de programar).

Podemos controlar este error asegurándonos de que el término $b^2-4ac$ (que recibe el nombre de «discriminante») sea mayor o igual que cero *antes* de calcular la raíz cuadrada:

segundo_grado_18.py — segundo_grado.py

```
from math import sqrt

a = float(raw_input('Valor de a: '))
b = float(raw_input('Valor de b: '))
c = float(raw_input('Valor de c: '))

if a != 0:
    if b**2 - 4*a*c >= 0:
        x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
        x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
        print 'Soluciones de la ecuación: x1=%4.3f y x2=%4.3f' % (x1, x2)
    else:
        print 'No hay soluciones reales.'
else:
    if b != 0:
        x = -c / b
        print 'Solución de la ecuación: x=%4.3f' % x
    else:
        if c !=0:
            print 'La ecuación no tiene solución.'
        else:
            print 'La ecuación tiene infinitas soluciones.'
```

EJERCICIOS

▶ 76 Un programador ha intentado solucionar el problema del discriminante negativo con un programa que empieza así:

segundo_grado.py

```
from math import sqrt

a = float(raw_input('Valor de a: '))
b = float(raw_input('Valor de b: '))
c = float(raw_input('Valor de c: '))

if a != 0:
    if sqrt(b**2 - 4*a*c) >= 0:
        x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
        x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
    ...
```

Evidentemente, el programa es incorrecto y te sorprenderá saber que algunos estudiantes proponen soluciones similares a ésta. El problema estriba en el posible valor negativo *del argumento* de *sqrt*, así que la comparación es incorrecta, pues pregunta por el signo *de la raíz* de dicho argumento. Pero el programa no llega siquiera a dar solución alguna (bien o mal calculada) cuando lo ejecutamos con, por ejemplo, $a = 4$, $b = 2$ y $c = 4$. ¿Qué sale por pantalla en ese caso? ¿Por qué?

Dado que sólo hemos usado sentencias condicionales para controlar los errores, es posible que te hayas llevado la impresión de que ésta es su única utilidad. En absoluto. Vamos a utilizar una sentencia condicional con otro propósito. Mira qué ocurre cuando tratamos de resolver la ecuación $x^2 - 2x + 1 = 0$:

```
Valor de a: 1
Valor de b: -2
Valor de c: 1
Soluciones de la ecuación: x1=1.000 y x2=1.000
```

Las dos soluciones son iguales, y queda un tanto extraño que el programa muestre el mismo valor dos veces. Hagamos que, cuando las dos soluciones sean iguales, sólo se muestre una de ellas:

segundo_grado_19.py — segundo_grado.py

```
from math import sqrt

a = float(raw_input('Valor de a: '))
b = float(raw_input('Valor de b: '))
c = float(raw_input('Valor de c: '))

if a != 0:
    if b**2 - 4*a*c >= 0:
        x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
        x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
        if x1 == x2:
            print 'Solución de la ecuación: x=%4.3f' % x1
        else:
            print 'Soluciones de la ecuación: x1=%4.3f y x2=%4.3f' % (x1, x2)
    else:
        print 'No hay soluciones reales.'
else:
    if b != 0:
        x = -c / b
        print 'Solución de la ecuación: x=%4.3f' % x
    else:
        if c !=0:
            print 'La ecuación no tiene solución.'
```

```
24        else:
25            print 'La ecuación tiene infinitas soluciones.'
```

### *Optimización*

Podemos plantear un nuevo refinamiento que tiene por objeto hacer un programa más rápido, más eficiente. Fíjate que en las líneas 8, 9 y 10 del último programa se calcula cada vez la expresión *b***2 - 4**a***c*. ¿Para qué hacer tres veces un mismo cálculo? Si las tres veces el resultado va a ser el mismo, ¿no es una pérdida de tiempo repetir el cálculo? Podemos efectuar una sola vez el cálculo y guardar el resultado en una variable.

segundo_grado_20.py — segundo_grado.py

```
1  from math import sqrt
2
3  a = float(raw_input('Valor de a: '))
4  b = float(raw_input('Valor de b: '))
5  c = float(raw_input('Valor de c: '))
6
7  if a != 0:
8      discriminante = b**2 - 4*a*c
9      if discriminante >= 0:
10         x1 = (-b + sqrt(discriminante)) / (2 * a)
11         x2 = (-b - sqrt(discriminante)) / (2 * a)
12         if x1 == x2:
13             print 'Solución de la ecuación: x=%4.3f' % x1
14         else:
15             print 'Soluciones de la ecuación:',
16             print 'x1=%4.3f y x2=%4.3f' % (x1, x2)
17     else:
18         print 'No hay soluciones reales.'
19 else:
20     if b != 0:
21         x = -c / b
22         print 'Solución de la ecuación: x=%4.3f' % x
23     else:
24         if c !=0:
25             print 'La ecuación no tiene solución.'
26         else:
27             print 'La ecuación tiene infinitas soluciones.'
```

Modificar un programa que funciona correctamente para hacer que funcione más eficientemente es *optimizar* el programa. No te obsesiones con la optimización de tus programas. Ahora estás aprendiendo a programar. Asegúrate de que tus programas funcionan correctamente. Ya habrá tiempo para optimizar más adelante.

## 4.1.9. Otro ejemplo: máximo de una serie de números

Ahora que sabemos utilizar sentencias condicionales, vamos con un problema sencillo, pero que es todo un clásico en el aprendizaje de la programación: el cálculo del máximo de una serie de números.

Empezaremos por pedirle al usuario dos números enteros y le mostraremos por pantalla cuál es el mayor de los dos.

Estudia esta solución, a ver qué te parece:

maximo_5.py — maximo.py

```
a = int(raw_input('Dame el primer número: '))
b = int(raw_input('Dame el segundo número: '))

if a > b:
    maximo = a
else:
    maximo = b

print 'El máximo es', maximo
```

EJERCICIOS

▶ 77 ¿Qué líneas del último programa se ejecutan y qué resultado aparece por pantalla en cada uno de estos casos?

a) $a = 2$ y $b = 3$. b) $a = 3$ y $b = 2$. c) $a = -2$ y $b = 0$. d) $a = 1$ y $b = 1$.

Analiza con cuidado el último caso. Observa que los dos números son iguales. ¿Cuál es, pues, el máximo? ¿Es correcto el resultado del programa?

▶ 78 Un aprendiz de programador ha diseñado este otro programa para calcular el máximo de dos números:

maximo_6.py — maximo.py

```
a = int(raw_input('Dame el primer número: '))
b = int(raw_input('Dame el segundo número: '))

if a > b:
    maximo = a
if b > a:
    maximo = b

print 'El máximo es', maximo
```

¿Es correcto? ¿Qué pasa si introducimos dos números iguales?

Vamos con un problema más complicado: el cálculo del máximo de tres números enteros (que llamaremos $a$, $b$ y $c$). He aquí una estrategia posible:

1. Me pregunto si $a$ es mayor que $b$ y, si es así, de momento $a$ es candidato a ser el mayor, pero no sé si lo será definitivamente hasta compararlo con $c$. Me pregunto, pues, si $a$ es mayor que $c$.
   a) Si $a$ también es mayor que $c$, está claro que $a$ es el mayor de los tres.
   b) Y si no, $c$ es el mayor de los tres.
2. Pero si no es así, es decir, si $a$ es menor o igual que $b$, el número $b$ es, de momento, mi candidato a número mayor. Falta compararlo con $c$.
   a) Si también es mayor que $c$, entonces $b$ es el mayor.
   b) Y si no, entonces $c$ es el mayor.

Ahora que hemos diseñado el procedimiento, construyamos un programa Python que implemente ese algoritmo:

maximo_de_tres_3.py — maximo_de_tres.py

```
a = int(raw_input('Dame el primer número: '))
b = int(raw_input('Dame el segundo número: '))
c = int(raw_input('Dame el tercer número: '))
```

```
4
5  if a > b:
6      if a > c:
7          maximo = a
8      else:
9          maximo = c
10 else:
11     if b > c:
12         maximo = b
13     else:
14         maximo = c
15
16 print 'El máximo es', maximo
```

EJERCICIOS

▶ 79 ¿Qué secuencia de líneas de este último programa se ejecutará en cada uno de estos casos?

a) $a = 2$, $b = 3$ y $c = 4$. b) $a = 3$, $b = 2$ y $c = 4$. c) $a = 1$, $b = 1$ y $c = 1$.

Ayúdate con el modo de depuración de PythonG.

Puede que la solución que hemos propuesto te parezca extraña y que tú hayas diseñado un programa muy diferente. Es normal. No existe un único programa para solucionar un problema determinado y cada persona desarrolla un estilo propio en el diseño de los programas. Si el que se propone como solución no es igual al tuyo, el tuyo no tiene por qué ser erróneo; quizá sólo sea distinto. Por ejemplo, este otro programa también calcula el máximo de tres números, y es muy diferente del que hemos propuesto antes:

maximo_de_tres_4.py — maximo_de_tres.py

```
1  a = int(raw_input('Dame el primer número: '))
2  b = int(raw_input('Dame el segundo número: '))
3  c = int(raw_input('Dame el tercer número: '))
4
5  candidato = a
6  if b > candidato:
7      candidato = b
8  if c > candidato:
9      candidato = c
10 maximo = candidato
11
12 print 'El máximo es', maximo
```

EJERCICIOS

▶ 80 Diseña un programa que calcule el máximo de 5 números enteros. Si sigues una estrategia similar a la de la primera solución propuesta para el problema del máximo de 3 números, tendrás problemas. Intenta resolverlo como en el último programa de ejemplo, es decir con un «candidato a valor máximo» que se va actualizando al compararse con cada número.

▶ 81 Diseña un programa que calcule la menor de cinco palabras dadas; es decir, la primera palabra de las cinco en orden alfabético. Aceptaremos que las mayúsculas son «alfabéticamente» menores que las minúsculas, de acuerdo con la tabla ASCII.

▶ 82 Diseña un programa que calcule la menor de cinco palabras dadas; es decir, la primera palabra de las cinco en orden alfabético. *No* aceptaremos que las mayúsculas sean «alfabéticamente» menores que las minúsculas. O sea, `'pepita'` es menor que `'Pepito'`.

▶ 83 Diseña un programa que, dados cinco números enteros, determine cuál de los cuatro últimos números es más cercano al primero. (Por ejemplo, si el usuario introduce los números 2, 6, 4, 1 y 10, el programa responderá que el número más cercano al 2 es el 1.)

▶ 84 Diseña un programa que, dados cinco puntos en el plano, determine cuál de los cuatro últimos puntos es más cercano al primero. Un punto se representará con dos variables: una para la abcisa y otra para la ordenada. La distancia entre dos puntos $(x_1, y_1)$ y $(x_2, y2)$ es $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Las comparaciones pueden incluir cualquier expresión cuyo resultado sea interpretable en términos de cierto o falso. Podemos incluir, pues, expresiones lógicas tan complicadas como deseemos. Fíjate en el siguiente programa, que sigue una aproximación diferente para resolver el problema del cálculo del máximo de tres números:

maximo_de_tres.py

```
a = int(raw_input('Dame el primer número: '))
b = int(raw_input('Dame el segundo número: '))
c = int(raw_input('Dame el tercer número: '))

if a >= b and a >= c:
    maximo = a
if b >= a and b >= c:
    maximo = b
if c >= a and c >= b:
    maximo = c
print 'El máximo es', maximo
```

La expresión `a >= b and a >= c` por ejemplo, se lee «*a* es mayor o igual que *b* y *a* es mayor o igual que *c*».

## EJERCICIOS

▶ 85 Indica en cada uno de los siguientes programas qué valores o rangos de valores provocan la aparición de los distintos mensajes:

a) aparcar.py

```
dia = int(raw_input('Dime qué día es hoy: '))

if 0 < dia <= 15:
    print 'Puedes aparcar en el lado izquierdo de la calle'
else:
    if 15 < dia < 32:
        print 'Puedes aparcar en el lado derecho de la calle'
    else:
        print 'Ningún mes tiene %d días.' % dia
```

b) estaciones.py

```
mes = int(raw_input('Dame un mes: '))

if 1 <= mes <= 3:
    print 'Invierno.'
else:
    if mes == 4 or mes == 5 or mes == 6:
        print 'Primavera.'
    else:
        if not (mes < 7 or 9 < mes):
            print 'Verano.'
        else:
            if not (mes != 10 and mes != 11 and mes != 12):
```

```
13            print 'Otoño.'
14          else:
15            print 'Ningún año tiene %d meses.' % mes
```

c) identificador.py

```
1  car = raw_input('Dame un carácter: ')
2
3  if 'a' <= car.lower() <= 'z' or car == '_':
4    print 'Este carácter es válido en un identificador en Python.'
5  else:
6    if not (car < '0' or '9' < car):
7      print 'Un dígito es válido en un identificador en Python,',
8      print 'siempre que no sea el primer carácter.'
9    else:
10     print 'Carácter no válido para formar un identificador en Python.'
```

d) bisiesto.py

```
1  anyo = int(raw_input('Dame un año: '))
2
3  if anyo % 4 == 0 and (anyo % 100 != 0 or anyo % 400 == 0):
4    print 'El año %d es bisiesto.' % anyo
5  else:
6    print 'El año %d no es bisiesto.' % anyo
```

▶ 86 La fórmula $C' = C \cdot (1 + x/100)^n$ nos permite obtener el capital final que lograremos a partir de un capital inicial ($C$), una tasa de interés anual ($x$) en tanto por cien y un número de años ($n$). Si lo que nos interesa conocer es el número de años $n$ que tardaremos en lograr un capital final $C'$ partiendo de un capital inicial $C$ a una tasa de interés anual $x$, podemos despejar $n$ en la fórmula del ejercicio 67 de la siguiente manera:

$$n = \frac{\log(C') - \log(C)}{\log(1 + x/100)}$$

Diseña un programa Python que obtenga el número de años que se tarda en conseguir un capital final dado a partir de un capital inicial y una tasa de interés anual también dados. El programa debe tener en cuenta cuándo se puede realizar el cálculo y cuándo no en función del valor de la tasa de interés (para evitar una división por cero, el cálculo de logaritmos de valores negativos, etc)... con una excepción: si $C$ y $C'$ son iguales, el número de años es 0 independientemente de la tasa de interés (incluso de la que provocaría un error de división por cero).

(Ejemplos: Para obtener 11 000 € por una inversión de 10 000 € al 5% anual es necesario esperar 1.9535 años. Obtener 11 000 € por una inversión de 10 000 € al 0% anual es imposible. Para obtener 10 000 € con una inversión de 10 000 € no hay que esperar nada, sea cual sea el interés.)

▶ 87 Diseña un programa que, dado un número real que debe representar la calificación numérica de un examen, proporcione la calificación cualitativa correspondiente al número dado. La calificación cualitativa será una de las siguientes: «Suspenso» (nota menor que 5), «Aprobado» (nota mayor o igual que 5, pero menor que 7), «Notable» (nota mayor o igual que 7, pero menor que 8.5), «Sobresaliente» (nota mayor o igual que 8.5, pero menor que 10), «Matrícula de Honor» (nota 10).

▶ 88 Diseña un programa que, dado un carácter cualquiera, lo identifique como vocal minúscula, vocal mayúscula, consonante minúscula, consonante mayúscula u otro tipo de carácter.

### *De Morgan*

Las expresiones lógicas pueden resultar complicadas, pero es que los programas hacen, en ocasiones, comprobaciones complicadas. Tal vez las más difíciles de entender son las que comportan algún tipo de negación, pues generalmente nos resulta más difícil razonar en sentido negativo que afirmativo. A los que empiezan a programar les lían muy frecuentemente las negaciones combinadas con **or** o **and**. Veamos algún ejemplo «de juguete». Supón que para aprobar una asignatura hay que obtener más de un 5 en dos exámenes parciales, y que la nota de cada uno de ellos está disponible en las variables *parcial*1 y *parcial*2, respectivamente. Estas líneas de programa muestran el mensaje «Has suspendido.» cuando no has obtenido al menos un 5 en los dos exámenes:

```
if not (parcial1 >= 5.0 and parcial2 >= 5.0):
    print 'Has␣suspendido.'
```

Lee bien la condición: «si no es cierto que has sacado al menos un 5 *en ambos* (por eso el **and**) parciales...». Ahora fíjate en este otro fragmento:

```
if not parcial1 >= 5.0 or not parcial2 >= 5.0:
    print 'Has␣suspendido.'
```

Leámoslo: «si no has sacado al menos un cinco *en uno u otro* (por eso el **or**) parcial...». O sea, los dos fragmentos son equivalentes: uno usa un **not** que se aplica al resultado de una operación **and**; el otro usa dos operadores **not** cuyos resultados se combinan con un operador **or**. Y sin embargo, dicen la misma cosa. Los lógicos utilizan una notación especial para representar esta equivalencia:

$$\begin{aligned} \neg(p \wedge q) &\longleftrightarrow \neg p \vee \neg q, \\ \neg(p \vee q) &\longleftrightarrow \neg p \wedge \neg q. \end{aligned}$$

(Los lógicos usan '$\neg$' para **not**, '$\wedge$' para **and** y '$\vee$' para **or**.) Estas relaciones se deben al matemático De Morgan, y por ese nombre se las conoce. Si es la primera vez que las ves, te resultarán chocantes, pero si piensas un poco, verás que son de sentido común.

Hemos observado que los estudiantes cometéis errores cuando hay que expresar la condición contraria a una como «*a* **and** *b*». Muchos escribís «**not** *a* **and not** *b*» y está mal. La negación correcta sería «**not** (*a* **and** *b*)» o, por De Morgan, «**not** *a* **or not** *b*». ¿Cuál sería, por cierto, la negación de «*a* **or not** *b*»?