Capítulo 3-3
Modelo de objetos

1. Todo es un objeto

1.1 Principios

1.1.1 Qué sentido dar a «objeto»

Python es un lenguaje que utiliza varios paradigmas y, entre ellos, el paradigma orientado a objetos. Este se elaboró durante los años 1970 y es, ante todo, un concepto. Un objeto representa:

– un objeto físico:
  – parcela de terreno, bien inmueble, apartamento, propietario, inquilino...;
  – coche, piezas de un coche, conductor, pasajero...;
  – biblioteca, libro, página de un libro...;
  – dispositivo de hardware, robot...;
– un objeto informático:
  – archivo (imagen, documento de texto, sonido, vídeo...);
  – servicio (servidor, cliente, sitio de Internet, servicio web...);
  – un flujo de datos, pool de conexiones...;
– un concepto:
  – portador de alguna noción que pueda compartir;
  – secuenciador, ordenador, analizador de datos...

Observación

Aprovechamos para llamar la atención del lector sobre ciertos aspectos cognitivos: a partir del momento en que se modelizan personas como objetos, es fácil olvidar que detrás del código hay seres vivos que piensan, y cuya libertad puede ser limitada por las elecciones que se hacen sobre la manera de modelizar, la cantidad o la calidad de las informaciones que se decide tratar o conservar.

Aquí vemos un ejemplo muy simple y concreto: un gestor de horarios que solo permite crear franjas cada media hora puede tener una repercusión real sobre el funcionamiento de un servicio que necesita una granularidad más fina. Por lo tanto, influye de manera muy negativa en los usuarios que, antes de la llegada del software, tenían horarios que permitían más libertad.

Además, a partir del momento en que almacena datos de sus usuarios, es responsable de su conservación y seguridad; por eso tiene que asegurarse de que no se puedan filtrar, robar o incluso corromper.

Hay organismos que permiten dar una visión correcta de las reglas básicas que se deben seguir para tratar datos de este tipo, y también legislaciones, como el RGPD, que a su vez determinan algunos principios útiles.

Para modelizar como objeto, es necesario seguir algunos principios.
Uno de los principios es la encapsulación de datos. Esto significa que cada objeto posee en su seno no solo los datos que lo describen y que contiene (bajo la forma de atributos), sino también el conjunto de métodos necesarios para gestionar sus propios datos (modificación, actualización, compartición...).
El desarrollo orientado a objetos consiste, simplemente, en crear un conjunto de objetos que representa de la mejor forma posible aquello que modelan y en gestionar sus interacciones.
Cada funcionalidad se modela, de este modo, bajo la forma de interacciones entre objetos. De su correcto modelado y de la naturaleza de sus interacciones dependen la calidad del programa y también su estabilidad y mantenibilidad.
El paradigma orientado a objetos define, entonces, otros mecanismos para dar respuesta a las distintas problemáticas que se le plantean al desarrollador: polimorfismo, interfaces, herencia, sobrecarga de métodos, sobrecarga de operadores…
Es aquí donde se diferencian los lenguajes entre sí, pues cada uno propone soluciones que le son propias utilizando o no ciertos mecanismos del lenguaje orientado a objetos y de forma más o menos fiel a su espíritu.

1.1.2 Adaptación de la teoría de objetos en Python

En lenguajes como PHP, por ejemplo, se agrega una **semántica de objetos** que permite a los desarrolladores escribir de forma similar a un lenguaje orientado a objetos. Esto se realiza en dos etapas: la posibilidad de declarar clases (con interfaces y herencia simple) y la posibilidad de crear instancias de estas clases y acceder a los atributos de los métodos. Pero no es más que una semántica de objetos, puesto que detrás se trata en realidad de tablas (que contienen los atributos) que se asocian a una lista de métodos que pueden aplicarse al objeto.

La implementación está, por tanto, muy lejos de un paradigma orientado a objetos, aunque la semántica esté presente y sea suficiente para este lenguaje.

En los lenguajes orientados a objetos, como Java, el paradigma orientado a objetos está en el núcleo del lenguaje y, por tanto, de la gramática. Se han realizado adaptaciones del concepto para amoldarse a distintos escenarios técnicos o a una filosofía propia del lenguaje. No se dispone de herencia múltiple, y el concepto de interfaz se ha transformado en su totalidad para ofrecer una alternativa. Como no existen más que objetos, es necesario pasar por el proceso de bootstrap y las arquitecturas se han vuelto difíciles o restrictivas debido a limitaciones técnicas que debían respetarse.

C++ también propone sus propias adaptaciones e innovaciones. El modelo orientado a objetos que ofrece es la referencia absoluta de un lenguaje de bajo nivel estáticamente tipado.

Estas son las características esenciales que diferencian estos lenguajes del lenguaje de programación Python y que hacen que el modelo de objetos de Python sea, necesariamente, muy diferente.

Pero, además de ser diferente, el lenguaje ha tratado de aprovechar sus cualidades básicas que lo diferencian de otros lenguajes para adaptar completamente la teoría de objetos a su filosofía y encontrar aplicaciones particularmente novedosas que permitan proponer un conjunto a la vez completo, preciso y con buen rendimiento.

Por este motivo se encuentran tantas diferencias. Por tanto, la forma de trabajar de Python está completamente adaptada al lenguaje, aunque no puede decirse que el modelo de objetos de Python sea mejor que el de C++, por ejemplo. El modelo de C++ está adaptado a C++ y el de Python lo está a Python. Si se hubieran retomado los conceptos de C++ en Python, estos no habrían encontrado lugar, y viceversa.

Al final, cuando se viene de trabajar en otro lenguaje, adquirir práctica puede resultar más o menos fácil en un primer lugar, aunque para comprender realmente las diferencias y sutilidades es necesario ir más allá en el modelo de objetos, en la teoría, y comprender las elecciones realizadas y su adaptación a las características del lenguaje.

Por ello, no se deje sorprende por el hecho de que no existan las palabras clave **`new`** o **`this`**, que la firma de los métodos sea diferente, sino comprenda la filosofía general y saque provecho de las posibilidades que se ofrecen.

Python se ha creado en un momento en el que los lenguajes de referencia ya existían y habían marcado su tiempo. Ha aprovechado su experiencia y sacado el mejor provecho. A día de hoy, el propio lenguaje Python es una fuente de inspiración.

1.1.3 Generalidades

El objeto es uno de los pilares esenciales de Python, que decide proporcionar un lenguaje donde todo es un objeto, con el objetivo de responder de manera sencilla y eficaz a problemáticas complejas, permitir una gran flexibilidad y ofrecer una gran libertad de acción a los desarrolladores, como veremos en este capítulo.

Python tiene un único principio, que es «todo es un objeto», lo cual no es simplemente un concepto genérico. En efecto, si es evidente que una instancia es un objeto, el hecho de que todo sea un objeto quiere decir que la propia clase es un objeto, que un método es un objeto y que una función es un objeto.

Esto significa que todas las clases, funciones y métodos disponen de atributos y de métodos particulares, y que pueden modificarse tras su creación.
De este modo, es posible declarar clases, métodos y funciones de manera imperativa, mediante el uso de las palabras clave **`class`** o **`def`**, aunque también pueden declararse por asignación, abriendo así posibilidades muy interesantes.
Pero Python no es un lenguaje doctrinal con una única visión y buscando imponerla. Si bien el objeto está en el núcleo de sus funcionalidades, los demás paradigmas no se han rechazado o dejado de lado. Son tan importantes los unos como los otros.
En efecto, en función de la tarea que quiera cumplir, habrá una única manera evidente de proceder y aun así se podrá recurrir a uno de los tres paradigmas: imperativo, orientado a objetos o funcional.
Python no preconiza la superioridad del objeto, ni busca impedir la programación imperativa para obligar a que se utilicen objetos simplemente porque el objeto sea un enfoque más moderno o más de moda.
Es, por otro lado, interesante, cuando se conocen varios lenguajes, ver cómo Python es capaz de vincular la experiencia imperativa con la orientación a objetos y hacer emerger lo mejor de cada una.
Los debutantes que ya conozcan alguno de los paradigmas podrán desarrollar utilizando preferentemente el paradigma que conozcan y, a continuación, descubrir los demás poco a poco, en función de su experiencia.

1.2 Clases

1.2.1 Introducción

Una clase se define, simplemente, así:

```
>>> class A:
...     pass
```

Esta definición es de naturaleza imperativa, en el sentido de que una clase es un bloque que contiene un conjunto de instrucciones imperativas que se recorren y ejecutan unas detrás de otras.
Estas instrucciones pueden ser un docstring, por ejemplo:

```
>>> class A:
...     """Descripción de mi clase"""
```

Existen, en realidad, dos formas de describir una clase: bien utilizando este modo imperativo, descriptivo, que pone de relieve la encapsulación (utilizada a menudo), o bien mediante un prototipo, que también permite Python, de manera similar a JavaScript, como veremos más adelante.

1.2.2 Declaración imperativa de una clase

Una clase puede contener instrucciones declarando una variable, que se convierte en un atributo de clase, o una función, que se convierte en un método.

```
>>> class A:
...     """Descripción de mi clase"""
...     atributo = "Esto es un atributo"
...     def método(self, *args, **kwargs):
...         return "Esto es un método"
```

Una clase encapsula, así, con claridad todos sus datos, que son accesibles:

```
>>> A.__doc__
'Descripción de mi clase'
>>> A.atributo
'Esto es un atributo'
>>> A.método
<function método at 0x257ea68>
```

De este modo, el método es un atributo como los demás, pues cuando se accede sin invocarlo se devuelve la instancia correspondiente al método.
La única complejidad en la creación de clases se desprende de la complejidad funcional, del correcto modelado de objetos y de sus relaciones. Se recomienda trabajar de modo que los datos de una clase o de una instancia le pertenezcan y estén gestionados por la propia instancia, y no desde el exterior.

1.2.3 Instancia

Creemos una instancia:

```
>>> a = A()
```

Y accedamos al contenido de la instancia, definido en la clase:

```
>>> a.__doc__
'Descripción de mi clase'
>>> a.atributo
'Esto es un atributo'
>>> a.método
<bound method A.método of <__main__.A object at 0x25dfa90>>
```

Los atributos y métodos de la clase están, ahora, disponibles para la instancia, puesto que incluye un vínculo hacia los elementos de la clase, tal y como sugiere el término «bound».
Como puede verse, el hecho de acceder al método devuelve, simplemente, un objeto, aunque no invoca al método. Para realizar dicha llamada es necesario agregar los paréntesis y pasar los eventuales argumentos.
Recordemos que la firma del método espera un parámetro. Este parámetro representa, en realidad, la instancia. El vínculo entre el primer argumento del método definido en la clase y la instancia se realiza de manera natural:

```
>>> a.método()
'Esto es un método'
```

Es la gramática del lenguaje la encargada de informar el primer argumento situando la instancia. En Python no se hace magia, no existe ninguna variable mágica que represente automáticamente la instancia en curso; esta última está realmente visible y presente en la firma. Por convención, se denomina **`self`**.

La noción de interconexión entre una instancia y su clase es un elemento importante que debe dominarse. En efecto, si de una u otra manera se modifican los elementos de la clase, entonces se modifican también los elementos de la instancia:

```
>>> class B:
...     a = 'Otro atributo'
...     def m(self, *args, **kwargs):
...         return 'Otro método'
...
>>> A.atributo = B.a
>>> A.método = B.m
>>> a.atributo
'Otro atributo'
>>> a.método()
'Otro método'
```

Igual que la instancia no tiene su propio atributo, su valor es el de la clase y, como el atributo de la clase es dinámico, cualquier cambio realizado sobre este afectará a la instancia. No tener claro este aspecto puede llevarnos a generar errores inesperados. No obstante, preste atención: no es así como declaramos los atributos únicos de cada instancia, sino que se pasan al constructor, como veremos más adelante.

Informar los atributos directamente a nivel de la clase sirve, también, para que se compartan entre todas las instancias. Son, de algún modo, atributos de clase, en la semántica de Python, lo que equivale a atributos estáticos en la mayoría de los lenguajes.

Si bien estos atributos son de la clase, nada impide que un atributo con el mismo nombre aparezca en una instancia.

En este momento, podemos considerar que el atributo de la clase contiene el valor por defecto y el atributo de la instancia contiene el valor asociado de manera durable a la instancia.

Cuando el atributo de una instancia se modifica y recibe otro valor diferente al de la clase, se encuentra desconectado del atributo de la clase.

```
>>> a.atributo = 'Atributo de instancia'
>>> A.atributo
'Otro atributo'
>>> a.atributo
'Atributo de instancia'
```

El atributo de la instancia está conectado al de la clase. Ahora:

```
>>> a.atributo = A.atributo
```

Nos contentamos con realizar una asignación, sin cambiar de valor:

```
>>> A.atributo = 'B'
>>> a.atributo
'A'
```

1.2.4 Objeto en curso

Se denomina objeto «en curso» a la instancia en curso de la clase.
En Python, dicha instancia se denomina **`self`**, aunque no es más que una convención. Lo que importa es que el objeto en curso es, sistemáticamente, el primer objeto que recibe como parámetro un método, y dicho vínculo se establece de forma automática.
En la mayoría de los lenguajes existe una palabra clave **`this`** que permite ejecutar un método como una función, un poco de forma mágica, pues **`this`** representa a la instancia en curso.
Como a Python no le gusta la magia y quiere preservar la legibilidad, se contenta con exigir un primer argumento que representa a la instancia y el vínculo se establece a bajo nivel, pero no hay ningún elemento mágico de por medio. Lo que se utiliza en la función es, simplemente, variables que se presentan en la firma del método.
Cabe destacar que no se utiliza la palabra clave **`this`** ni la palabra clave **`new`** para crear la instancia.

1.2.5 Declaración por prototipo de una clase

La programación orientada a objetos por prototipo consiste en crear una clase y, a continuación, asignarle atributos y métodos como se hace, por ejemplo, en JavaScript.
Esto es muy diferente a la programación orientada a objetos clásica, puesto que nos contentamos con declarar una clase que es un recipiente vacío con un nombre y, a continuación, se le agregan atributos y métodos.
Estos métodos pueden ser, para ciertos lenguajes, simples funciones que transforman un objeto que se pasa como parámetro o que reciben un objeto como parámetro para devolver otro objeto sin que exista ningún vínculo entre ellos, salvo el hecho de agregarse en la misma clase.
El recurso de una palabra clave permite, por tanto, crear un vínculo artificial pero suficiente entre los métodos de una misma clase y sus propiedades. Esto puede parecer una agregación de propiedades y de funciones similares a lo que serían atributos y métodos.
Semánticamente, el uso de una clase así es idéntico al de una clase declarada de manera clásica, aunque los mecanismos internos sean totalmente distintos.
Esto no entra, en absoluto, en el espíritu de la programación orientada a objetos, pues si bien la encapsulación se resuelve de una manera diferente, aunque comprensible, los demás mecanismos tales como la instanciación, la diferenciación de instancias o el polimorfismo, por ejemplo, no pueden resolverse, o bien se resuelven de manera poco satisfactoria. Además, ciertos lenguajes hacen todas las clases puramente estáticas.
Estos lenguajes son, entonces, una interpretación del paradigma orientado a objetos bastante reducida, aunque por el contrario representan una ventaja indiscutible, que es la capacidad evolutiva, dado que, en cualquier momento, es posible agregar o modificar funciones.
En efecto, en la mayoría de los lenguajes, una vez declarada la clase, es imposible agregar nuevos métodos o atributos. En ocasiones, una permisividad natural permite agregar atributos de manera lateral. No obstante, esto es una limitación importante que hace que la programación orientada a objetos por prototipo encuentre su verdadero lugar.

En lo relativo a Python, esto es muy distinto. Por un lado, su lectura extrema del paradigma orientado a objetos hace que las propias clases, funciones y métodos sean objetos sobre los que es posible actuar como con cualquier otro objeto. Por otro lado, el hecho de que sea dinámico implica que, en todo momento, sea posible realizar una asignación o una modificación.
De este modo, es posible declarar una clase y, a continuación, añadir más tarde un atributo, por agregación. Para comenzar, creemos una clase de manera declarativa, como hemos hecho hasta ahora:

```
>>> class Declarativa(object):
...     """Clase escrita de manera declarativa"""
...
...     atributo_de_clase = 42
...
...     def __init__(self, name):
...         self.name = name
...         self.subs = []
...
...     def __str__(self):
...         return "{} ({})".format(self.name, ", ".join(self.subs))
...
 ...    def mostrar(self):
...         print(self)
```

Ahora podemos utilizar este objeto:

```
>>> a = Declarativa("test")
>>> a.subs.append("cosa", "chisme")
>>> print(a)
test (cosa, chisme)
>> dir(a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'atributo_de_clase']
>>> Declarativa.mostrar
<function Declarativa.mostrar__ at 0x7fb456895bf8>
```

Presentaremos, ahora, el código equivalente al anterior, escrito mediante prototipo. Veremos que primero se escriben los métodos:

```
>>> def proto__init__(self, name):
...     self.name = name
...     self.subs = []
...
>>> def proto__str__(self):
...     return "{} ({})".format(self.name, ", ".join(self.subs))
...
>>> Prototipo = type("Prototipo", (object,), {
```

```
    "__init__": proto__init__,
    "__str__": proto__str__,
    "atributo_de_clase": 42})
```

También es posible agregar funciones más tarde:

```
>>> def mostrar(self):
...     print(self)
...
>>> Prototipo.mostrar = mostrar
```

El resultado es completamente idéntico a nuestra clase declarada de manera clásica:

```
>>> dir(Prototipo)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'atributo_de_clase']
```

Esta forma de operar no es un error de programación o de diseño, es algo natural y que está previsto en Python.

Como un método no es más que una función encapsulada en una clase (si bien sigue algunas reglas particulares suplementarias que se presentan en la sección Métodos), Python no tiene ningún problema con esta forma de trabajar.

```
>>> def m(self):
...     return "Definido por prototipo"
...
>>> A.método = m
```

Esta forma de trabajar es bastante diferente a la de los lenguajes específicamente cualificados como «programación orientada a objetos por prototipo», como por ejemplo JavaScript, uno de los más conocidos y utilizados en este dominio.

Sin embargo, es bastante limitada en Python, a parte de las librerías que utilizan masivamente nociones complejas, tales como metaclases; por ejemplo, para resolver requisitos específicos de diseño.

La gran ventaja de esta técnica es que permite modificar las clases en cualquier momento, o extenderlas tanto como se quiera. Podemos perfectamente declarar una clase de la manera habitual, y después, en otro módulo importarla y agregarle métodos o atributos.

1.2.6 Tuplas con nombre

Existen muchos casos de uso en los que se necesita la flexibilidad de un objeto pero no se desea pasar demasiado tiempo escribiendo una clase. Para ello, existen las tuplas con nombre:

```
>>> from collections import namedtuple
>>> Punto = namedtuple('Punto', ['x', 'y'])
```

`Punto` es una clase particular que dispone de dos atributos **`x`** e **`y`**. Puede instanciarse:

```
>> p = Punto(4, 2)
```

Capítulo 3
Condiciones, pruebas y buleanos

1. Pruebas y condiciones

1.1 Las condiciones son esenciales

En nuestra vida, nuestro comportamiento se rige por una multitud de decisiones que debemos tomar. Volviendo al ejemplo del paso de peatones, cruzaremos si el semáforo de peatones está en verde, de lo contrario esperaremos **si** está en rojo. Lo mismo ocurre con los algoritmos y los programas: tenemos que guiar al ordenador para que tome las decisiones adecuadas y ejecute correctamente nuestras instrucciones.

Recuerde que hay que explicárselo todo a la máquina, nunca tomará una decisión por sí sola, salvo quizás apagarse en caso de cortocircuito. Hay que indicarle al ordenador cuándo puede llevar a cabo las instrucciones. Por ejemplo, como cuando un adulto enseña a dibujar a un niño pequeño: el adulto enseña al niño cómo coger el lápiz, cuál es el extremo adecuado para dibujar, que solo se puede dibujar sobre una hoja de papel y no sobre una mesa o una pared, etc. La ventaja de la programación para nosotros es que nuestras explicaciones son mucho más sencillas de formular y que la máquina nos escucha necesariamente sin hacer nunca lo que ella quiera y, sobre todo, lo entiende perfectamente a la primera.

Las pruebas y condiciones representan una idea básica muy simple para guiar nuestro programa: elegir ejecutar una instrucción determinada en función de la validez de una condición. De esta manera, comprobamos la validez de una condición para dar permiso o no para seguir ejecutando el programa, por ejemplo: **si** el semáforo de peatones está en verde (condición), **entonces** cruzo (instrucción).

La prueba también puede contener una o más alternativas: **si** el semáforo de peatones está en verde (condición), **entonces** cruzo (instrucción), **de lo contrario** espero a que el semáforo de peatones se ponga en verde.

Cuando se habla de condición, en realidad se está hablando valor buleano. Los buleanos son los tipos más sencillos de la informática. Solo pueden recibir dos valores: `VERDADERO` o `FALSO`. por tanto, la relación entre una condición y un buleano es totalmente explícita, porque una condición siempre es una expresión buleana.

Una condición siempre es el resultado de una o varias comparaciones unidas por operadores lógicos (`Y`, `O` y `NO`).

Vamos a empezar viendo las estructuras condicionales con una única comparación, y luego introduciremos la lógica buleana (o álgebra buleana), para gestionar pruebas con varias condiciones.

1.2 Estructuras condicionales

En los algoritmos, existen dos estructuras condicionales:

- **SI ENTONCES SINO** permite probar cualquier condición con o sin una o más alternativas.
- En cambio, **CASO ENTRE** solo permite probar varias condiciones de igualdad con la misma variable en una única instrucción.

1.2.1 SI ENTONCES SINO

El SI algorítmico es un bloque de instrucciones sujeto que se ejecuta en función de una condición. Por este motivo, todas las líneas incluidas en el SI deben ir **indentadas** con una nueva tabulación.

Observación

Le recordamos la importancia de indentar un algoritmo o programa. La indentación simplifica la lectura, porque podemos detectar dónde empieza y dónde acaba un bloque sin tener que pensar. Por el momento, no puede ver realmente lo importante que es, pero a medida que avance el libro, será más que relevante con nuestros primeros programas complejos y completos.

Para indicar las instrucciones que se deben ejecutarse si la condición es verdadera, debemos precederlas de la palabra clave ENTONCES. Después de la palabra clave SINO, el algoritmo continúa normalmente, sin ninguna prueba.

Esta es la sintaxis de SI ENTONCES:

```
SI (condicion)
ENTONCES
   ...      // instrucciones a ejecutar si la condición
   ...      // es verdadera
FINSI
```

Ejemplo:

```
PROGRAMA Entero_positivo
VAR
   x : ENTERO
INICIO
   ESCRIBIR("Escriba un entero de su elección")
   x <- LEER()
   SI x > 0
   ENTONCES
      ESCRIBIR("Su entero es positivo")
   FINSI
FIN
```

En el algoritmo anterior, comprobamos si un número entero introducido por el usuario es positivo. ¿Qué ocurre si también queremos comprobar si el número entero es negativo? Lo primero que se le ocurriría sería añadir un nuevo SI.

Con dos SI consecutivos, el algoritmo comprueba ambas condiciones en cualquier caso, si el entero es positivo y después si el entero es negativo o viceversa, según el orden de los dos SI.

Sin embargo, estamos de acuerdo en que un número entero no puede ser positivo y negativo al mismo tiempo. Así que simplemente añadamos un SINO a nuestro algoritmo para los negativos:

```
PROGRAMA Entero_positivo_negativo
VAR
   x : ENTERO
INICIO
   ESCRIBIR("Escriba un entero de su elección")
   x <- LEER()
   SI x > 0
   ENTONCES
      ESCRIBIR("Su entero es positivo")
   SINO
      ESCRIBIR("Su entero es negativo")
   FINSI
FIN
```

Nuestro algoritmo es cada vez más preciso, pero aún nos queda un punto por definir: el entero con valor cero. El cero no es ni positivo ni negativo, así que es un caso especial.

Podemos añadir un SI al inicio del algoritmo para comprobar el valor cero, pero esta solución no es óptima. Sea cual sea el valor del número entero, se comprobará dos veces: una para la igualdad y otra para la superioridad, debido a lo SI que se suceden.

Una solución limpia y optimizada consiste en definir el cero como el caso por defecto del SINO añadiendo un nuevo SI, verificando si el entero es negativo SINO significa que es cero (ni positivo ni negativo). Poner un SI dentro de otro SI se llama instrucciones anidadas o **pruebas anidadas**.

```
PROGRAMA Entero_positivo_negativo_cero
VAR
   x : ENTERO
INICIO
   ESCRIBIR("Escriba un entero de su elección")
   x <- LEER()
   SI x > 0
```

```
   ENTONCES
         ESCRIBIR("Su entero es positivo")
   SINO
      SI x < 0
      ENTONCES
         ESCRIBIR("Su entero es negativo")
      SINO
         ESCRIBIR("Su entero es cero")
      FINSI
   FINSI
FIN
```

Recuerde siempre limitar las instrucciones y variables en sus algoritmos y programas para optimizar la gestión de la memoria, por las razones expuestas en el capítulo anterior y, por lo tanto, hacer que tu código sea más eficiente.

1.2.2 CASO ENTRE

Cuando anidamos muchos SI, nuestro algoritmo puede volverse difícil de leer y, por tanto, difícil de entender y corregir en caso de error. Una posible alternativa es utilizar el `CASO ENTRE`. Esta estructura condicional permite comprobar el valor de una variable y compararlo, **solo en términos de igualdad**, con otros valores. Al igual que el `SI`, permite tener un caso por defecto si ninguno de los valores probados es correcto. A continuación, se muestra la sintaxis:

```
CASO variable ENTRE :
   CASO1 : valor1
      ...   // instrucciones a realizar si variable vale valor1
   CASO2 : valor2
      ...   // instrucciones a realizar si variable vale valor2
   ...
   PREDETERMINADO
      ...    // instrucciones a realizar por defecto. Opcional
FINCASOENTRE
```

Puede probar tantos casos como necesite. El caso por defecto es completamente opcional.

Vamos a escribir un algoritmo para mostrar el nombre del mes en función de un número dado por el usuario. Por razones de legibilidad, nos limitaremos a los seis primeros meses del año. Nuestra primera versión se escribirá utilizando solo SI y la segunda utilizando CASO ENTRE.

```
PROGRAMA Mes_si_anidados
VAR
   mes : ENTERO
INICIO
   ESCRIBIR("Escriba un número entre 1 y 6 incluídos")
   mes <- LEER()
   SI mes = 1
   ENTONCES
       ESCRIBIR("Enero")
   SINO
       SI mes = 2
       ENTONCES
          ESCRIBIR("Febrero")
       SINO
          SI mes = 3
          ENTONCES
              ESCRIBIR("Marzo")
          SINO
             SI mes = 5
             ENTONCES
               ESCRIBIR("Mayo")
               SINO
               ESCRIBIR("Junio")
                 FINSI
           FINSI
       FINSI
   FINSI
FIN
```

```
PROGRAMA Mes_caso_entre
VAR
   mes : ENTERO
INICIO
   ESCRIBIR("Escriba un número entre 1 y 6 incluídos")
   mes <- LEER()
   CASO mes ENTRE :
      CASO : 1
         ESCRIBIR("Enero")
      CASO : 2
         ESCRIBIR("Febrero")
      CASO : 3
         ESCRIBIR("Marzo")
      CASO : 4
         ESCRIBIR("Abril")
      CASO : 5
         ESCRIBIR("Mayo")
      PREDETERMINADO
         ESCRIBIR("Junio")
   FINCASOENTRE
FIN
```

Es fácil ver, ya sea escribiendo o leyendo, que el algoritmo que utiliza `SI` se hace rápidamente complejo y puede dar lugar a errores, como la indentación o la sintaxis. Con el algoritmo usando `CASO ENTRE`, la escritura y la lectura son realmente naturales porque la sintaxis es más simple. Sin embargo, no olvide que `CASO ENTRE` solo puede probar igualdades, mientras que `SI` puede probar cualquier tipo de comparación.

Ahora veamos cómo combinar varias pruebas en una estructura condicional.

2. Lógica buleana

2.1 Condiciones múltiples

Escribir una condición que compruebe la validez de un único hecho es bastante lógico, incluso sencillo. La complejidad de las condiciones aumenta a medida que aumenta el número de hechos que hay que validar, ya sea en la vida cotidiana o en la informática.

Cuando ponemos a prueba múltiples condiciones como seres humanos, nuestros cerebros razonan tan rápido que no parece que estemos pensando o incluso resolviendo una ecuación. Pero en realidad sí lo estamos haciendo.

Volvamos al ejemplo del paso de peatones. Parece que es normal que, antes de cruzar un semáforo, se compruebe si el semáforo para peatones está en verde y también que ningún coche se salta el semáforo. Por tanto, está analizando dos condiciones y tiene la impresión de que está haciendo las dos cosas a la vez con una sola condición. Pero su cerebro recibe dos condiciones para comprobar, así que resuelve estas dos condiciones en una ecuación.

Como dijimos en el capítulo introductorio de este libro, es necesario describir todo a la máquina, paso a paso, no hay atajos. Así que necesitamos una forma sencilla de representar múltiples condiciones.

Para formular correctamente esta ecuación con varias condiciones, George Boole, matemático del siglo XIX, creó un álgebra binaria que Claude Shannon utilizó más de un siglo después en informática. El álgebra de Boole, también llamada lógica de Boole según el contexto, permite analizar varias condiciones a la vez en una misma ecuación y, por tanto, en una misma estructura condicional. Así que ya sabe de dónde viene el nombre de Buleano.

Implementar esta álgebra es casi natural en informática. Un ordenador funciona con impulsos eléctricos, por lo que `FALSO` se representa por 0 o ausencia de corriente, y `VERDADERO` por 1 o presencia de corriente.