Fundamentos de Informática

(Universidad de Oviedo)

Índice general

Presentación 4

I Introducción 5

1 Introducción 6
- 1.1 Visión general de la Informática . . . 6
  - 1.1.1 Conceptos y términos fundamentales . . . 6
- 1.2 Estructura y funcionamiento de un ordenador . . . 7
  - 1.2.1 Arquitectura y diagrama de bloques de un computador . . . 7
- 1.3 Representación de la Información en un ordenador . . . 9
  - 1.3.1 Sistema binario . . . 9
- 1.4 El lenguaje de la Lógica de Proposiciones . . . 16
  - 1.4.1 El alfabeto de las proposiciones . . . 16
  - 1.4.2 La sintaxis de las proposiciones . . . 17
  - 1.4.3 La semántica de las proposiciones . . . 19
  - 1.4.4 Las leyes lógicas de las operaciones con proposiciones . . . 23
  - 1.4.5 Traducción de lenguaje natural al lenguaje de la Lógica de Proposiciones . . . 24
- 1.5 Ejercicios . . . 27
  - 1.5.1 Ejercicios resueltos . . . 27
  - 1.5.2 Ejercicios propuestos . . . 28

II Introducción a la programación 30

2 Introducción a la programación 31
- 2.1 Abstracción de problemas para su programación. Conceptos fundamentales . . . 31
  - 2.1.1 ¿Qué es un programa? . . . 31
  - 2.1.2 Lenguajes de programación . . . 31
  - 2.1.3 Programas y algoritmos . . . 32
  - 2.1.4 Diseño de programas . . . 33
  - 2.1.5 `Python` . . . 34
- 2.2 Variables, expresiones, asignación . . . 36
  - 2.2.1 Valores . . . 36
  - 2.2.2 Tipos . . . 36
  - 2.2.3 Conversiones entre tipos . . . 37
  - 2.2.4 Operadores . . . 38
  - 2.2.5 Variables . . . 40
  - 2.2.6 Asignación . . . 42
  - 2.2.7 Otras consideraciones sobre las variables . . . 43

2.2.8 Expresiones . . . 45
2.3 Uso de entrada/salida por consola . . . 46
2.3.1 Entrada por teclado . . . 47
2.3.2 Salida por pantalla . . . 48
2.4 Manejo de estructuras básicas de control de flujo . . . 48
2.4.1 Estructura secuencial (BLOQUE) . . . 49
2.4.2 Estructura alternativa . . . 50
2.4.3 Estructura repetitiva `while` . . . 53
2.4.4 Estructura repetitiva `for` . . . 57
2.5 Funciones . . . 60
2.5.1 Uso de funciones . . . 61
2.5.2 Funciones que ya existen . . . 61
2.5.3 Definición de nuevas funciones . . . 62
2.5.4 Variables locales y variables globales . . . 67
2.5.5 Procedimientos: funciones sin devolución de valor . . . 68
2.6 Tipos y estructuras de datos básicas: listas y cadenas . . . 69
2.6.1 El tipo `list` . . . 70
2.6.2 Acceso a los elementos individuales de la lista . . . 71
2.6.3 El operador de corte . . . 76
2.6.4 Listas que contienen listas . . . 77
2.6.5 Bucles para recorrer listas . . . 78
2.6.6 Listas y funciones . . . 80
2.6.7 Listas y cadenas . . . 85
2.6.8 Un caso frecuente en ingeniería: listas de listas rectangulares, matrices . . . 89
2.6.9 Ejercicios resueltos. . . . 93
2.6.10 Ejercicios Propuestos . . . 101
2.7 Almacenamiento permanente . . . 101
2.7.1 Apertura y cierre de ficheros . . . 101
2.7.2 Lectura de líneas a lista . . . 102
2.7.3 Escritura en ficheros . . . 104
2.7.4 Ejercicios resueltos . . . 106

III Introducción a las bases de datos **108**

3 Introducción a las Bases de Datos **109**
3.1 Conceptos de bases de datos . . . 109
3.1.1 Definición de Bases de Datos (BD) y de Sistema de Gestión de Bases de Datos (SGBD). . . . 109
3.1.2 Funcionalidades de un SGBD . . . 109
3.1.3 Aplicaciones sobre Bases de Datos. . . . 111
3.2 Un ejemplo sencillo . . . 111
3.3 Diseño de Bases de Datos . . . 111
3.4 Modelos de Datos . . . 112
3.5 Modelo Entidad-Relación . . . 112
3.5.1 Introducción . . . 112
3.6 Modelo Relacional . . . 115
3.7 Uso básico del lenguaje SQL . . . 118
3.7.1 Ejemplos de órdenes SQL . . . 118
3.8 SGBD en entornos profesionales de la ingeniería . . . 120

3.8.1 Bases de Datos espaciales y geográficas 120

IV Componentes hardware y software de un sistema informático 122

4 Componentes hardware y software de un sistema informático 123
4.1 Estructura y funcionamiento de un computador 123
4.1.1 El sistema informático, hardware y sofware 123
4.1.2 Componentes físicos. El hardware 123
4.1.3 Dispositivos periféricos 130
4.2 Dispositivos de almacenamiento 133
4.3 Dispositivos de red 136
4.4 Tipos de software 137
4.5 Tipos de sistemas informáticos y sus ámbitos de aplicación 139

V Introducción a los sistemas operativos 141

5 Introducción a los Sistemas Operativos 142
5.1 Concepto y funciones que desempeña un sistema operativo 142
5.1.1 Estructura de un sistema computarizado 143
5.1.2 Arranque de un sistema computarizado 143
5.1.3 Funciones de un sistema operativo 145
5.2 Funciones que el sistema operativo presta a los programas 145
5.2.1 Administración de procesos 145
5.2.2 Administración de memoria 146
5.2.3 Administración del sistema de ficheros 146
5.2.4 Administración de dispositivos 148
5.3 Funciones que el sistema operativo presta a los usuarios 148
5.3.1 Interfaz usuario-ordenador 148
5.3.2 Acceso a las Redes de Computadores 149
5.3.3 Aplicaciones relativas a la seguridad del sistema 151
5.3.4 Herramientas para el mantenimiento del Sistema Operativo 155
5.4 Sistemas operativos utilizados en entornos profesionales de ingeniería 164
5.4.1 Sistemas operativos en tiempo real 164
5.4.2 Sistemas operativos empotrados 164

VI Referencias 166

Presentación

Este documento es un compendio cuyo objetivo no es otro que introducir al lector en los sistemas informáticos. Como se podrá observar a través de una primera lectura del índice, cubre aspectos desde el concepto de computador y cómo se representa el universo en el mismo hasta conceptos fundamentales en el diseño de bases de datos, pasando por programación de computadores, nociones de hardware y software de sistemas computarizados y sistemas operativos.

Como tal, este documento es fruto del trabajo colaborativo de muchos profesores del Departamento de Informática de la Universidad de Oviedo, quienes han prestado voluntariamente su tiempo y esfuerzo al desarrollo del mismo. El fin último no es otro que los alumnos de la asignatura Fundamentos de Informática dispongan de un material de referencia apropiado para la distribución de tiempo y créditos disponible para aquella.

Nos gustaría que el lector reflexione sobre este objetivo, ambicioso a la vez que complejo. Esto es así dado que la asignatura en cuestión es común a todas las ingenierías que se imparten en la Universidad de Oviedo: tanto ingenieros mecánicos, químicos, de montes, informáticos, etc. Y todos ellos deben utilizar el mismo documento, los mismos contenidos. Esta tarea, la de adaptar un documento a tal variedad de áreas de conocimiento es un reto que, desde nuestra óptica, se ha conseguido en este trabajo.

La organización de este documento es como sigue. Está organizado en capítulos, cada uno de ellos describe un tema de entre los contenidos de la asignatura. Cada capítulo puede estar dividido en uno o más secciones, en un intento de facilitar la lectura y estudio de sus contenidos. Para cada capítulo se incluyen ejercicios resueltos y ejercicios propuestos al alumno para el trabajo autónomo de éste.

Los materiales docentes relacionados con este documento están disponibles a través de `www.campusvirtual.uniovi.es` de la Universidad de Oviedo; en los sitios web de cada uno de los grupos de clases expositivas a impartir.

Parte I

Introducción

CAPÍTULO 1

Introducción

1.1. Visión general de la Informática

Informática, palabra cuyo origen proviene de la fusión de las palabras “Información” y de “Automática”, está definida por la Real Academia de la Lengua Española (RAE) como “Conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores.” Entenderemos las palabras computador (o computadora) y ordenador como sinónimos.

Pero, ¿Qué es información? ¿Cómo puede definirse la información? Las definiciones que ofrece la RAE son: *Acción y efecto de informar*, *Oficina donde se informa sobre algo*, etc. No parece que ayude mucho ... Quizás una adecuada definición de información sería “Conjunto de símbolos que representan hechos, objetos o ideas y se consideran como relevantes en un contexto o problema concreto”. O lo que indica la Wikipedia: “Secuencia ordenada de símbolos que almacenan o transmiten un mensaje”.

Lo mejor para entenderlo es poner ejemplos. En una transacción económica son relevantes los siguientes hechos: los ítems intercambiados, el coste unitario, cantidad de cada uno de los ítems que intervienen. Es también relevante cada uno de los actores (objetos) que intervienen. Sin embargo, en un principio no es relevante el estado del tiempo meteorológico del día de la transacción, luego este dato no es información.

En este tema se pretende introducir algunos conceptos básicos que serán útiles a lo largo del curso. Posteriormente se verán los temas relacionados con la introducción a la programación, una descripción más formal de los computadores y los sistemas operativos para, finalmente, introducir los conceptos fundamentales de bases de datos.

1.1.1. Conceptos y términos fundamentales

Podemos definir en este apartado una serie de conceptos que vamos a utilizar muy frecuentemente en esta asignatura. Estos son solo los fundamentales, los necesarios para entender este tema de introducción, por lo que se trata de una lista corta. En los diferentes temas a tratar se irán incluyendo el resto de conceptos a medida que surja la necesidad de utilizarlos.

computador o computadora es una máquina que genera una salida de información al aplicar una secuencia de operaciones lógicas y aritméticas a un conjunto de datos inicial.

ordenador en muchos países es un sinónimo de computador. Deriva del francés *ordinateur*, que significa “que ordena o pone en orden”... Luego, ¿qué sería más correcto, el uso de la palabra computador o, por contra, el uso de ordenador? Esta pregunta tiene una respuesta: ¡son sinónimos y no perdamos tiempo en tonterías!

algoritmo un procedimiento no ambiguo que resuelve un problema. Por procedimiento se entiende la secuencia de operaciones bien definidas, cada una de las cuales requiere una cantidad finita de memoria y se realiza en un tiempo finito.

programa es la secuencia ordenada de operaciones lógicas y aritméticas que se introduce en un computador para que las realice. Está expresado en términos que la máquina entienda.

programación de computadores es parte de la Informática dedicada al estudio de las distintas metodologías, algoritmos y lenguajes para construir programas.

sistema operativo es el conjunto de programas que gestionan los recursos y procesos de un computador, permitiendo el uso y explotación del mismo. El sistema operativo es el responsable de la normal ejecución de las diferentes aplicaciones.

aplicación conjunto de programas que llevan a cabo una tarea al completo. Puede incluir la integración de más de un programa. Ejemplo, aplicación informática de venta en internet: utiliza servidores web, sistemas gestores de bases de datos, etc.

1.2. Estructura y funcionamiento de un ordenador

En esta sección se introducirá de forma breve qué es un computador. Este punto de vista nos permitirá entender por qué se estudia la representación binaria. Igualmente, nos permitirá entender la diferencia entre programa y código del programa. Al desarrollo del código del programa, cómo llegar desde el enunciado de un problema al programa que lo resuelve, se dedicará el capítulo II.

1.2.1. Arquitectura y diagrama de bloques de un computador

Los ordenadores están diseñados, en su mayoría, siguiendo la conocida como **Arquitectura Von Neumann**. El **procesador** (μP) es el cerebro, es la responsable de controlar y ejecutar todas las operaciones que se ejecutan dentro del ordenador. Para que el ordenador pueda trabajar necesita, además, otros componentes hardware: la **memoria principal** -donde se almacena la información-, las **unidades de entrada/salida** -para acceso a los dispositivos perifétricos- y los **buses de interconexión** -para conectar todos los componentes del sistema entre sí- (ver figura 1.1).

En la figura 1.2 se puede observar un microprocesador bastante antiguo (el 68000 de Motorola). Este μP tiene muchas patillas, cada una tiene una función concreta. Por todas estas patillas se transmiten señales eléctricas a través de los buses hacia o desde los componentes conectados, en este caso la memoria (ver la parte izquierda de dicha imagen).

Las señales eléctricas que se utilizan sólo tienen dos posibles valores, por ejemplo, ó 0 Voltios ó 5 Voltios (parte derecha de la figura 1.2). Esto se asocia a los estados ALTO y BAJO, y por extensión, a los estados 1 y 0, o los estados CIERTO y FALSO. Es decir, se puede decir que el μP se expresa en **binario**. Para acceder a la dirección 25, por ejemplo, en las patillas del μP correspondientes al bus de direcciones debe aparecer la combinación binaria que sea equivalente al valor 25. Pero el bus de direcciones son cables ...

Y, ¿Cómo se puede representar un número con señales eléctricas? Bien, cada cable puede llevar en cada instante un valor, que puede ser un valor cercano a 0 voltios o bien distinto de cero. Es decir, cada cable lleva información de estado: o es BAJO o es ALTO. ¿Que significa esto? Que los buses -al igual de el μP- sólo pueden gestionar por cada cable dos posibles valores: 0 ó 1, BAJO o ALTO. Es decir, que **un computador sólo puede gestionar información binaria**. Cada una de estas señales binarias se denomina **bit** (acrónimo de binary digit).

El ser humano no maneja este tipo de información (al menos no lo hace de buen gusto). El tipo de información que utiliza el ser humano son números (tanto reales como enteros), texto, imágenes, vídeo, etc. Y sin embargo, somos capaces de interactuar con las máquinas y ellas con nosotros. ¿Cómo? La cuestión es que el computador almacena la información de forma binaria y la presenta

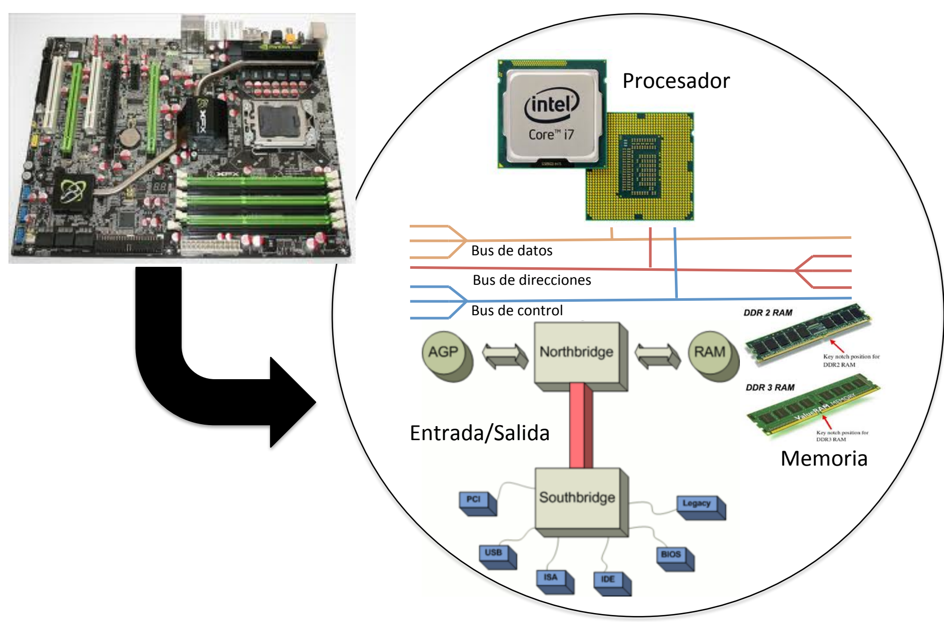


Figura 1.1: Elementos de un computador y su interconexión. Fuentes de las imágenes: http://hardzone.es/2012/05/09/intel-core-i5-3570k-vs-core-i7-3770k-datos-de-rendimiento-a-4-8ghz/, http://www.afsur.com/wp-content/uploads/2012/08/ddr3-vs-ddr2-ram.jpg, http://img.hexus.net/v2/motherboards/intel/X58/XFX/MoboT.jpg.

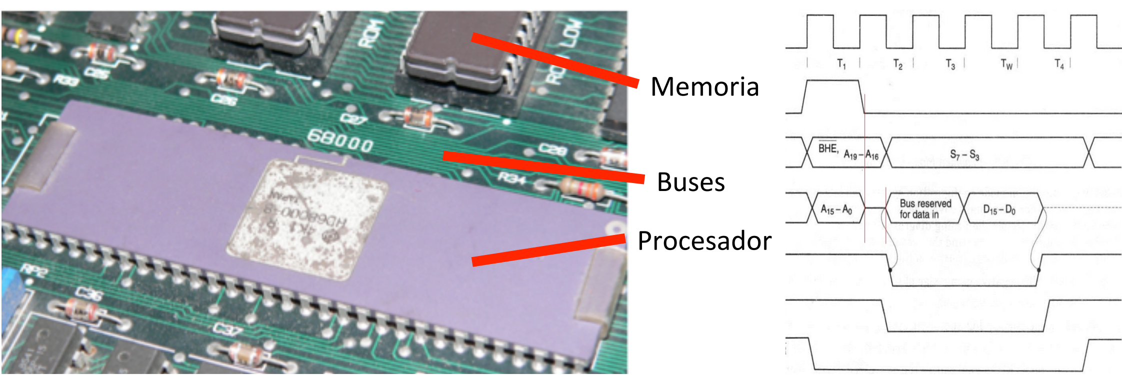


Figura 1.2: Ejemplo de un μP antiguo, pero que debido a la baja densidad de integración se puede observar fácilmente a simple vista. A la izquierda se puede ver las conexiones entre los diferentes dispositivos, mientras que en la parte derecha se puede observar la naturaleza de las señales eléctricas que se utilizan. Fuentes de las imágenes: http://retro.co.za/ccc/mac/, http://computersbrain.blogspot.com.es/2012/03/ale-signal-of-80868088.html.

al ser humano adecuadamente. Por ejemplo, el número 25 se almacena como 00011001, mientras que lo visualiza como 25.

1.3. Representación de la Información en un ordenador

Un computador representa la información utilizando dígitos binarios, es decir, en base 2. En esta base sólo se dispone de 2 valores diferentes: {0, 1}. El humano usa la base decimal, base 10, con 10 posibles valores diferentes: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Sin embargo, una combinación binaria se evalúa de igual manera que una combinación decimal, como puede verse en la figura 1.3.

En el sistema decimal
dígitos {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
$538_{10} = 5*10^2 + 3*10^1 + 8*10^0$
5 está en la posición 2
3 está en la posición 1
8 está en la posición 0

En el sistema binario
dígitos {0, 1}
$101_2 = 1*2^2 + 0*2^1 + 1*2^0 = 4+0+1 = 5_{10}$
1 está en la posición 2
0 está en la posición 1
1 está en la posición 0

Figura 1.3: Evaluación de una combinación decimal y de una binaria. Sólo se diferencia en i) los posibles dígitos y ii) la base. ¡El método es el mismo!

1.3.1. Sistema binario

El sistema binario se basa en utilizar la base 2. Aplicando lo visto anteriormente, los posibles dígitos son {0, 1}. Sólo se dispone de estos dos posibles valores. La unidad de información que es capaz de almacenar un dígito binario se denomina **bit**, que es la contracción de binary digit como se ha dicho previamente.

Utilizando bits tenemos que ser capaces de representar todo tipo de información si queremos que un computador sea capaz de procesarlo. Recordemos que los computadores sólo son capaces de gestionar información binaria. En las siguientes subsecciones se presentará cómo se pueden representar los diferentes tipos de información en formato binario.

Representación de números enteros positivos

Supongamos un computador de 8 bits -en vez de los megafantásticos actuales de 64 bits-. Con 8 bits debemos representar los enteros sin signo. Como se pudo ver en la figura 1.3, los enteros sin signo se representan mediante una combinación ordenara de bits, donde cada posición p tiene un peso de $base^p = 2^p$.

Para convertir de binario a decimal solo se deben sumar las sucesivas multiplicaciones de dígito por peso y se obtendrá el valor decimal equivalente. Luego el valor binario $01011001_2 = 0_7 1_6 0_5 1_4 1_3 0_2 0_1 1_0 = 0*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 64 + 16 + 8 + 1 = 89$.

Por el contrario, para llevar de decimal a binario se se realizan divisiones enteras sucesivas entre la base, utilizando el resto de la división como dígito binario. Sea el número 123_{10} a expresar o codificar en binario, el proceso se visualiza en la tabla 1.1. Observar el orden de los dígitos binarios: cada uno lleva implícitamente asociado el número de divisiones necesarias para poder extraerlo. Además, si hacemos la analogía con lo realizado para números decimales y cómo extraer sus unidades, se puede deducir que es la misma.

Ejemplos resueltos:

De decimal a binario 35:

1. $35/2 \rightarrow 17$ resto 1
2. $17/2 \rightarrow 8$ resto 1
3. $8/2 \rightarrow 4$ resto 0
4. $4/2 \rightarrow 2$ resto 0
5. $2/2 \rightarrow 1$ resto 0

Cociente	Resto	Posición
$123/2 = 61$	$123\,\%2 = 1$	0
$61/2 = 30$	$123\,\%2 = 1$	1
$30/2 = 15$	$123\,\%2 = 0$	2
$15/2 = \ \ 7$	$123\,\%2 = 1$	4
$7/2 = \ \ 3$	$123\,\%2 = 1$	4
$3/2 = \ \ 1$	$3\,\%2 = 1$	5
$1/2 = \ \ 0$	$1\,\%2 = 1$	6
Número binario	$1_6 1_5 1_4 1_3 0_2 1_1 1_0 \rightarrow$	01111011_2

Tabla 1.1: Conversión de un número codificado en decimal a su correspondiente codificación binaria. El operador % es el resto de la división entera o módulo.

6. $1/2 \rightarrow 0$ resto 1
7. El número es 100011_2

De decimal a binario 52:

1. $52/2 \rightarrow 26$ resto 0
2. $26/2 \rightarrow 13$ resto 0
3. $13/2 \rightarrow 6$ resto 1
4. $6/2 \rightarrow 3$ resto 0
5. $3/2 \rightarrow 1$ resto 1
6. $1/2 \rightarrow 0$ resto 1
7. El número es 110100_2

De binario a decimal 11011:

1. enumeramos las posiciones: $1_4 1_3 0_2 1_1 1_0$
2. elevamos las posiciones a potencia de 2 y multiplicamos por el dígito correspondiente: $1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$
3. sumamos todos los elementos: $16 + 8 + 0 + 2 + 1 = 27$

De binario a decimal 101101:

1. enumeramos las posiciones: $1_5 0_4 1_3 1_2 0_1 1_0$
2. elevamos las posiciones a potencia de 2 y multiplicamos por el dígito correspondiente: $1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$
3. sumamos todos los elementos: $32 + 0 + 8 + 4 + 0 + 1 = 45$

Sistema hexadecimal Ahora bien, ¿Cuál es el valor decimal del número binario 10001111101100? ¿Lo puedes recordar fácilmente? Menuda cantidad de operaciones o de memoria para poder gestionar esta combinación binaria ... Sin embargo, existe un truco: Aprovechando que $2^4 = 16$, es decir, que con 4 bits podemos representar 16 valores, se propuso el **sistema de números hexadecimales** o **base 16**.

Este sistema de representación utiliza 16 dígitos: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, $A_{=10}$, $B_{=11}$, $C_{=12}$, $D_{=13}$, $E_{=14}$, $F_{=15}$}. Su principal bondad es que permite acortar la representación binaria, lo que facilita la lectura por parte del ser humano.

La idea se basa en lo siguiente. Existe una tabla que relaciona directamente dígitos hexadecimales con sus correspondientes combinaciones de 4 bits (ver tabla 1.2). Los dígitos hexadecimales suelen llevar delante $0x$ para indicar que se utiliza la base 16.

Supongamos se tiene el número binario `10011100110011`, el cuál es muy difícil de recordar. Sin embargo, podemos agrupar sus bits de cuatro en cuatro, comenzando por la derecha, y dejarlo como `10 0111 0011 0011`, y si añadimos ceros por la izquierda para completar también cuatro bits

Base 10	Base 16	Base 2
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111

Base 10	Base 16	Base 2
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Tabla 1.2: Correlación entre los dígitos de los sistemas codificados en decimal (Base 10), hexadecimal (Base 16 ó 0x) y binario (base 2).

en el primer grupo, el número es el mismo: `0010 0111 0011 0011`. Utilizando la tabla 1.2 podemos decir que es el número `0x2733`, que es más facil de recordar y de llevar a decimal.

Para traducir de hexadecimal a decimal y viceversa se reutilizan las mismas operaciones que las vistas para traducir de decimal a binario:

- de hexadecimal a decimal: cada dígito h_i expresa una potencia de 16, por lo $0x3A2 = 0x3_{pos=2}A_{pos=1}2_{pos=0} = 3 \cdot 16^2 + A \cdot 16^1 + 2 \cdot 16^0 = 3 \cdot 256 + A \cdot 16 + 2 \cdot 1 = 768 + 160 + 2 = 930$.
- de decimal a hexadecimal: divisiones sucesivas entre la base (16).

Veamos otros ejemplos:

Ej1 110110

- agrupar de 4 en 4 bits, desde la derecha a la izda.: 11 0110
- completar con 0's hasta tener grupos de 4 bits: 0011 0110
- utilizar la tabla: 36 $_{16}$ = 0x36
- a decimal: $3 * 16^1 + 6 * 16^0$=48 + 6=54

Ej2 10001111101100

- agrupar de 4 en 4 bits, desde la derecha a la izda.: 10 0011 1110 1100
- completar con 0's hasta tener grupos de 4 bits: 0010 0011 1110 1100
- utilizar la tabla: 2 3 E C $_{16}$ = 0x23EC
- a decimal: $2 * 16^3 + 3 * 16^2 + 15 * 16^1 + 12 * 16^0$=8192 + 768 + 240 + 12=9212

Representación de números enteros negativos

Retornando al tema de representar información en un computador, analizaremos ahora los números enteros negativos. Para representar *números enteros negativos* se utiliza mayoritariamente el denominado **complemento a 2** dado que facilita el diseño de los circuitos integrados que deban operar con números enteros, positivos o negativos (los microprocesadores son circuitos integrados, p.e.). Se define el complemento a 2 de un número binario n (negativo) como el resultado de la operación $2^N - |n|$. Es decir, que si utilizo 8 bits ($N = 8$) y quiero representar $n = -5$, entonces, realizaré la operación $2^8 - 5$, cuyo resultado es 251, y representaré en binario el número 251. Se deja al lector el ejercicio de pasar a binario 251, y comprobará que el resultado es 11111011_b.

Un "atajo" para llegar al mismo resultado consiste en seguir los siguientes pasos: i) representar el número positivo en binario, ii) recorrer de derecha a izquierda la combinación resultante, iii) a partir del primer 1 -sin incluirlo-, cambiar los 0's por 1's y viceversa. Ejemplo: $n = -5$ en 8 bits. Representación de $|n|$ en binario es 00000101_2. Cambio los 0's y 1's a partir del primer 1 por la derecha: 11111011_2.

A la hora de realizar la suma final, ya que esta debe hacerse en binario, conviene conocer las reglas de la suma binaria, que se resumen en:

$0 + 0$ vale 0
$0 + 1$ vale 1
$1 + 0$ vale 1
$1 + 1$ vale 0 y me llevo 1
$1 + 1 + 1$ vale 1 y me llevo 1

Rango de los números enteros con y sin signo

Supongamos se utilizan N bits para representar enteros. La pregunta es ¿Cuántos enteros sin signo puedo representar con estos N bits?

Para entenderlo de forma sencilla, pongamos varios ejemplos.

$N = 1$ se puede representar 0 ó 1. Luego puedo representar 2 valores, o lo que es igual, 2^1.

$N = 2$ se puede representar 00, 01, 10 ó 11. Luego puedo representar 4 valores, o lo que es igual, $2^2 = 2 \cdot 2^1$.

$N = 3$ se puede representar 000, 001, 010, 011, 100, 101, 110, ó 111. Luego puedo representar 8 valores, o lo que es igual, $2^3 = 2 \cdot 2^2$.

... Cada vez que añadimos un dígito duplicamos el número de combinaciones, luego por inducción se llega a que, para un N cualquiera, el número de combinaciones son $2^N = 2 \cdot 2^{N-1}$.

Luego, con 8 bits puedo representar $2^8 = 256$ combinaciones binarias diferentes, o lo que es igual, 256 enteros.

¿Qué pasa con los enteros con signo? Pues que debido a la forma de representar los números negativos el bit más significativo discrimina el signo: si este bit vale 1 es un número negativo, positivo en caso contrario. Por lo tanto, con N bits puedo representar desde -2^{N-1} hasta $2^{N-1} - 1$. El que se represente un entero menos en caso de número entero positivo (observar el -1 que resta a 2^{N-1}) se debe a que el valor 0 está incluido entre los que tiene el bit de signo a 0... luego puedo representar un entero positivo menos.

Representación de números reales

La representación binaria de números reales se realiza en la notación científica -formato exponencial- mediante *ciertas* codificaciones binarias del signo, el exponente y la mantisa (en el caso de estas dos últimas la codificaciones no constituyen simplemente el cambio de base). El estándar IEEE 754, normalmente el de los ordenadores personales, de 32 bits el formato utilizado tiene el siguiente patrón:

seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm

donde la *s* representa el bit que codifica el signo, las *es* los 8 bits que codifican el exponente y las *emes* los 23 bits que codifican la mantisa. Además tienen las siguientes particularidades:

- El bit del signo será 0 para el positivo y 1 para el negativo.
- la codificación binaria del exponente representa un entero positivo o negativo, variando -expresándolo en decimal- entre -254 y 255.
- La codificación binaria que se adopta para la mantisa, y que aquí no se detalla, se corresponde con un número entero positivo (la magnitud, pues el signo ya se trató).

Un ejemplo de representación en el formato en 32 bits se muestra en la figura 1.4. Para el caso de 64 bits, el exponente se expresa en 11 bits y la parte fraccionaria en 52.

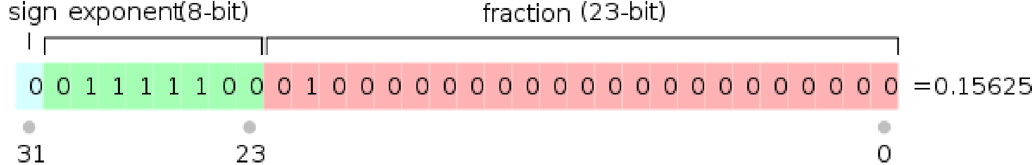


Figura 1.4: Representación de números reales en formato IEEE754 para 32 bits.

Representación de los datos booleanos: VERDADERO o FALSO

Siempre que el resultado de una operación o de una pregunta sea exclusivamente VERDADERO o FALSO estamos hablando de respuesta booleana (en honor a Georges Boole, matemático del siglo XIX y padre de la Lógica de Proposiciones). Los datos con este tipo de valores se denominan datos booleanos y son los datos que menor cantidad de información aportan. Para codificar esa información es suficiente con los dígitos 1(VERDADERO) y 0 (FALSO); es decir con los dos dígitos utilizados en el sistema binario de ahí que la unidad de información se denomine **bit** (**b**inary **d**igit). Por ejemplo, el resultado de lanzar al aire una moneda es un dato booleano: cara o cruz, pero no así el de lanzar un dado: 1,2 ... ,6.

Teóricamente los tipos de datos booleanos solo necesitan un bit para almacenarlos, aunque en los ordenadores su arquitectura hace que se reserve más memoria de la que utilizan. Por último, observar que en lenguajes de programación (como Python) FALSO (False) se identifica con el 0 y todo valor que no sea cero se identifica con VERDADERO (True).

Unidades de información y sus múltiplos Como se ha dicho, un bit es la unidad básica de información. Sin embargo, como resulta ser tan poca información lo habitual es hablar de múltiplos de bit y generalmente múltiplos muy grandes. Por otra parte, al ser una unidad muy pequeña y pensando en el almacenamiento de diferentes tipos datos, es necesario utilizarla en *paquetes*. Así, se conoce como **byte** la agrupación de 8 bits para formar otro unidad de información más funcional. Con el discurrir de los tiempos y el avance de la tecnología fueron necesarios empaquetaduras de bits mayores, por lo que todos hemos oído hablar, por ejemplo, de arquitecturas de ordenadores personales de 32 bits, que son 4 bytes, y de 64 bits, son 8 bytes. La notación **b** se refiere a bit, mientras que **B** se refiere a byte. Se utilizan indistintamente ambas unidades hasta el punto de que suele haber cierta confusión, a veces creada interesadamente en el plano comercial para confundir al comprador. Los múltiplos del bit (b) y del byte (B) se pueden expresar, siguiendo el Sistema Internacional de Unidades, en términos de potencias de 10 (observa que la abreviatura de Kilo, lleva la k minúscula), aunque a veces se utiliza otro sistema en términos de potencias de 2 (por ser 2 la base del sistema de numeración binario con el que se codifica la información) y que aquí omitimos; ese otro sistema facilita más todavía el equívoco, a veces intencionado, en la magnitud de la cantidad de información que se desean transmitir. Las denominaciones y los símbolos en el sistema Internacional de Unidades de los múltiplos del bit y del byte son:

Kilo expresa 10^3: 1 kb son 1000 bits, 1 kB son 1000 bytes

Mega expresa 10^6: 1 Mb son 1000 kb, 1 MB son 1000 kB

Giga expresa 10^9: 1 Gb son 1000 Mb, 1 GB son 1000 MB

Tera expresa 10^{12}: 1 Tb son 1000 Gb, 1 TB son 1000 GB

Peta expresa 10^{15}: 1 Pb son 1000 Pb, 1 PB son 1000 TB

Representación de caracteres

Como ya se ha expuesto, los computadores sólo almacenan combinaciones de bits, cada bit representa un valor $\{0, 1\}$ o binario. Por lo tanto, toda información a almacenar se debe representar como una combinación binaria.

Esto incluye a los caracteres. Un carácter (bien sea un espacio en blanco, un carácter alfabético, un dígito, un signo de puntuación, etc.) se representa con una combinación binaria. Pero no una representación binaria cualquiera, sino una representación binaria bien conocida: se han definido **códigos estándar**.

Los códigos estándar son tablas publicadas como si de estándares industriales se tratasen (es decir, están aceptadas por los fabricantes de programas y sistemas operativos), donde básicamente a cada carácter se le asigna un valor entero, lo que permite representarlo con un código binario.

Uno de éstos códigos es el conocido como **ASCII**, que es el acrónimo de la *American Standard Code for Information Interchange*. Esta tabla de códigos (disponible en `http://www.lookuptables.com` o en `http://www.asciitable.com/`) utiliza 7 bits, por lo que el número de combinaciones diferentes es de $2^7 = 128$ caracteres.

Así, al carácter 'a' se le asigna el valor $97_{10} = 61_{16}$, mientras que al carácter 'A' se le asigna el $65_{10} = 41_{16}$. Un observador habituado a trabajar con representaciones de enteros en *formato hexadecimal* verá que la diferencia entre mayúsculas y minúsculas es un bit (en este tema se presenta el sistema numérico hexadecimal en el apartado 1.3.1). Esto no es fruto de la casualidad, sino del trabajo de matemáticos e ingenieros, permitiendo codificaciones que reduzcan la capacidad computacional necesaria para trabajar con caracteres.

De igual manera, las letras mayúsculas van seguidas una de otra en el orden alfabético. Igual ocurre con las letras minúsculas y también con los dígitos. Esto permite operar con los caracteres, como se opera con los enteros:

- Si resto dos letras mayúsculas obtengo el número entero correspondiente al número de letras que hay entre ambas, e.g., 'J'-'A'$\rightarrow 74 - 65 = 9$.
- Ídem con la letras minúsculas, 'j'-'a'$= 9$.
- Los caracteres correspondientes a dígitos se pueden restar, resultando en la diferencia entre el valor de ambos, '5'-'3'$= 53 - 51 = 2$.

La extensión a 8 bits del código ASCII es el conocido como CP852 (Code Page 852, utilizado por IBM; conocido también como OEM 852, MS-DOS Latin 2, PC Latin 2, Slavic -Latin II-)

No obstante, el código ASCII y otros códigos similares presentan un grave problema: no pueden representar todos los símbolos de todos los lenguajes. Si pensamos en la gran cantidad de lenguajes existentes, cada uno con sus caracteres específicos (la Ñ, por ejemplo), se desprende que 128 son pocos... El ASCII se extendió a 8 bits (256 posibilidades o combinaciones diferentes), pero el problema continua existiendo para el **ASCII extendido**.

Por ello se han diseñado otros sistemas de codificación de caracteres, intentando universalizar el código. Así nace el **Unicode**, cuya intención fue la de permitir una representación de caracteres común a través de internet, y que contemple todos los caracteres de todos los alfabetos existentes. Este estándar asigna un código binario único a cada posible letra de cada posible carácter. Para almacenar o transmitir estos códigos binarios se definen diferentes formatos (UTF-8, UTF-16 o UTF-32). Nos centraremos en el UTF-8 al ser el estándar de facto en WWW, debido a que requiere menos espacio de almacenamiento y menos ancho de banda.

¿Qué hace especial al UTF-8? Pues que utiliza de uno a 4 bytes de 8 bits para referenciar a cada uno de los caracteres utilizados en la tabla **Unicode Standard** (`http://www.unicode.org/charts/`). Las 128 primeras posiciones utilizan un byte y se corresponden con el ASCII estándar. A medida que se va avanzando en la página de códigos se van utilizando más bytes, hasta un máximo

de 4 bytes. Cabe señalar que con 3 bytes bastan para codificar cualquier letra de cualquier alfabeto vivo.

Cuatro bytes son sólo necesarios para alfabetos muy raros y que no se usan hoy día para la comunicación escrita. En el caso del idioma español, la mayoría de los caracteres requieren un solo byte, por ser compatibles con ASCII, salvo las vocales acentuadas y la eñe, que no forman parte del ASCII y requieren 2 bytes en UTF-8. El signo del euro requiere tres bytes.

Además de texto plano...

Hemos visto cómo se almacenan los datos de texto: los archivos de texto plano -que no tienen formato, como cursiva, etc. se almacenan usando los códigos ASCII o UTF correspondientes a todos y cada uno de los caracteres que contiene. Pero, ¿Qué pasa con las imágenes, con los vídeos, con el audio? ¿Cómo se representa en un ordenador? La respuesta simplista: en binario. Bien, para representar cada tipo de información se requiere de un formato de almacenado.

Así como la industria se puso de acuerdo para el tema de los caracteres, con estándares que permiten un uso eficaz del hardware del computador, lo mismo ocurre con los diferentes medios audiovisuales: la industria adopta estándares que permiten un manejo eficaz de los dispositivos.

Así surgen los diferentes estándares para almacenamiento e intercambio de datos. A modo de ejemplo se enumeran algunos de los más conocidos.

Archivos comprimidos

zip que es onomatopeya de algo que pasa raudo, veloz. Es un compresor de archivos sin pérdidas cuyo algoritmo de compresión (PKZIP), desarrollado por Phil Katz y dejado libre.

rar es un algoritmo de compresión de archivos sin pérdidas desarrollado por Eugene Roshal.

tar es un formato para embeber diferentes archivos en un sólo núcleo de información que pueda ser comprimido y/o almacenado eficazmente.

Archivos con imágenes

JPEG, JPG Joint Photographic Experts Group, utilizando un algoritmo de compresión con pérdidas parametrizable.

GIF Graphics Interchange Format, que presenta pérdida de información, utilizando bitmaps con 8 bits por pixel para especificar el color y compresión sin pérdidas LZW.

TIFF Tagged Image File Format, permite contener en un solo archivo varias imágenes en calidad JPEG conjuntamente con etiquetas, etc. Actualmente está bajo control de Adobe. Se puede almacenar información con o sin pérdidas.

PNG Portable Network Graphics. Formato gráfico diseñado para sustituir a otros de similar funcionalidad (GIF, TIFF), pero sobre los que pesaban patentes. Almacena las imágenes con un algoritmo de compresión sin pérdidas, y admite diferentes sub-formatos con diferente número de bits por color. También soporta colores “transparentes”.

Documentos electrónicos

PDF Portable Document Format diseñado por Adobe, soporta tanto texto como imágenes vectoriales y en mapas de bits. Es un formato definido para facilitar la impresión y el intercambio de archivos.

PS PostScript es un lenguaje para la descripción de páginas que es utilizado en muchos sistemas de impresión así como para el intercambio de documentos entre múltiples plataformas y en talleres de impresión profesional.

XML Extensible Markup Language es un conjunto de reglas para codificar documentos (incluyendo texto, imágenes, etc., en general, información) de forma que puedan ser intercambiados sin ambigüedad entre máquinas y fácilmente presentables para lectura al ser humano.

Archivos de audio/video

MPEG 1 a 4 Moving Pictures Experts Group-Audio para almacenado de audio sin distorsión ni pérdidas.

MPEG video Moving Pictures Experts Group para transferencia de video y audio.

1.4. El lenguaje de la Lógica de Proposiciones

En lenguaje natural -el español en nuestro caso- una **proposición lógica** es una frase en forma de aseveración, o aserto, para la que un aspecto de su valor semántico es que puede ser únicamente verdadera o falsa, no caben valores intermedios.
Por ejemplo, *llueve* es una proposición, como también lo es *él come*. Sin embargo, no es una proposición *cuadrado* ni *esquina*.

Para determinar si la proposición *llueve* es cierta o falsa hay que asociarle por el contexto una **interpretación**, o si no proporcionarle una específica; por ejemplo, pensado en lo que ocurre ahora mismo cuando se escriben estas líneas, que efectivamente llueve, esa proposición sería verdadera. Por el contrario, con la interpretación de la meteorología de ayer a la misma hora, que fue un magnífico día de sol, sería falsa. Si hubiese que proporcionarle una interpretación concreta, como puede ser que fuese *verdadera*, se pondría: $llueve = \mathbf{V}$. También caben dentro de la definición de *proposición* frases como *llueve y come* cuya veracidad o falsedad dependerá de los veracidad o falsedad de proposiciones simples que la componen y de la conjunción que las une. En en lenguaje natural, una vez asociada una interpretación que abarque cada una de las proposiciones que la componen, todos sabemos establecer la verdad o falsedad de la frase resultante de una manera intuitiva.

La **lógica de proposiciones** es un *lenguaje formal* (es decir, establecido mediante reglas que especifican su construcción) que trata de captar del lenguaje natural las propiedades descritas de las proposiciones. La utilidad de ese lenguaje es servir como *modelo* para el estudio de los razonamientos que en lenguaje natural utilizan solamente proposiciones, lo que conlleva el estudio de otras propiedades colaterales que nos será útiles en nuestro entorno; de hecho serán algunas de esas propiedades el único objetivo de nuestra introducción a la lógica de proposiciones.

Un **lenguaje formal** consta de un *alfabeto* que contiene los símbolos con los que se construirán la palabras, una *sintaxis* o reglas de construcción de las palabras, y una *semántica*, o reglas para determinar su significado.

El **lenguaje formal de la lógica de proposiciones** consta de:

1.4.1. El alfabeto de las proposiciones

El **alfabeto** de las proposiciones está formado por:

1. Un conjunto de *símbolos proposicionales* que comprende un conjunto de símbolos prefijados, en este caso serán:

$$\{A, B, C, A_1, A_2, A_3, ..., B_1, B_2, B_3, ..., C_1, C_2, C_3\}$$

2. Los símbolos **V** y **F** de *Verdadero* y *Falso*, respectivamente.

3. Los *símbolos de conectivas* u *operadores lógicos*:

$$\{\wedge, \vee, \neg\}.$$

4. Los *símbolos auxiliares*: paréntesis:*(,)*, corchetes: *[,]* y llaves: {,}.

Los símbolos de conectivas servirán para señalar operaciones entre proposiciones. Los dos primeros, conjunción y disyunción, son binarios, porque operarán sobre dos proposiciones y el último, la negación, es unario, porque operará solo sobre una proposición.

Los paréntesis corchetes y llaves sirve para romper -a veces enfatizar- el orden de aplicación de los operadores que viene preestablecido por convenio, como se verá más adelante.

Además, para trabajar con proposiciones utilizaremos en ocasiones metasímbolos; es decir, símbolos que están fuera -más allá- del alfabeto y que se utilizarán para representar proposiciones cualesquiera. Estos símbolos jugarán el papel de las variables en lógica y en nuestro caso serán: P, Q, R, S y T . Si hiciesen falta más utilizaríamos subíndices bajo esas letras: $P_1, P_2, ...$.

Los conjuntos de símbolos proposicionales y el de conectivas no son universales, hay mucha variación según los diferentes autores a la hora de definir el alfabeto de las proposiciones.

Por ejemplo, en los símbolos proposicionales es frecuente encontrar en la literatura conjuntos de símbolos de letras minúsculas: $\{a, b, c, a_1, a_2, a_3...\}$, $\{p, q, r, s, p_1, p_2, p_3, ...\}$, etc. Incluso alfabetos específicos de un entorno muy concreto, como el de nuestro ejemplo de introducción a la lógica de proposiciones: {*llueve,come, ...*}, que sería en ese caso un alfabeto. En cada ocasión se puede utilizar el más conveniente, si bien lo hay que definir previamente.

También los juegos de símbolos de conectivas suelen variar con los autores. Para representar las operaciones que asociamos en este curso a $\vee$, $\wedge$, y $\neg$, se suelen utilizar respectivamente los siguientes juegos de símbolos dependiendo del entorno de trabajo: $\{O, Y, NO\}$, o su versión en inglés $\{OR, AND, NOT\}$, predomina en electrónica digital, $\{or, and, not\}$, predominante en programación, donde también se utiliza $\{|, \&, \neg\}$, o $\{+, ., \neg\}$, que también predomina en ambientes de matemáticas. Y muchos más conjuntos, mezclas de algunos de esos símbolos que se vieron con otros símbolos. No hay que recordarlos, simplemente estar alerta de que, según el entorno de trabajo, pueden variar los conjuntos de símbolos de referencia de las proposiciones y de las conectivas que hemos definido.

1.4.2. La sintaxis de las proposiciones

Una vez fijado un alfabeto, la sintaxis de las proposiciones; es decir, la construcción de proposiciones correctas, viene establecida por las siguientes reglas:

1. Los símbolos **V** y **F** que representan los valores de verdad *Verdadero* y *Falso*, respectivamente, son proposiciones elementales correctas.

2. Cada símbolo proposicional (es decir, los símbolos del alfabeto que se ha fijado) es una proposición elemental correcta.

3. Si P y Q son proposiciones correctas, entonces también los son:

   a) $P \vee Q$,
   b) $P \wedge Q$ y
   c) $\neg P$.

Ejemplos de proposiciones

Con el alfabeto por defecto 1:

- Los símbolos **V** y **F** son proposiciones.
- Los símbolos del alfabeto de referencia son proposiciones (y se suelen denominar elementales pero no necesariamente)
  Si A, B y C son proposiciones elementales, también son proposiciones:
- $A \wedge B$, $A \vee B$, $\neg A$,
- $(A \wedge B) \vee C$, $\neg(\neg C \wedge B)$, $\neg(\neg C))$.

Se observa en el último apartado la introducción de paréntesis para señalar las prioridades a la hora de aplicar los operadores, pues es evidente que $(A \wedge B) \vee C$ y $A \wedge (B \vee C)$ son distintas proposiciones. Pero hay que tener definido un ordenamiento implícito de la prioridad de los operadores para tener claros los casos en ausencia de paréntesis y para no tener que abusar de ellos hasta el punto de hacer ilegible la expresión de una proposición.

Por convenio, el orden de prioridad de los operadores por niveles de mayor prioridad a menor es el siguiente[1]:

1. $\neg$
2. $\wedge$
3. $\vee$

Según esas prioridades la siguiente proposición sin paréntesis: $A \vee B \wedge C$ es la misma que esta otra escrita con paréntesis -redundantes en este caso-: $A \vee (B \wedge C)$. Por supuesto, la proposición $(A \vee B) \wedge C$ no tendría nada que ver con ellas.

Si al construir una proposición por exigencia de su estructura es necesario romper esas prioridades se utilizarán paréntesis y, si fuesen necesarias,llaves y corchetes. El añadir paréntesis en una proposición cuando las prioridades de los operadores que intervienen en ella son coherentes con su estructura, y por tanto no son necesarios, no es ninguna incorrección; todo lo contrario, se suelen añadir muchas veces de forma redundante en aras de conseguir durante la lectura de la proposición una comprensión más rápida de su estructura.

[1]El orden de prioridad que aquí se expone no es el clásico en Lógica, sino el que rige sobre estos operadores en los lenguajes de programación, como es el caso de Python. Como a fin de cuentas en ambos casos son convenios, en vez de manejar dos diferentes y por simplificar hemos preferido quedarnos con uno solo, sin que ello suponga ninguna pérdida de generalidad

La proposición $\neg A \wedge B \vee C$ sería la misma, si añadimos paréntesis, que $(\neg A \wedge B) \vee C$. Pudo haberle cabido la duda al lector que la equivalente fuese: $\neg(A \wedge B) \vee C$, pero no es así. Reflexionemos; si estuviésemos operando la expresión de acuerdo a sus conectivas y las reglas de prioridades establecidas más arriba, cuando le llega el turno a la conectiva $\wedge$ ya habría operado antes -por ser la más prioritaria- la conectiva unaria $\neg$, por lo que *la proposición ya evaluada más pequeña inmediatamente a la izquierda de la* $\wedge$ sería $\neg A$ y no A simplemente, así que se realizaría la operación $(\neg A \wedge B)$ y finalmente $(\neg A \wedge B) \vee C$.

Otro alfabeto importante para las proposiciones

En la mayoría de las ocasiones en que nos será útil en este curso la lógica de proposiciones, el alfabeto que se tomará de referencia será del estilo a $\{3 \leq 5, x > 3, x < y, ...\}$, por lo que de acuerdo a ese alfabeto y a las reglas sintácticas anteriores, podremos construir proposiciones tales como:

1. $(x \neq y)$
2. $(x = y) \vee \neg(y < 1)$
3. $\neg(x \leq 5) \vee ((x < y) \wedge \neg(3 < x))$

Notemos cómo estas proposiciones conllevan implícitamente símbolos de relaciones conocidas en el entorno matemático: $=, \neq, <, >, \leq, y \geq$.

1.4.3. La semántica de las proposiciones

La semántica o significado de una proposición se reduce al valor de **Verdadero** o de **Falso**. Estos valores se denominan **valores de verdad** y son representados, respectivamente, por V y F.

Una interpretación **I** para una proposición constituida por proposiciones elementales $A_1, A_2, A_3, ...$ es una asignación de valores de verdad para cada símbolo de proposición elemental de dicha proposición; es decir, por ejemplo $\mathbf{I} = \{A_1 = V, A_2 = F, A_3 = V, ...\}$.

Dada una interpretación **I** para una proposición R cuyo valor de verdad se pretende averiguar bajo aquella, su semántica viene establecida por las siguientes reglas mediante las cuales se podrá obtener el *valor de verdad* de cualquier proposición:

1. El valor de verdad de **V** es siempre V -es decir, bajo cualquier interpretación- y el de **F** es siempre F.
2. El valor de verdad de toda proposición elemental de R viene dado por la interpretación **I**
3. Para obtener el valor de verdad de R se aplica el valor de verdad que le corresponde por la interpretación a cada una de las proposiciones elementales que la forman y se aplica reiteradamente las reglas semánticas determinadas por las tablas de verdad de la $\vee$ (figura 1.3.) de la $\wedge$ (figura 1.4.) y de la $\neg$ (figura 1.5.) Dichas tablas reflejan el comportamiento semántico de cada conectiva para todas las posibles combinaciones de valores de verdad de sus proposiciones componentes. En las tablas P y Q son proposiciones cualesquiera.

Ejemplos de evaluaciones de proposiciones:

Consideremos la proposición $A \vee (B \wedge \neg C)$ e I= $\{A = V, B = F, C = V\}$ una interpretación para ella (así es puesto que todas la proposiciones elementales que interviene en ella tiene valor de verdad asignado por la interpretación). ¿Cuál es el valor de verdad de la proposición bajo esta interpretación? Para calcular el valor de verdad de la proposición inicialmente se aplicarán los valores de verdad que asigna la interpretación a las proposiciones elementales y, seguidamente, para calcular

P	Q	P ∨ Q
V	V	**V**
V	F	**V**
F	V	**V**
F	F	**F**

Tabla 1.3: Tabla de verdad del operador lógico ∨.

P	Q	P ∧ Q
V	V	**V**
V	F	**F**
F	V	**F**
F	F	**F**

Tabla 1.4: Tabla de verdad del operador lógico ∧.

los valores de verdad de las subproposiciones que la componen se irá aplicando el comportamiento semántico dado en la tabla de verdad de las diferentes conectivas de la proposición, de acuerdo al orden establecido por los paréntesis, si los hubiera, y a las prioridades establecidas para esas conectivas.

El siguiente esquema detalla el proceso, de arriba a abajo, según la descripción anterior y plasmado en la tabla 1.6:

1. Se rellenan los valores de verdad que asigna la interpretación a las proposiciones elementales (fila 1 de la tabla citada)

2. Se colocan valores de verdad en la segunda fila bajo las conectivas que se operan en primer lugar de acuerdo a los paréntesis y a las prioridades de los operadores. El valor que se coloca será el que corresponde según la tabla de verdad para la conectiva correspondiente y según el/los valores de verdad que tiene/n la/s proposición/es que intervienen en la conectiva. (Recuérdese que el operador negación opera sobre una sola proposición y los otros dos sobre dos.)

3. Se continua con la tercera fila siguiendo el procedimiento anterior con los operadores del siguiente nivel de prioridad, conectando *subproposiciones* que en general serán cada vez mayores.

4. Se concluye con la fila que tenga un único valor, correspondiente a la conectiva de la proposición que ocupará última posición en la escala de prioridades. El valor de esa fila es el valor de verdad de la fórmula de partida: V en el ejemplo desarrollado en la tabla 1.5

A veces es interesante conocer cómo se comporta una proposición ante todas sus interpretaciones posibles. En ese caso se realiza una evaluación exhaustiva de la proposición contemplando todas las interpretaciones. Para ello primero conviene tener presente que si una proposición contienen n proposiciones elementales, el número de sus interpretaciones distintas es 2^n. Para cada uno de los dos valores de verdad que se le puede dar a una de sus proposiciones elementales se podrán dar otros dos a una segunda proposición elemental; es decir, entre esas dos proposiciones elementales ya se contabilizan 2 x 2 interpretaciones posibles y distintas. Pero si existe una tercera proposición elemental tendríamos 2 x 2 x 2, y si hubiese una cuarta ... Entonces, al considerar una proposición con n proposiciones elementales tendremos para aquella 2 x 2 x 2 ... x 2= 2^n interpretaciones posibles y distintas.

El procedimiento para realizar la evaluación exhaustiva de una proposición es nuevamente la tabla de verdad, en esta ocasión con tantas filas como interpretaciones distintas tenga la proposición

P	¬ P
V	F
F	V

Tabla 1.5: Tabla de verdad del operador lógico ¬.

A	∨	(B	∧	¬ C)
V	?	F	?	V
V	?	F	?	F
V	?	?	F	?
?	V	?	?	?

Tabla 1.6: Evaluación de una proposición.

que tratamos de examinar.

Como regla procedimental (obsérvese sobre la tabla 1.6), la primer fila se rellena inicialmente con las proposiciones elementales que intervienen en la proposición objetivo, que etiquetarán las cabeceras de otras tantas columnas. Comenzando por la primera columna, se rellena esta con V's en su primera mitad de posiciones y seguidamente con F's en su segunda mitad. La segunda columna se rellena en su cuarta parte con V's, la siguiente cuarta parte con F's y así, alternando, hasta completar la columna. La tercera columna de igual manera pero con las octavas partes. Y así sucesivamente hasta concluir con las demás columnas siguiendo el mismo procedimiento descendente en el tamaño de las partes. La última columna se rellenará con V's y F's alternándose. Observar la tabla dada como ejemplo.

Seguidamente se etiquetan nuevas columnas con las subproposiciones que resultan de considerar las operaciones de los distintos niveles de prioridad. En el caso de nuestro ejemplo, el primer nivel corresponde a: $\neg C$, el segundo nivel de prioridad a: $B \wedge \neg C$ y el tercer nivel a: $A \vee (B \wedge \neg C)$, que coincide ya con la fórmula de origen. A continuación se evalúan todas las proposicionales de las cabeceras de *menor a mayor tamaño* según la interpretación que encabeza cada fila. Observar que el orden de evaluación viene marcado por la necesidad de conocer el valor de verdad de dos proposiciones para evaluar una tercera que sea composición de aquellas por medio de un operador; por ejemplo, hay que tener evaluadas A y $(B \wedge \neg C)$ para llegar a evaluar $A \vee (B \wedge \neg C)$.

A	**B**	**C**	$\neg C$	$B \wedge \neg C$	$A \vee (B \wedge \neg C)$
V	V	V	F	F	V
V	V	F	V	V	V
V	F	V	F	F	V
V	F	F	V	F	V
F	V	V	F	F	F
F	V	F	V	V	V
F	F	V	F	F	F
F	F	F	V	F	F

Tabla 1.7: Tabla de verdad de una proposición.

Si la columna de la proposición objetivo en la tabla de verdad resultase rellena de V's en todas las filas, la proposición se calificaría como proposición **válida** o **tautología**; por otra parte, si estuviesen etiquetadas todas las filas con F's se calificaría como proposición **contradictoria** o simplemente

contradicción.

Evaluación de expresiones con símbolos relacionales

Anteriormente hemos hablado del alfabeto $\{3 \leq 5, x > 3, x < y, ...\}$ cuyos puntos suspensivos le dan un carácter infinito porque dependen números y variables cuyas combinaciones son en número infinito, y no es incorrecto hablar de se tipo de alfabeto y por tanto de ese tipo de proposiciones; si embargo, a la hora de evaluar estas proposiciones no se podrá por el método general anteriormente descrito.

La verdad de la proposición $3 < 5$ nos viene dada intuitivamente por la familiaridad que tenemos con la relación $<$ (*menor que*) y lo que significa aplicada a los enteros o reales. No necesitaremos de una interpretación explícita que le asigne un valor de verdad a la proposición; igual que si se dice *llueve ahora*, el valor de verdad en ese instante real es indiscutible según llueva o no llueva en ese momento, por lo que que mentalmente se le asigna un valor de verdad. Si se tiene $(3 < 5) \wedge (2 < 1)$, inmediatamente deducimos que la proposición es falsa apelando a esa interpretación implícita que todos conocemos para esa simbología: la primera proposición elemental es verdadera pero la segunda es falsa; consecuencia, la proposición principal es falsa. Hasta ahí todo correcto, pero si tuviésemos la *proposición* x<4 para evaluarla habría que subir irremediablemente en la escala de la Lógica al siguiente tipo del lógica: la Lógica de Predicados. Nosotros pasaremos de puntillas habilitando un método de evaluación de este tipo de expresiones por el que podremos obviar que estamos en esa lógica, *de predicados*, y mantenernos hablando de proposiciones.

La posibilidad de realizar tal evaluación viene dada por el hecho de que cuando tengamos la necesidad práctica de evaluar, por ejemplo, x<3 siempre conoceremos o al menos pensaremos en valores concretos del valor de x, o al menos delimitados por intervalos, luego finalmente estaríamos en una situación (o varias) semejante a las que evaluamos sin dudar cuando no había variables.

Consecuentemente **una interpretación para este tipo de proposiciones conllevará un valor concreto de las variables que intervengan en ella, en el dominio que corresponda** -enteros o reales-. Por ejemplo, para evaluar una proposición elemental de este tipo, tal como $x < 5$, necesitaremos una interpretación de la forma I=$\{x = 3\}$ bajo la cual la proposición resulta obviamente verdadera. No hace falta un valor de verdad para la proposición resultante, 3<5, tras sustituir x por 3, porque lo obtenemos nosotros mismos al realizar mentalmente la comparación habitual 3<5: V en este caso.

Cuando se trata de evaluar una proposición cualquiera de este tipo, compuesta de proposiciones elementales, simplemente una vez evaluadas las elementales se aplican las reglas de evaluación de las conectivas intervinientes siguiendo el método general de evaluación.

Ejemplo.

Evaluar $(x < 5 \wedge y \geq 6) \wedge A$ bajo la interpretación $I = \{x = 2, y = 6, A = V\}$. (El símbolo proposicional A se introdujo en el ejemplo para hacer notar que pueden construirse expresiones que conlleven los dos tipos de proposiciones estudiados). En ese caso la interpretación proporcionada tiene que contemplar una asignación de un valor de verdad a A, como así hace la del enunciado.

Sustituyendo en la expresión el valor de x y el de y por sus valores numéricos dados por la interpretación tenemos: $(3 < 5 \wedge y \geq 6) \wedge A$. El proceso que resta se aprecia en la tabla siguiente:

(3<5	∧	$6 \geq 6$)	∧	A
V	?	V	?	V
	V		?	V
			V	

Tabla 1.8: Evaluación de una proposición con símbolos relacionales.

1.4.4. Las leyes lógicas de las operaciones con proposiciones

Las leyes lógica consisten en igualdades semánticas destacables entre proposiciones; esto es, son igualdades entre proposiciones que aún teniendo una escritura diferente tendrán igual comportamiento ante cualquier interpretación. Dicho de otro modo, sus tablas de verdad serán idénticas. Tales proposiciones se denominan **equivalentes**. Si dos proposiciones P y Q son equivalentes escribiremos P = Q y se leerá *P equivale a Q*

En este curso por tratarse de un estudio informal de la lógica y para simplificar utilizaremos el símbolo = para denotar la equivalencia entre dos proposiciones, en contra de la simbología que suele utilizarse en estudios más formales de la lógica, en los que encontraremos, por ejemplo, $P \Leftrightarrow Q$, o $P \equiv Q$, etc., leyéndose en cualquiera de los casos *P equivale a Q*.

Las mismas leyes lógicas que veremos a continuación constituyen los ejemplos más característicos de proposiciones equivalentes.

Siendo P, Q y R proposiciones cualesquiera, las leyes lógicas que necesitaremos en este curso son:

Asociativas: $(P \vee Q) \vee R = P \vee (Q \vee R)$. Ídem para la $\wedge$

Conmutativas: $P \vee Q = Q \vee P$. Ídem para la $\wedge$

Distributivas: $(P \vee Q) \wedge R = (P \wedge R) \vee (Q \wedge C)$. Este caso se corresponde con la distributividad de la $\wedge$ respecto de la $\vee$. Ídem para la $\vee$ respecto de la $\wedge$. Por la propiedad conmutativa las distributivas se verifican por ambas manos: izq y derch.

Complementarios: $P \vee \neg P = \mathbf{V}$; $P \wedge \neg P = \mathbf{F}$

Leyes de la V y de la F:

- $\mathbf{V} \vee P = \mathbf{V}$; $\mathbf{V} \wedge P = P$
- $\mathbf{F} \vee P = P$; $\mathbf{F} \wedge P = \mathbf{F}$

Idempotencia: $\neg(\neg P) = P$

Leyes de De Morgan:

- $\neg(P \vee Q) = \neg P \wedge \neg Q$,
- $\neg(P \wedge Q) = \neg P \vee \neg Q$.

Ejemplo

Simplificar lo más posible la proposición $(A \wedge B) \vee \neg A)$.

Por simplificar se entiende obtener una proposición semánticamente equivalente mediante la aplicación discrecional de las leyes lógicas.

$(A \wedge B) \vee \neg A)$ = (por aplicación de la ley distributiva de la $\vee$ respecto de la $\wedge$ por la derecha)
$(A \vee \neg A) \wedge (B \vee \neg A)$ = (por aplicación de una ley de complementarios)
$\mathbf{V} \wedge (B \vee \neg A)$ = (por aplicación de una de las leyes de la **V**)
$(B \vee \neg A)$

Ejemplo
Considerar la proposición $\neg((A \wedge \neg B) \vee C$. Obtener una proposición que equivalga semánticamente a la anterior y tal que en ella las negaciones afecten exclusivamente a proposiciones elementales.

$\neg((A \wedge \neg B) \vee C$ = (por aplicación de una de las leyes de De Morgan en una ocasión)
$\neg(A \wedge \neg B) \wedge \neg C$ = (por aplicación de una de las leyes de De Morgan en otra ocasión)
$(\neg A \vee \neg\neg B) \wedge \neg C$ = (por aplicación de la ley de idempotencia -doble negación)
$(\neg A \vee B) \wedge \neg C$
Habiendo alcanzado una proposición conforme con lo solicitado.

Ejemplo
Simplificar lo más posible la proposición $\neg(\neg A \wedge (A \vee \neg B))$

$\neg(\neg A \wedge (A \vee \neg B))$ = (por aplicación de una de las leyes de De Morgan en una ocasión)
$\neg\neg A \vee \neg(A \vee \neg B)$ = (por aplicación de una de las leyes de De Morgan en una ocasión)
$\neg\neg A \vee (\neg A \wedge \neg\neg B)$ = (por aplicación de la ley de idempotencia en una ocasión)
$A \vee (\neg A \wedge \neg\neg B)$ = (por aplicación de la ley de idempotencia en una ocasión)
$A \vee (\neg A \wedge B)$ = (por aplicación de la ley distributiva por la derecha de la $\vee$ respecto de la $\wedge$ en una ocasión)
$(A \vee \neg A) \wedge (A \vee B)$ = (por aplicación de una de las leyes de los complementarios)
$\mathbf{V} \wedge (A \vee B)$= (por aplicación de una de las leyes de de la **V**)
$(A \vee B)$

1.4.5. Traducción de lenguaje natural al lenguaje de la Lógica de Proposiciones

Téngase presente que en este curso hemos omitido alguna conectiva importante, que si bien formalmente no es necesaria para poder realizar cualquier traducción que se plantee, sí facilitaría mucho la labor en ocasiones si dispusiésemos de ellas. En lo que sigue omitiremos enunciados de frases en español que conlleven en la traducción al lenguaje de la lógica de proposiciones el uso de esas conectivas.

La traducción del lenguaje natural al de la lógica de proposiciones se aprende con la experiencia; no obstante se pueden dar algunas pautas. Se recomienda seguir estos pasos:

1. Identificar las proposiciones elementales del lenguaje natural.
2. Fijar un alfabeto par representar las proposiciones detectadas
3. Identificar y clasificar las conectivas de la frase:
   - **y** (alternativamente puede aparecer: e, pero, no obstante, además, ...): $\wedge$,
   - **o** (alternativamente puede aparecer: al menos una de las dos es cierta, como mínimo una de las dos es cierta, ...): $\vee$,

- **no** (alternativamente puede aparecer: es falso que ..., no es cierto que ..., se niega que ...): $\neg$

4. Construir la proposición guardando el orden de actuación de las conectivas según el orden establecido en la frase en lenguaje natural. Es decir, construyendo las proposiciones más sencillas y componiéndolas con conectivas para obtener otra más compleja y así sucesivamente hasta obtener la traducción completa.

Ejemplo
Traducir al lenguaje de la Lógica de Proposiciones las frase:

No llora y ríe pero le duelen los dientes

Las proposiciones que aparecen en la frase son: *llora*, *ríe*, *duelen los dientes* se pueden identificar, por ejemplo, con las letras A, B y C respectivamente. Así que el alfabeto se fija como A, B, C. Las conectivas en el lenguaje natural son la *y* y el *pero*, que en los dos casos se identifican con la $\wedge$ de la lógica de proposiciones. Finalmente la traducción es:

$$A \wedge B \wedge C,$$

que a falta de paréntesis, por las prioridades de las conectivas, se identifica con $(A \wedge B) \wedge C$, que por la ley asociativa es equivalente a $A \wedge (B \wedge C)$; así pues en este caso, no tiene importancia el orden en que se apliquen.

Ejemplo
Traducir al lenguaje de la Lógica de Proposiciones las frase:

Llora y ríe o no come un caramelo pero le duelen los dientes

Se fija como alfabeto $\{A, B, C, D\}$, que se identifican:

- $A \equiv$ *llora*,
- $B \equiv$ *rie*,
- $C \equiv$ *come un caramelo* y
- $D \equiv$ *le duelen los dientes*.

Hay tres conectivas nítidas: *y*, *o*, *no*, y el *pero* que se corresponde también con la conjunción *y*, con lo que los símbolos correspondientes en lógica que se utilizarán son ya directos:$\wedge, \vee, \neg$ e $\wedge$. Al tratar de fijar las prioridades de esas partículas en el lenguaje natural, vemos que hay una ambigüedad: La primera en actuar es la negación, pero de la *y*, la *o* y el *pero* ,¿cuál actúa primero? No está claro por lo que necesitamos que se nos proporcione más información, por ejemplo escribiendo la frase así:

Llora y ríe, o no come un caramelo; pero además le duelen los dientes

El añadido de la puntuación -poco natural pero imprescindible en este caso- determina cómo proceder. Formaremos primero las proposiciones $A \wedge B$, $\neg C$. Después $(A \wedge b) \vee \neg C$. Y finalmente construimos la proposición final observando que la última conectiva en actuar es el *pero*.

$$((A \wedge B) \vee \neg C) \wedge D$$

Ejemplo
Traducir al lenguaje de la Lógica de Proposiciones las frase:

x no es menor que 3 pero y no es distinto de x

Se fija el alfabeto $\{x < 3$ e $y \neq x\}$. Fijarse que se toma como proposición elemental la parte que resta después de eliminar la negación, *no*, correspondiente de la frase: *x no es menor que 3* $\rightarrow$ *x es menor que 3* $\rightarrow x < 3$.
La proposición resultante es:

$$\{\neg(x < 3) \wedge \neg(y \neq x)\}.$$

Si hacemos uso de las relaciones complementarias podría escribirse de manera que no apareciese negaciones actuando sobre proposiciones elementales:

$$\{(x \geq 3) \wedge (y = x)\}$$

Observar que para realizar esta última transformación no se aplicó ninguna de las leyes lógicas formuladas anteriormente ni ninguna otra propiedad de la lógica que estamos introduciendo, sino que se hizo apoyándonos en nuestro conocimiento, mucho anterior, sobre las relaciones complementarias de las relaciones de comparación.

Ejemplo
Traducir al lenguaje de la Lógica de Proposiciones las frase siguiente, de manera que a) las negaciones, si aparecen, afecten exclusivamente a proposiciones elementales y b) posteriormente hacer desaparecer las que hubiese:

No es cierto que: x no es menor que 3 pero y no es distinto de x

Se puede observa que la frase anterior es simplemente la negación de la frase del último ejemplo, por lo que aceptemos que hemos llegado a la traducción de la parte de la frase en forma afirmativa, que coincidirá cualquiera de las formas que resultaron en el ejemplo anterior; cojamos por ejemplo la segunda:

$$(x \geq 3) \wedge (y = x)$$

Ahora solamente habrá que negarla y tendremos una primera traducción:

$$\neg((x \geq 3) \wedge (y = x))$$

Seguidamente, aplicando las veces necesarias las leyes de De Morgan, y si fuera necesario otras para simplificar los resultados, llegaríamos a la forma en la que las negaciones afectarían exclusivamente a las proposiciones elementales.

$$\neg(x \geq 3) \vee \neg(y = x)$$

Con lo que tendríamos concluido el apartado a) del ejemplo. Por último, haciendo uso de los conocimientos sobre la complementariedad de las relaciones de comparación se podría escribir como solución al apartado b):

$$(x < 3) \vee (y \neq x)$$

1.5. Ejercicios

1.5.1. Ejercicios resueltos

1. Expresar el número decimal 28 en binario
   - $28/2 = 14$, resto 0
   - $14/2 = \ 7$, resto 0
   - $7/2 = \ 3$, resto 1
   - $3/2 = \ 1$, resto 1
   - $1/2 = \ 0$, resto 1 $\quad \rightarrow 11100$

2. Expresar el número decimal -33 en binario usando 8 bits
   - $33/2 = 16$, resto 1
   - $16/2 = \ 8$, resto 0
   - $8/2 = \ 4$, resto 0
   - $4/2 = \ 2$, resto 0
   - $2/2 = \ 1$, resto 0
   - $1/2 = \ 0$, resto 1 $\quad \rightarrow 33$ es 100001 en binario
   - ahora hay que obtener el complemento a 2. Para ello:
     - rellenamos por la izquierda con ceros hasta completar 8 bits: 00100001
     - hallamos su complemento a 1 (cambiar unos por ceros y ceros por unos) 00100001 $\rightarrow$ 11011110
     - sumamos 1 al resultado anterior $11011110 + 1 = 11011111$

3. Expresar el número binario 1101110 en decimal
   - $1_6 1_5 0_4 1_3 1_2 1_1 0_0$
   - $1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
   - $64 + 32 + 0 + 8 + 4 + 2 + 0$
   - 110

4. Expresar el número binario 10110110111101 en hexadecimal
   - 10 1101 1011 1101
   - 0010 1101 1011 1101
   - 2DBD

5. Expresar el número hexadecimal $\texttt{3AFE4}_{16}$ en decimal
   - $3_4 A_3 F_2 E_1 4_0$
   - $3 \cdot 16^4 + A \cdot 16^3 + F \cdot 16^2 + E \cdot 16^1 + 4 \cdot 16^0$
   - $3 \cdot 65536 + 10 \cdot 4096 + 15 \cdot 256 + 14 \cdot 16 + 4$
   - $196608 + 40960 + 3840 + 224 + 4$
   - 241636

6. Expresar el número hexadecimal $\texttt{3AFE4}_{16}$ en binario
   - $3AFE4$
   - 00111010111111100100

7. Simplificar las siguientes expresiones explicando la ley lógica aplicada en cada paso:
   - $\neg((\neg(A \wedge B) \wedge C) \vee \neg(D \vee \neg A)) =$ (Aplicación de la ley de ....)
     - $\neg(\neg(A \wedge B) \wedge C) \wedge \neg\neg(D \vee \neg A) =$ (Aplicación de la ley de ....)
     - $(\neg\neg(A \wedge B) \vee \neg C) \wedge (D \vee \neg A) =$ (Aplicación de la ley de ....)
     - $((A \wedge B) \vee \neg C) \wedge (D \vee \neg A) =$ (Aplicación de la ley de ....)

- $(A \wedge B \wedge D) \vee (A \wedge B \wedge \neg A) \vee (\neg C \wedge D) \vee (\neg C \wedge \neg A)$ = (Aplicación de la ley de ....)
- $(A \wedge B \wedge D) \vee (\neg C \wedge D) \vee (\neg C \wedge \neg A)$ = (Aplicación de la ley de ....)
- $(A \wedge B \wedge D) \vee (\neg C \wedge (D \vee A))$

- $\neg(\neg(a < b \wedge a < c) \vee a >= 0)$ = (Aplicación de la ley de ....)
  - $\neg\neg(a < b \wedge a < c) \wedge \neg(a >= 0)$ = (Aplicación de la ley de ....)
  - $(a < b \wedge a < c) \wedge (a < 0)$
- $\neg((c > a \vee a > d) \wedge (c > b \vee b > d))$ = (Aplicación de la ley de ....)
  - $\neg(c > a \vee a > d) \vee \neg(c > b \vee b > d)$ = (Aplicación de la ley de ....)
  - $(\neg(c > a) \wedge \neg(a > d)) \vee (\neg(c > b) \wedge \neg(b > d))$ = (Aplicación de la ley de ....)
  - $((c <= a) \wedge a <= d)) \vee ((c <= b) \wedge (b <= d))$

1.5.2. Ejercicios propuestos

1. Complete la tabla 1.9 realizando las conversiones necesarias

Base 10	Base 16	Base 2
84		
	6F	
		11001111011
		10101010101
	3F5	
556		
100		
	FABADA	
		11111111000111000

Tabla 1.9: Completar la tabla con los valores que faltan en la correspondiente base.

2. Buscar el código ASCII de la 'A', 'B', 'F' y 'Z' y determinar el número de letras que hay entre la 'Z' y el resto

3. ¿Cómo llevaría de letra minúscula a mayúscula usando el código ASCII de los caracteres?

4. a) Simplifique las siguientes expresiones usando las leyes de De Morgan y otras leyes lógicas especificando en cada paso la ley lógica que se aplica. b) Enúnciese cada expresión en lenguaje natural utilizando, cuando sea posible, términos como *está en el intervalo ...*, o su negación, *no está el intervalo ...*, etc.

   - $\neg(\neg A \wedge (B \vee C))$
   - $\neg(x >= a \vee x <= b)$
   - $\neg(b >= c \vee a <= d)$

5. Obtener las tablas de verdad de las siguientes proposiciones:

   - $\neg A \vee A$
   - $\neg A \wedge A$
   - $\neg(A \vee B) \wedge \neg C$
   - $A \wedge (\neg A \vee B))$
   - $A \vee B \wedge C$

6. Expresar en el lenguaje de la lógica de proposiciones las siguientes frases; después, expresarlas de forma que no parezca la conectiva de negación en ellas -si es que inicialmente la contuviesen-, mediante la utilización de relaciones complementarias ($>$ de $\leq$ y viceversa, $=$ de $\neq$ y viceversa, ...):
   - *La x es menor o igual que 1 y mayor que 14;a su vez la y pertenece al intervalo [0,2).*
   - *La x esta entre 1 y 2 ambos límites inclusive, o entre -1 y 0 ambos exclusive.*
   - *La x o la y son menores que 2, y la x pero no la y es positiva o cero.*
   - *Al menos una de las dos, a, b, es cero.*
   - *A lo sumo una de las dos, a, b, es cero*
   - *Ninguna de las dos, a, b, es cero.*
   - *a o b son mayores que cero pero no ambas.*
   - *No es verdad que x sea mayor o igual que 3 e y sea 0, pero si es cierto que y es mayor que 0 y no menor que 14.*
   - *a y b son las dos cero o positivas pero al menos una es cero.*
   - *0 es mayor o igual que cero.* Pensar cuál seria la sintaxis más sencilla y compacta teniendo en cuenta la posible aplicación de las leyes lógicas.

Parte II

Introducción a la programación

CAPÍTULO 2

Introducción a la programación

2.1. Abstracción de problemas para su programación. Conceptos fundamentales

2.1.1. ¿Qué es un programa?

La habilidad más importante del ingeniero es la capacidad para **solucionar problemas**. La solución de problemas incluye poder formalizar problemas, pensar en la solución de manera creativa, y expresar esta solución en términos que un ordenador pueda entender. La solución de problemas mediante el uso del ordenador se consigue a través de los **programas**.

> Definición
>
> Un *programa* es un texto con una secuencia de instrucciones que un ordenador puede interpretar y ejecutar.

Los programas son, por tanto, el medio de comunicación con el ordenador. Mediante su utilización conseguimos que las máquinas aprovechen sus capacidades para resolver los problemas que nos interesan. Sin los programas los ordenadores no son capaces de hacer nada.

2.1.2. Lenguajes de programación

Los programas se escriben en lenguajes especializados llamados **lenguajes de programación**. Hay muchos lenguajes de programación, pero para programar no es necesario conocerlos todos. En esencia, la técnica básica necesaria para programar es común a todos los lenguajes.

> Definición
>
> Un *lenguaje de programación* es un idioma formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen la estructura y el significado de las instrucciones de que consta el lenguaje.

Cuando escribimos un programa utilizando un determinado lenguaje de programación llamamos **código fuente**, o simplemente **código**, al texto del programa. Cuando un ordenador lleva a cabo una tarea indicada por un programa, decimos que **ejecuta** el código.

Aunque no vamos a entrar en los detalles de cada uno, es necesario mencionar que a la hora de programar se pueden seguir diversas técnicas o **paradigmas de programación**: programación imperativa, declarativa, estructurada, modular, orientada a objetos, etc. Los lenguajes de programación suelen soportar varios de estos paradigmas de programación.

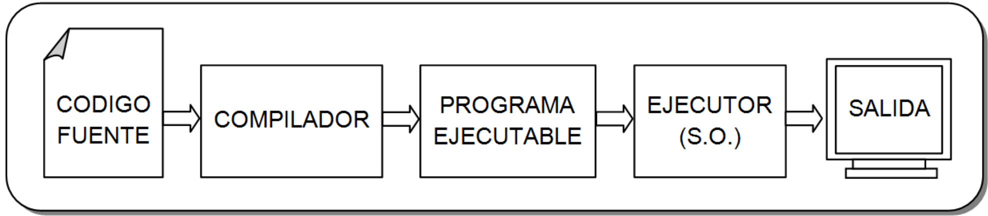
Independientemente del lenguaje que se utilice, es importante que el alumno se acostumbre a seguir los principios de la **programación modular**, que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

En general, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación.

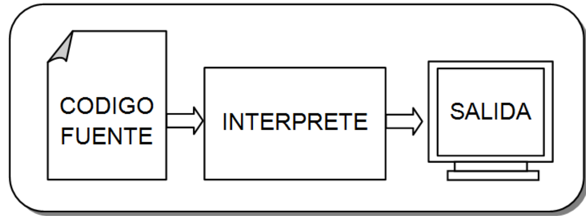
En la actualidad los lenguajes de programación, llamados de **alto nivel**, están pensados para que los programas sean comprensibles por el ser humano. Sin embargo, el código fuente de los mismos no es comprendido por el ordenador, ya que éste sólo maneja el **lenguaje binario** o **lenguaje máquina**.

Es necesario hacer una **traducción** de los programas escritos en un lenguaje de programación de alto nivel a lenguaje máquina. Hay dos tipos diferentes de lenguajes dependiendo de la forma de hacer esta traducción:

- **Lenguajes compilados** El *Compilador* realiza una traducción completa del programa en lenguaje de alto nivel a un programa equivalente en lenguaje máquina. Este programa resultante, **programa ejecutable**, se puede ejecutar todas las veces que se quiera sin necesidad de volver a traducir el programa original.



- **Lenguajes interpretados** En este caso, el *Intérprete* va leyendo el programa en lenguaje de alto nivel instrucción a instrucción, cada una de ellas se traduce y se ejecuta. No se genera un programa directamente ejecutable.



El enfoque compilado genera código ejecutable que utiliza directamente las instrucciones nativas de la CPU. Esto suele implicar que el programa se ejecutará mucho más velozmente que si se hiciera en un lenguaje interpretado. A cambio, presenta el inconveniente de que el ejecutable resultante de la compilación sólo sirve para ser ejecutado en una CPU concreta y un Sistema Operativo concreto (aquellos para los que se compiló), o bien versiones compatibles de la CPU y el operativo, mientras que si se hiciera en un lenguaje interpretado podría ser ejecutado en cualquier arquitectura y operativo para los que exista el intérprete.

2.1.3. Programas y algoritmos

Cuando se aprende el primer lenguaje de programación se piensa que la parte difícil de resolver un problema con un ordenador es escribir el programa siguiendo las reglas del lenguaje. Esto es

totalmente falso. La parte más difícil es encontrar el método de resolución del problema. Una vez encontrado este método, es bastante sencillo traducirlo al lenguaje de programación necesario: `Python`, `C++`, `Java` o cualquier otro.

Definición

Se denomina *algoritmo* a una secuencia no ambigua, finita y ordenada de instrucciones que han de seguirse para resolver un problema.

Importante

Un programa de ordenador es un algoritmo escrito en un lenguaje de programación.

En un algoritmo siempre se deben de considerar tres partes:

- **Entrada**: información dada al algoritmo.
- **Proceso**: operaciones o cálculos necesarios para encontrar la solución del problema.
- **Salida**: respuestas dadas por el algoritmo o resultados finales de los procesos realizados.

Los algoritmos se pueden expresar de muchas maneras: utilizando lenguajes formales, lenguajes algorítmicos (pseudocódigo) o mediante el lenguaje natural (como el castellano). Esta última es la forma más adecuada para la asignatura que nos ocupa.

Ejemplo: Algoritmo para calcular el área de un triángulo

1. Obtener la *base* del triángulo
2. Obtener la *altura* del triángulo
3. Hacer la operación $(base \times altura)/2$
4. Mostrar el resultado

Este algoritmo de resolución del área del triángulo es independiente del lenguaje de programación que vayamos a utilizar. Cualquier programa que quiera calcular el área de un triángulo escrito en cualquier lenguaje de programación debe seguir estos pasos.

Terminología

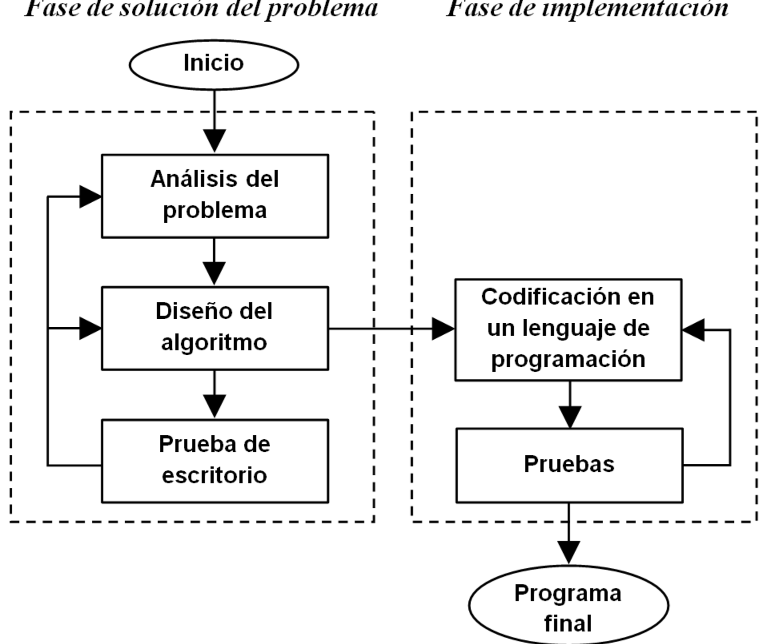
Dos programas distintos codificados en el mismo o en distinto lenguaje de programación y que resuelven el mismo problema con la misma secuencia de pasos, se dice que son *implementaciones* del mismo algoritmo.

2.1.4. Diseño de programas

Aunque el diseño de programas es un proceso creativo, hay que seguir una serie de pasos. Nadie empieza a escribir un programa directamente sin estudiar el problema que se quiere resolver.

Todo el proceso de diseño de un programa se puede dividr en dos fases:

- **Fase de resolución de problemas**. El resultado de esta primera fase es un algoritmo, expresado en español, para resolver el problema.
- **Fase de implementación**. A partir del algoritmo obtenido en la fase anterior, se codifica y se prueba el programa final.



Importante

Muchos programadores noveles no entienden la necesidad de diseñar un algoritmo antes de escribir un programa en un lenguaje de programación. La experiencia ha demostrado que la resolución de problemas en dos fases da lugar a programas que funcionan correctamente en menos tiempo.

Esta forma de diseñar programas requiere la realización de una serie de tareas:

- **Análisis del problema**. A partir de la descripción del problema, expresado habitualmente en lenguaje natural, es necesario obtener una idea clara sobre qué se debe hacer (**proceso**), determinar los datos necesarios para conseguirlo (**entrada**) y la información que debe proporcionar la resolución del problema (**salida**).
- **Diseño del algoritmo**. Se debe identificar las tareas más importantes para resolver el problema y disponerlas en el orden en el que han de ser ejecutadas. Los pasos en una primera descripción pueden requerir una revisión adicional antes de que podamos obtener un algoritmo claro, preciso y completo.
- **Transformación del algoritmo en un programa (codificación)**. Solamente después de realizar las dos etapas anteriores abordaremos la codificación de nuestro programa en el lenguaje de programación elegido.
- **Ejecución y pruebas del programa**. Se debe comprobar que el programa resultante está correctamente escrito en el lenguaje de programación y, más importante aún, que hace lo que realmente debe hacer.

2.1.5. `Python`

`Python` es un lenguaje de alto nivel como pueden ser el C, C++ o el Java. Entonces, ¿por qué hemos escogido precisamente `Phyton` en este curso? Python presenta una serie de ventajas que lo

hacen muy atractivo, tanto para su uso profesional como para el aprendizaje de la programación. Entre ellas podemos destacar las siguientes:

- La sintaxis es muy *sencilla* y cercana al lenguaje natural, lo que facilita tanto la escritura como la lectura de los programas.
- Es un lenguaje muy *expresivo* que da lugar a programas compactos, bastante más cortos que sus equivalentes en otros lenguajes.
- Es un lenguaje de programación *multiparadigma*, que permite al programador elegir entre varios estilos: programación orientada a objetos, programación imperativa (y modular) y programación funcional.
- Es *multiplataforma* por lo que podremos utilizarlo tanto en Unix/Linux, Mac/OS o Microsoft Windows.
- Es un lenguaje interpretado y por tanto *interactivo*. En el entorno de programación de `Python` se pueden introducir sentencias que se ejecutan y producen un resultado visible, que puede ayudarnos a entender mejor el lenguaje y probar los resultados de la ejecución de porciones de código rápidamente.
- Python es *gratuito*, incluso para propósitos empresariales.
- Es un lenguaje que está creciendo en popularidad. Algunas empresas que utilizan Python son Yahoo, Google, Disney, la NASA, Red Hat, etc.

Para saber más

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90 cuyo nombre está inspirado en el grupo de cómicos ingleses *Monty Python*. Toda la información necesaria sobre el lenguaje la puedes encontrar en `http://www.python.org/`.

Python permite dos modos de interacción con el lenguaje:

1. Escribir comandos directamente en el intérprete. En este modo, cada línea que escribimos es ejecutada al pulsar el retorno de carro, y el resultado de la ejecución se muestra inmediatamente, a continuación. Es útil para probar ejemplos simples y ver inmediatamente el resultado, y para experimentar con diferentes expresiones para comprender mejor sus diferencias y su funcionamiento.
   El inconveniente es que todo lo que se va escribiendo, se pierde cuando se cierra el intérprete.
2. Escribir un programa completo en un editor de textos, guardar el programa en un fichero y después usar el intérprete para ejecutarlo.
   En este caso no se ejecuta nada mientras se escribe, sino sólo cuando se “lanza” el intérprete para que ejecute el programa completo que está en el fichero. Entonces el intérprete irá leyendo línea a línea del fichero y las irá ejecutando. No se muestra ningún resultado por pantalla, a menos que el programa contenga la orden `print` que sirve precisamente para esto (como veremos en su momento).

A lo largo de este documento, presentaremos ejemplos en ambos formatos. Si se trata de ejemplos muy breves encaminados sólo a demostrar alguna característica del lenguaje, usaremos el intérprete directo. Si se trata de ejemplos más largos que tienen interés suficiente para merecer ser guardados en un fichero, usaremos el editor.

El lector puede diferenciar si usamos el intérprete o el editor, porque el estilo en que se muestra el código es diferente en cada caso. Cuando se usa el intérprete, puede verse el símbolo >>> delante de cada línea que el usuario teclea, y la respuesta del intérprete en la línea siguiente:

```
>>> "Ejemplo ejecutado en el interprete"
'Ejemplo ejecutado en el iterprete'
```

En cambio, cuando se muestra código para ser escrito en el editor, no se ve el resultado de su ejecución. Además las líneas están numeradas para facilitar el referirse a ellas si es necesario (el alumno no debe copiar estos números cuando escriba el código en el editor).

```
1 # Ejemplo para ser escrito en el editor
2 print "Ejemplo"
```

2.2. Variables, expresiones, asignación

2.2.1. Valores

El principal motivo por el que se escriben programas de ordenador es para manipular la información de una forma más eficiente y segura. Desde el punto de vista de la información, los programas manejan típicamente los datos del siguiente modo:

1. Reciben datos de entrada por parte de los usuarios,
2. procesan esos datos, haciendo cálculos y operaciones con ellos, y
3. producen resultados que son útiles para los usuarios.

Los resultados pueden ser tanto valores calculados (por ejemplo, en una aplicación de cálculo de estructuras, obtener las dimensiones que debe tener una viga), como obtener un informe impreso.

Aunque la información que manejan los programas puede parecer muy amplia, ya que existen aplicaciones informáticas para propósitos muy diversos, en realidad la información está formada habitualmente por la agregación de tres tipos de de datos básicos:

- **números**: `7`, `3.14165`
- **textos**: “Juan”, “Alonso”, “Universidad de Oviedo”
- **valores lógicos**: `True`, `False`

Combinando estos elementos se puede representar la información básica que usan los programas. Por ejemplo, si se quiere hacer una aplicación para gestionar los datos de cada cliente de un banco, necesitaremos guardar su nombre y apellidos (textos), el saldo de sus cuentas (números) o si el cliente es preferente o no (valor lógico). Los valores lógicos tal vez parezcan menos útiles, sin embargo es casi lo contrario, no solamente permiten representar aquella información que puede ser cierta o falsa, sino que además son los datos fundamentales para controlar el flujo de ejecución de los programas (como se verá más adelante, las sentencias condicionales y los bucles se basan en la manipulación de valores lógicos).

2.2.2. Tipos

Cada dato o valor que se maneja en un programa es de un **tipo**. Python detecta de qué tipo es cada dato por su *sintaxis*. Una secuencia de dígitos que no contenga el punto (`.`) se considera un entero. Si contiene el punto se considera un real (que python llama `float`). Una secuencia arbitraria de caracteres delimitados por el carácter de comillas dobles (`"`), o de comillas simples (`'`) es considerado una cadena de texto[1]. Las palabras especiales `True` y `False` se consideran datos de tipo booleano.

- `7` es de tipo `int` (de *integer*), es un número entero
- `3.14165` es de tipo `float`, un número en coma flotante (real)

[1] Para delimitar las cadenas se puede usar indistintamente la comilla simple o la doble, pero es forzoso usar el mismo tipo de comilla para abrir y para cerrar.

- "Juan", es de tipo `str` (de *string*), es una cadena de caracteres
- `True` es de tipo `bool` (de *booleano*), un valor de verdad

Los tipos definen conjuntos de valores que los programas pueden manejar y las operaciones que se pueden hacer con ellos. Por ejemplo, los programas pueden utilizar, gracias al tipo `float`, el conjunto de los números reales y podemos sumar dos reales. En concreto, el rango de los números reales que se pueden utilizar es de ±1.7e±308 que viene dado por el número de bytes (8) que se utilizan para representarlos[2].

Para saber más

Además del tipo `int`, python posee otro tipo entero, `long`. Normalmente se manejan los valores enteros como `int` ya que este tipo permite representar valores del orden de millones de billones. Si alguna vez un programa necesita guardar un valor entero aún mayor, entonces se representará automáticamente (sin que el programador deba hacer nada) como un `long`. La capacidad de representación de un `long` está sólo limitada por la cantidad de memoria libre, por lo que en la práctica puede representar valores enteros extremadamente grandes.

Para saber el tipo de un valor, se puede utilizar la función `type`. Simplemente hay que indicar el valor del que se quiere conocer su tipo:

```
>>>> type(7)
<type 'int'>
>>> type(3.1416)
<type 'float'>
>>> type("Juan")
<type 'str'>
>>> type(True)
<type 'bool'>
```

Como se puede apreciar, se imprime en la consola el tipo de cada uno de los valores. En realidad esto no es muy útil a la hora de escribir un programa, ya que normalmente el programador sabe de qué tipo es cada uno de los datos que el programa maneja.

2.2.3. Conversiones entre tipos

A veces los programas reciben valores que son de un cierto tipo y necesitan convertirlos en valores de otro tipo antes de hacer operaciones con ellos. Esta situación se produce, por ejemplo, con datos que se reciben como string pero que representan números; antes de hacer cálculos con ellos es necesario convertirlos en un valor numérico. La forma de hacer esas conversiones es bastante sencilla: primero hay que indicar el nombre del tipo al que se quiere convertir el valor, y posteriormente entre paréntesis el valor que se quiere convertir. Por ejemplo:

```
>>> int("7")
7
>>> str(7)
'7'
>>> float("3.1416")
3.1415999999999999
>>> str(3.1416)
'3.1416'
>>> int(3.1416)
3
>>> float(7)
7.0
```

[2] Cuando un número real se sale de ese rango se considera $\pm\infty$

En el primer ejemplo se convierte en `int` la cadena de caracteres formada solamente por el dígito '7'. El valor que se devuelve es el entero `7`, que obviamente no es lo mismo la cadena que contiene el dígito '7', ni en cuanto al dato en sí mismo, ni sobre todo a las operaciones que se van a poder hacer con él (la cadena con el dígito '7' no se podrá dividir por otro valor, mientras que el valor entero sí). El ejemplo contrario se da en la siguiente instrucción. En este caso se convierte en cadena (`str`) el valor entero `7`. El valor que se devuelve es ahora una cadena (está entre comillas) formada por el dígito '7'. Un caso análogo se da cuando se convierte números reales en cadenas y viceversa. De nuevo no es lo mismo el número real que la cadena formada por los dígitos que componen dicho número.

En los dos ejemplos finales, se ve cómo se pueden convertir números reales en enteros y al revés. Cuando se convierte un número real en entero, se devuelve la parte entera (equivale a truncar) y cuando se convierte un entero en real, la parte decimal vale `0`.

2.2.4. Operadores

Los programas usan los valores que manipulan para hacer cálculos y operaciones con ellos. La forma más sencilla de hacer operaciones con los datos es empleando los operadores del lenguaje. Cada uno de los tipos descritos anteriormente permite hacer operaciones diferentes con sus valores. Antes de estudiar los operadores más utilizados, es interesante ver unos ejemplos iniciales. En todos estos ejemplos se usan operadores binarios (tienen dos operandos) y su sintaxis es siempre la misma: *operando* `operador` *operando*:

```
>>> 7 + 4
11
>>> 3.1416 * 2
6.2831999999999999
>>> "Juan" + " Alonso"
'Juan Alonso'
>>> True and False
False
```

En las instrucciones anteriores se han usado operadores binarios: el operador suma (+) para sumar dos enteros, el operador producto (`*`) para multiplicar un real por un entero, de nuevo el operador + para concatenar dos cadenas, y por último el operador y-lógico (`and`) para hacer dicha operación entre dos valores lógicos.

El primer detalle importante que se debe observar es que todas las operaciones producen un valor: cuando sumamos `7` y `4` el resultado es el valor entero `11`, o cuando concatenamos las cadenas "Juan" y " Alonso" se produce la cadena "Juan Alonso". El segundo aspecto fundamental es entender qué hace cada operador. En los dos ejemplos anteriores el mismo operador + se comporta de manera diferente si se suman enteros que si se utiliza para concatenar cadenas. Por ello, se deben conocer los operadores que existen en el lenguaje y las operaciones que hace cada uno de ellos.

Aunque hay más operadores de los que se van a citar a continuación, los más utilizados y con los que se pueden hacer la gran mayoría de los programas están agrupados en tres grandes grupos:

- Aritméticos: + (suma), − (resta), `*` (producto), `/` (división), `%` (resto) y `**` (potencia). También hay un operador unario (que actúa sobre un solo dato, en lugar de dos) que es el − para cambiar de signo al dato que le siga.
- Relacionales: == (igual), `!=` (distinto), < (menor que), <= (menor o igual que), > (mayor que) y >= (mayor o igual que)
- Lógicos: `and` (y-lógico), `or` (o-lógico) y `not` (no-lógico)

Los operadores aritméticos se emplean habitualmente con los tipos numéricos y permiten hacer las operaciones matemáticas típicas. El único operador que tiene un comportamiento diferente según

los operandos sean `int` o `float` es el operador división (/): cuando recibe enteros hace una división entera, el cociente será un valor entero, y con valores reales la división es real. Es un buen ejemplo de que el resultado de una expresión que use un operador depende de los operandos que se empleen, en este caso basta con que uno de los dos operandos sea un número real para que la división sea real.

Para saber más

Existen dos variantes del lenguaje Python. La que explicamos en esta asignatura se denomina Python 2, la otra (más reciente pero aún menos extendida) se denomina Python 3.

En Python 3, el operador / se comporta de forma diferente a como se ha descrito, ya que siempre realiza la división real (flotante), incluso si los operandos son enteros. Para el caso en que se quiera realizar la división entera se tiene el operador // (que también existe en Python 2). Esto suele resultar más claro para los principiantes.

Si quieres que Python 2 se comporte como Python 3 en lo que respecta a la división, puedes poner al principio de tus programas una línea que diga:

```
1 from __future__ import division
```

Sin entrar a explicar la curiosa sintaxis de esta línea, basta saber que si la pones, entonces el operador / realizará siempre la división real, al igual que en Python 3.

Algunos operadores aritméticos, en concreto el operador + y el operador *, se pueden utilizar también con cadenas de caracteres[3]. Lo que producen son, respectivamente, la concatenación de dos cadenas, y la concatenación de una misma cadena varias veces (luego se verá algún ejemplo de esta operación).

Los operadores relacionales sirven para determinar si es cierta o no una relación de orden entre dos valores, o si éstos son iguales o distintos. Por ejemplo, podemos comprobar si `3 <= 5`. Intuitivamente es natural pensar que una relación de orden o igualdad puede ser cierta o no, es decir, el resultado de esa operación será un valor lógico siempre, independientemente de si los valores son números o cadenas de caracteres. Los valores de tipo `str` también se pueden comparar, el valor que se produce depende de su ordenación alfabética, distinguiéndose entre mayúsculas y minúsculas.

Por último, se pueden hacer las típicas operaciones lógicas “y”, “o” y “no”. De nuevo en este caso el valor que se produce es de tipo `bool` ya que la operación podrá ser cierta o falsa. Estos operadores se emplean fundamentalmente al escribir las condiciones de las sentencias condicionales y los bucles que se estudiarán más adelante.

La mejor forma de comprender los operadores es jugar con ellos usando el intérprete, introduciendo valores de distintos tipos y viendo el resultado que producen las expresiones resultantes. Algunos ejemplos descriptivos podrían ser los siguientes:

```
>>> 2 / 3
0
>>> 2 % 3
2
>>> 2.0 / 3
0.66666666666666663
>>> 2 ** 3
8
>>>> "Pe" * 4
'PePePePe'
>>> not True
```

[3]También con listas, como se indicará en el apartado dedicado a ellas

```
False
>>> 2 > 3
False
>>> 3 <= 3.1416
True
>>> 2 == 3
False
>>> 2 != 3
True
>>> "Antonio" < "Juan"
True
```

En el primer ejemplo se hace una división entre dos enteros, el resultado es el cociente, en ese caso `0`. Si hacemos el resto (`%`) de esa misma división, el valor obtenido es `2`. Cuando uno de los operandos es un número real, como pasa en el tercer ejemplo, entonces la división es real, produciendo un cociente que es un valor `float`. Aunque en otros lenguajes no existe este operador, en python es posible calcular la potencia de dos números con el operador `**`. Una operación más extraña es la que se muestra en el quinto ejemplo. Se utiliza el operador `*` con una cadena de caracteres y un número entero. Lo que hace es producir una cadena en la que se repite la cadena que actúa como primer operando tantas veces como indica el valor entero del segundo operando. En el ejemplo, la cadena "Pe" se repite `4` veces, produciendo la cadena "PePePePe".

El resto de operaciones son todas con operadores relacionales y lógicos, y por ello todas producen un valor `bool`. La primera es la única que utiliza un operador unario[4], esto es, un operador que solamente necesita un operando. En el ejemplo se hace la operación `not` con el valor `True`, y produce el valor lógico contrario, es decir, `False`. El resto de operaciones manejan operadores relacionales y producen un resultado cierto o falso en función de que la relación de orden o igualdad se cumpla o no: `2` no es mayor que `3` (falso), `3` sí es menor o igual que `3.1426` (cierto), `2` no es igual que `3` (falso) y `2` sí es distinto de `3` (cierto). También el orden puede comprobarse entre cadenas, como muestra el último ejemplo. Se trata de un orden alfabético en este caso, la cadena "Antonio" va antes, y por tanto es menor que la cadena "Juan", si ordenáramos ambas cadenas alfabéticamente.

Importante

Todas las operaciones que se hacen con operandos siempre producen un resultado. Ese resultado será un valor de un cierto tipo. El programador debe conocer siempre el tipo que producirá una expresión sabiendo el tipo de sus operandos y la operación que realiza el operador.

2.2.5. Variables

Como se dijo al principio de esta sección, los programas se escriben para, dados unos datos de entrada, hacer cálculos con ellos, y producir unos resultados de salida. Para que los programas sean útiles necesitamos un mecanismo que nos permita representar esa información que va a cambiar: los datos de entrada que serán diferentes en cada ejecución y los datos de salida que tomarán valores distintos en función de los datos de entrada recibidos. No nos sirve en este caso con usar valores; los valores concretos, como `7` o `3.1416` no cambian. La forma de representar toda esa información que *varía* es a través de variables.

Supongamos que necesitamos hacer un programa que sirva para calcular el área de un rectángulo, dados los valores de su base y su altura. Sabemos que el cálculo del área de un rectángulo se realiza mediante la ecuación:

$$área = base * altura$$

[4]También el operador - puede actuar como operador unario, cambiando el signo del operador numérico que le siga

Los usuarios de nuestro programa quieren poder ejecutarlo las veces que deseen, introducir de alguna forma (generalmente con el teclado) los valores de la base y la altura, y que el programa muestre el resultado del área. Por ejemplo, si el usuario quiere calcular el área de un rectángulo de base `5` y altura `4`, introducirá esos dos datos y nuestro programa debería responder que el área es `20`. Sin embargo, dentro de las instrucciones de nuestro programa no debemos escribir la operación `5*4` porque no sabemos qué valores va a introducir el usuario cuando lo ejecute. Además, queremos que nuestro programa calcule el área de *cualquier* rectángulo. Si el usuario introdujera `6` y `3`, entonces el área sería `18`.

La forma de generalizar la operación de calcular el área dentro del programa es idéntica a la manera en la que se expresa la ecuación que explica su cálculo. En lugar de usar valores concretos, emplearemos nombres de variables para referirnos a esos valores que cambian. Diseñaremos nuestro programa usando al menos dos variables, `base` y `altura`, que tomarán distintos valores en las diferentes ejecuciones, pero la forma de calcular el área del rectángulo que representan es siempre la misma:

```
>>> base * altura
```

cuando `base` o `altura` cambien de valor, también cambiará el producto de ambas y con ello el valor del área que calcula nuestro programa.

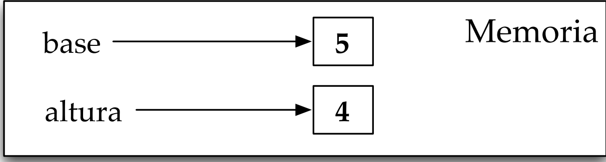
Definición

Una variable es el nombre que se utiliza en un programa para representar un dato o valor que puede cambiar.

Veamos el funcionamiento de las variables con un ejemplo práctico. En primer lugar, damos un valor a nuestras dos variables `base` y `altura`, en concreto los valores `5` y `4` respectivamente:

```
>>> base = 5
>>> altura = 4
```

Hemos usado dos asignaciones. En el siguiente apartado explicaremos con detalle cómo funciona una asignación. Intuitivamente, lo que hace la asignación es cambiar el valor de una variable. Es decir, ahora en nuestro ejemplo cuando usemos el nombre `base`, su valor será `5` y cuando usemos `altura` `4`. Las variables vienen a ser una forma de referirnos a un valor que está en la memoria usando un nombre. En memoria tenemos los valores `5` y `4`, el `5` es el valor de la `base` del rectángulo y el `4` su `altura`:



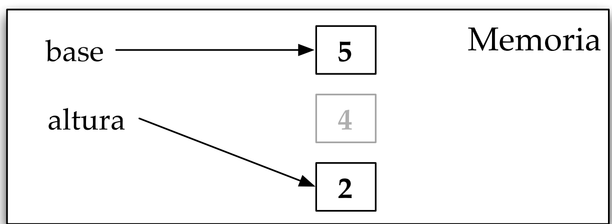
Cuando queremos utilizar estos datos para hacer alguna operación simplemente tenemos que poner su nombre. Así para calcular el área escribiremos:

```
>>> base * altura
20
```

El resultado es naturalmente `20` en función del los valores que en ese momento tienen la `base` y la `altura`. Sin embargo, a lo mejor el usuario del programa quiere calcular el área de otro rectángulo distinto, por ejemplo uno que tenga `base` `5` y `altura` `2`. Cambiaríamos el valor de la `altura`, pero la operación para calcular el área sería exactamente la misma, produciendo el resultado deseado:

```
>>> altura = 2
>>> base * altura
10
```

El dato `altura` ha cambiado y con él el resultado del área. En memoria, la variable `altura` representa el valor `2` en lugar del valor `4`. Ahora nuestro programa está usando los valores `5` y `2`, el `4` ya ha dejado de usarse (por eso aparece en gris en la figura siguiente).



Cada vez que se utiliza un valor nuevo en un programa en python, se reserva un nuevo espacio en la memoria para guardarlo. El valor de nuestro dato `altura` ha cambiado, está en otro sitio en la memoria, pero la forma de referirnos a él es la misma, usando el nombre de la variable.

Importante

Una variable es un nombre con el que representamos un valor, y por tanto, será de un tipo. Ese valor además se guardará en una posición de la memoria.

La variable `base` representa el valor `5`, es de tipo `int` y dicho valor está en una posición de la memoria concreta. Con el nombre de la variable podremos acceder al valor tantas veces como queramos y la variable podrá cambiar de valor tantas veces como se desee. Para conocer el tipo del valor que representa la variable, algo que habitualmente no es necesario, se puede utilizar la función `type` exactamente como se hacía con los valores anteriormente.

Para saber más

La posición de la memoria donde se encuentra el valor que representa una variable es intrascendente para la ejecución del programa; en ejecuciones distintas, el valor se situará en posiciones de la memoria diferentes. Si se quiere conocer la posición del valor que representa una variable se debe usar la función `id`, indicando entre paréntesis el nombre de la variable, p.e., `id(base)`.

2.2.6. Asignación

Una de las instrucciones más importantes para la construcción de programas es la asignación. La asignación es la instrucción que nos permite cambiar el valor de una variable. La sintaxis es sencilla, primero se indica el nombre de la variable, después el operador = y por último la expresión del valor que se desea asignar:

$$variable = expresión$$

El efecto que tiene una asignación es diferente en función de cómo sea la expresión que aparece en la parte derecha de la asignación. Hay dos opciones:

1. si se asigna un valor (o en general, el resultado de una operación), entonces se reserva un nuevo espacio en la memoria para contenerlo

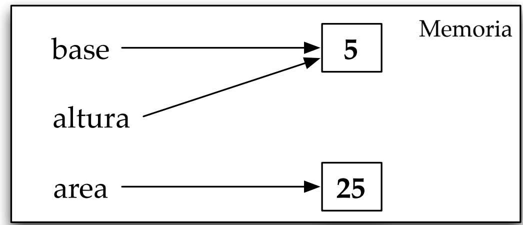
2. si se asigna otra variable, entonces las dos variables se referirán al mismo valor

Es decir, a efectos de la memoria que ocupan los datos del programa hay diferencias entre ambas situaciones. Vamos a verlo con un par de ejemplos, utilizando de nuevo el problema del cálculo del área de un rectángulo, nuestras dos variables `base` y `altura` y una nueva variable `area` con la que guardaremos mediante asignaciones el valor del área del rectángulo que vayamos calculando.

Analicemos en primer lugar las siguiente asignaciones:

```
>>> base = 5
>>> altura = base
>>> area = base * altura
>>> area
25
```

En la primera de ellas se asigna a la variable `base` el valor `5`. Como se está asignando un nuevo valor, en memoria se reserva el espacio para contener el valor `5`, valor al que nos referiremos usando la variable `base`. A continuación se asigna a la variable `altura` la variable `base`. Ahora estamos en el segundo caso antes descrito, se está asignando una variable. Por tanto, no se reserva un nuevo espacio en memoria, sino que las dos variables se refieren al mismo valor, `5`. Por último, a la variable `area` se le asigna el resultado de calcular el área del rectángulo, `base * altura`. Como es una operación, produce un nuevo valor, `25`, y estamos de nuevo en el primer caso. Por ello se crea una nueva posición de memoria para contener el valor `25`. Gráficamente la situación de la memoria tras estas asignaciones sería la siguiente:



Supongamos que ahora se hicieran las asignaciones siguientes:

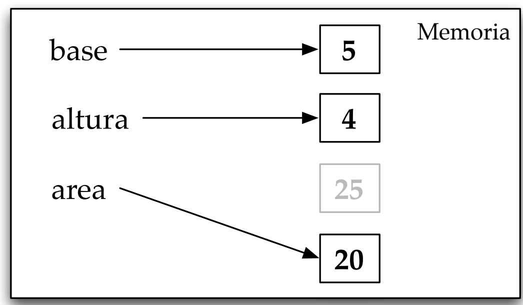
```
>>> altura = 4
>>> area = base * altura
>>> area
20
```

En el primer caso, a la variable `altura` se le asigna el valor `4`. Como es un nuevo valor, se crea el espacio de memoria para contenerlo y la variable `altura` será el nombre con el que podamos acceder a ese dato. A continuación, se recalcula el área del nuevo rectángulo. que en este caso produce el valor `20`. Como se asigna un nuevo valor, éste se guarda en una nueva posición de la memoria, a la que se podría acceder usando la variable `area`. En esta ocasión el valor `25` ha dejado de ser usado, por lo que aparece en gris en la imagen. Los valores que en ese momento tendría nuestro programa guardados en la memoria serían `5`, `4` y `20`.

2.2.7. Otras consideraciones sobre las variables

Hay dos aspectos importantes a la hora de trabajar con las variables que tendrán nuestro programas que a los programadores principiantes les cuesta decidir. La primera es elegir un nombre adecuado para cada variable del programa y la segunda seleccionar qué variables necesitamos usar.

El nombre de una variable puede estar formado por letras (a, b, c,. . .), dígitos (0,1,2,. . .) y el carácter guión bajo o subrayado (_). Además deben tenerse en cuenta las siguientes restricciones:



- el nombre no puede empezar por un dígito
- el nombre no puede ser igual que una palabra reservada del lenguaje
- las letras mayúsculas y minúsculas son diferentes

Es decir, no podemos usar como nombre de una variable `9numeros` ya que empieza por un dígito. Los nombres `Numero` y `numero` son diferentes, ya que no es lo mismo la letra 'N' que la 'n'. Y nunca se pueden usar las palabras de las que se compone el lenguaje python y que detallan en el recuadro anexo. Todas estas reglas son bastante similares en cualquier lenguaje de programación actual.

Curiosidad: Palabras reservadas

Las palabras reservadas de python son 31. No es necesario memorizarlas, se van aprendiendo a medida que se usan al programar y algunas de ellas no se usarán nunca en los programas de la asignatura.

and	as	assert	break	class
continue	def	del	elif	else
except	exec	finally	for	from
global	if	import	in	is
lambda	not	or	pass	print
raise	return	try	while	with
yield				

Pero sin duda el aspecto más importante a la hora de elegir el nombre de una variable es que el nombre sea descriptivo del dato que la variable representa dentro del programa. De esa forma, al leer el programa, tanto por nosotros mismos como autores, como por otros programadores que pudieran leerlo, se entenderá mejor la utilidad que tiene la variable dentro del programa. Siempre hay que utilizar nombres lo más descriptivos posibles, aunque sean largos y formados por varias palabras. El convenio que seguiremos en la asignatura es utilizar palabras en minúsculas separadas por guiones bajos. Por ejemplo: `nombre`, `velocidad_final`, `interes`, `codigo_postal`

El otro aspecto a tener en cuenta sobre las variables, es ¿cuándo hay que usar una variable en un programa? La idea fundamental que se debe tener presente es la siguiente:

Importante

Las variables son las herramientas que usan los programas para almacenar información en la memoria del ordenador.

Desde esa perspectiva, siempre que en un programa se desee guardar alguna información para luego acceder a ella, entonces es necesario crear una variable. Eso hace que no solamente creemos variables para los datos de entrada y salida del programa, sino también para todos aquellos valores intermedios que el programa calcule y de los que queramos mantener el resultado en memoria para acceder a ellos posteriormente. Las variables permiten a los programadores reservar espacio en la memoria para manipular datos. La ventaja de guardar datos calculados previamente es aumentar la velocidad de los programas al no tener que recalcularlos. Otras veces, se guardan datos en memoria desde otros dispositivos como los discos. Por ejemplo, imagina un programa de tratamiento de imágenes que cargue en memoria una imagen desde un archivo. Si la imagen se mantiene en memoria, el programa podrá tratarla de forma rápida ya que no necesitará leer su información de disco. Los datos que están en memoria siempre se pueden tratar de una forma más rápida que si necesitamos acceder a la misma información en un dispositivo secundario.

2.2.8. Expresiones

El objetivo de esta sección es explicar cómo se ejecutan las operaciones que escribimos en los programas y en las que aparecen valores, variables y operadores, especialmente cuando aparecen varios operadores juntos. Es lo que se denomina una expresión.

Definición

Una expresión es una combinación de operadores y operandos (valores, variables) que produce un resultado (valor) de un cierto tipo.

Varias cosas en esta definición. Primero, las expresiones se forman al mezclar los operadores con sus operandos. Los operadores son los que definen las operaciones que se van a hacer. En segundo lugar, las expresiones producen siempre un valor. Ese valor será, como es lógico, de algún tipo. Tanto el valor producido, como su tipo, dependerán de los operandos y de los operadores que se usen.

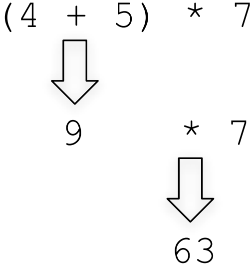
La mayor dificultad que entrañan las expresiones con varios operadores es saber el orden en que éstos se ejecutarán. Eso lo determina lo que se denomina la precedencia de los operadores. La precedencia es la propiedad que sirve para decidir qué operador debe aplicarse primero cuando en una expresión aparecen varios operadores de distintos grupos. Vamos a verlo con dos ejemplos, las expresiones `4+5*7` y `(4+5)*7`. En ambas aparecen dos operadores, la suma (`+`) y el producto (`*`), con los mismos valores, sin embargo el resultado será diferente por la presencia de los paréntesis.

En el caso de la expresión `4+5*7`, la primera operación que se hace es el producto ya que el operador `*` tiene más precedencia que el operador `+`. La expresión `5*7` produce el valor `35`, al que posteriormente se le suma el valor `4`. El resultado final de la expresión es `39`. Gráficamente:

```
4 + 5 * 7
        ⇩
4 +   35
  ⇩
 39
```

En cambio en la expresión `(4+5)*7`, tenemos los mismos valores y operadores, pero además se han incluido unos paréntesis que delimitan la operación de suma. Los paréntesis sirven para agrupar las operaciones que se quieren hacer primero. En este caso, lo que se indica es que se debe

hacer antes la suma que el producto. Primero se realiza `4+5`, produciendo el valor entero `9`, y a continuación se hace el producto de dicho valor por `7`, con lo que el resultado final de la expresión es `63`:



Puedes consultar la tabla de precedencia de operadores en python en múltiples páginas web, por ejemplo en `http://docs.python.org/reference/expressions.html`. Sin embargo memorizarse la tabla es difícil, y en realidad tampoco es necesario. Como se ha visto en los dos ejemplos anteriores, usando los paréntesis adecuadamente se pueden cambiar el orden en que se ejecutan los operadores de una expresión. Además, el uso de los paréntesis hace que algunas expresiones se entiendan mejor al leerlas.

En todo caso, las tablas de precedencia de todos los lenguajes suelen seguir una serie de reglas que si se conocen pueden permitirnos escribir expresiones complejas sin abusar tanto del uso de los paréntesis. Citaremos las cuatro reglas más básicas:

1. lo que está entre paréntesis se hace primero
2. los operadores aritméticos tienen más precedencia que los relacionales, y los relacionales más que los lógicos
3. los operadores unarios de un grupo más que los binarios de ese mismo grupo
4. `**` más que los multiplicativos (`*`, `/`, `%`), y éstos más que los aditivos (`+`, `-`). Esta regla también se da con los lógicos (`and` más precedencia que `or`)

La primera regla es la más importante de todas, ya la hemos comentado suficientemente. En el caso de la segunda, se comprende si se piensa en una expresión como `x-1<=10`. Lo lógico al leer esa expresión en matemáticas es que queremos comprobar si al restarle `1` a `x` el valor resultante es menor o igual que `10`. La expresión puede escribirse tal cual, ya que el operador aritmético (`-`) tiene más precedencia que el relacional (`<=`). La tercera regla se puede entender al analizar una expresión como `a*-b`. En ese caso lo que se quiere es que se multiplique la variable `a` por el valor resultante de cambiar el signo de la variable `b`, esto es, `-b`. Como el operador `-` es en este caso unario, se hace antes que el operador `*`. La única excepción a la regla tercera se da en el caso de el operador binario `**` que tiene más precedencia que los operadores unarios aritméticos, como el `-`, lo que explica que `-1**2` de como resultado -1, mientras que `(-1)**2` da como resultado 1. Para finalizar, la última regla se ha visto en los dos ejemplos utilizados para explicar el concepto de precedencia y el uso de paréntesis, las multiplicaciones se hacen antes que las sumas.

2.3. Uso de entrada/salida por consola

Una característica muy importante de los programas es que sean capaces de *interaccionar con el usuario*, es decir, que sean capaces de pedir información al usario por teclado (entrada estándar) y de mostrar los resultados por pantalla (salida estándar).

El siguiente programa en `Pyhton` calcula el área de un triángulo, pero la base y la altura siempre son las mismas y no se informa del área calculada.

```
1 base=5.0
2 altura=3.0
3 area=(base*altura)/2
```

Evidentemente, sería mucho más útil si se pudiera aplicar a cualquier triángulo y, por supuesto, si se informa al usuario del resultado final. Para ello es necesario conocer las operaciones de entrada/salida por consola que nos ofrece `Python`. Mediante su aplicación, el programa anterior quedaría de la forma siguiente:

```
1 base = float (raw_input("Dame la base:"))
2 altura = float (raw_input("Dame la altura:"))
3 area =(base*altura)/2
4
5 print area
```

2.3.1. Entrada por teclado

La forma más sencilla de obtener información por parte del usuario es mediante la función `raw_input`. Esta función devuelve una cadena con los caracteres introducidos por el usuario mediante el teclado.

Atención

Cuando se llama a esta función, el programa se **para y espera** hasta que el usuario teclea algo. Cuando el usuario presiona `Return` o `Enter`, el programa continúa y `raw_imput` retorna lo que ha tecleado el usuario como cadena de caraceteres (*string*).

```
>>> entrada = raw_input()
_
```

```
>>> entrada = raw_input()
Hola
>>> print entrada
Hola
```

Antes de pedir un dato por teclado al usuario, es una buena idea sacar un mensaje informando de lo que se espera. Por este motivo, *raw_input* puede llevar como parámetro una cedena de caracteres que se saca por la pantalla justo antes de pedir la entrada al usuario (*indicador* o *prompt* en inglés).

```
>>> entrada = raw_input("Introduce tu nombre: ")
Introduce tu nombre:
```

```
>>> entrada = raw_input("Introduce tu nombre: ")
Introduce tu nombre: Jorge Javier
>>> print entrada
Jorge Javier
```

Si necesitamos un número como entrada, un entero o un flotante, en lugar de una cadena, podemos utilizar las funciones *int* y *float* para convertir la cadena al tipo numérico correspondiente.

```
>>> base = float (raw_input("Introduce la base de un triángulo:\n"))
Introduce la base de un triángulo:
7
```

Aunque hay que tener en cuenta que si lo que introduce el usuario no es un número válido se genera un error al no poder realizar la conversión.

```
>>> base = float (raw_input("Introduce la base de un triángulo:\n"))
Introduce la base de un triángulo:
A
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for float(): A
```

Como se comentará mas adelante, el carácter especial `\n` al final del mensaje del *raw_imput* (*prompt*) causa un salto de línea.

2.3.2. Salida por pantalla

La forma más sencilla de mostrar algo en la salida estándar es mediante el uso de la sentencia `print`, como hemos visto en varios ejemplos anteriores. En su forma más básica a la palabra clave `print` le sigue una cadena de caracteres, que se mostrará en la pantalla del ordenador al ajecutarse la sentencia. Si se omite la cadena de caracteres se produce un salto de línea.

```
>>> print

>>> print "Hola mundo"
Hola mundo
```

Atención

Las comillas que se usan para identificar una cadena de caracteres no aparecen en la salida por pantalla.

Por defecto, la sentencia `print` muestra algo por la pantalla y se posiciona en la línea siguiente. Si queremos mostrar más de un resultado en la misma línea, basta con separar con *comas* todos los valores que deseamos mostrar. Esto es debido a que `Python` interpreta la *coma* como un espacio de separación.

```
>>> print "Hola mundo", "de la programación"
Hola mundo de la programación
```

También se puede usar `print` para imprimir por pantalla valores que no sean cadenas de caracteres, como números enteros o flotantes.

```
>>>print "Te costará entre", 30, "y", 49.5, "euros"
Te costará entre 30 y 49.5 euros
```

Las cadenas de caracteres que muestra la sentencia `print` pueden contener *caracteres especiales*. Se trata de caracteres que van precedidos por la barra invertida `\` y que tienen un significado específico. Los más utilizados son `\n`, el carácter de nueva línea, y `\t`, el de tabulación. Por ejemplo, la siguiente sentencia imprime la palabra "Hola" seguida de un renglón vacío y en la línea siguiente (debido a los dos caracteres de nueva línea, `\n`) la palabra "mundo" indentada (debido al carácter tabulador, `\t`).

```
>>>print "Hola\n\n\tmundo"
Hola

    mundo
```

2.4. Manejo de estructuras básicas de control de flujo

Control de flujo, en programación, se refiere a las diferentes secuencias de sentencias que opcionalmente se pueden tomar en la ejecución de un programa. Las estructuras de control de flujo son las herramientas que permiten definir estas opciones. Existen 3 estructuras de control fundamentales:

- Secuencial (BLOQUE)
- Aternativa simple (SI-ENTONCES) o doble (SI-ENTONCES-SI_NO)
- Repetitiva (MIENTRAS)

Estas estructuras se representan gráficamente en la figura 2.1. Obsérvese que todas ellas tienen un único punto de entrada y un único punto de salida en el rectángulo en línea de trazos, lo que permitirá más adelante componer unas con otras, enlazándolas por dichos puntos de entrada y salida.

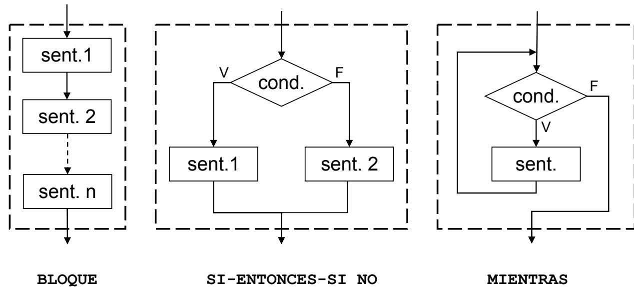


Figura 2.1: Estructuras de control fundamentales. *cond.* se refiere a condición, *sent.* a sentencia, *V* a True, *F* a False.

Böhm y Jacopini demostraron en los años 60 que todo programa puede realizarse a base de las 3 estructuras anteriores: La secuencial, la alternativa, y una cualquiera de las 2 repetitivas, así como de "anidamientos" de unas estructuras en otras.

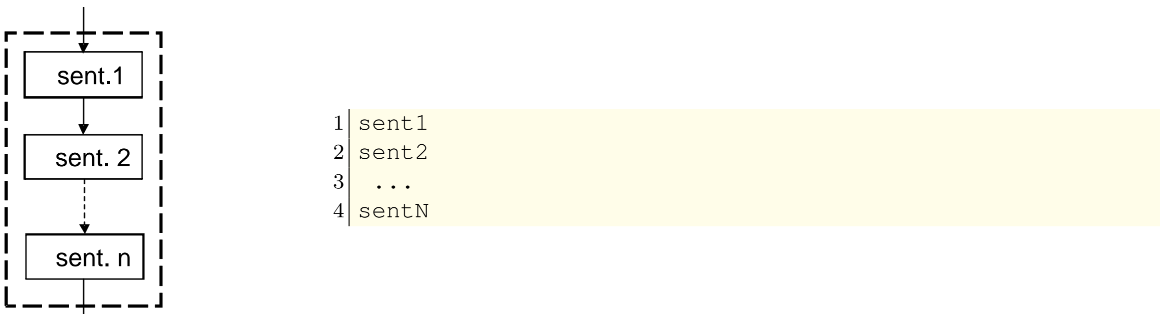
2.4.1. Estructura secuencial (BLOQUE)

Consiste en ejecutar una sentencia a continuación de otra.

Se pueden agrupar varias sentencias para formar una única sentencia, denominada "sentencia compuesta". En otros lenguajes se usan delimitadores de bloque (p.ej: llaves).

Implementación En Python, lo que "delimita" el bloque es que todas tengan el mismo nivel de indentación, es decir, el mismo número de espacios por la izquierda. Aunque el número de espacios puede ser cualquiera que se desee, ha de ser el mismo para todas las sentencias que componen el bloque secuencial. Es costumbre que este número de espacios sea múltiplo de 4.

Por ejemplo, en la figura 2.2 se muestra la implementación en lenguaje Python del bloque correspondiente, asumiendo que `sent1`, `sent2`, etc. son diferentes sentencias Python, (asignaciones, expresiones, etc).




```
1 sent1
2 sent2
3  ...
4 sentN
```

Figura 2.2: Bloque secuencial

Hay que tener en cuenta que el "bloque principal" en python ha de tener cero espacios de indentación, y que sólo los bloques que aparezcan dentro de otras estructuras de control irán indentados. En los ejemplos que hemos visto hasta este momento, todos los programas tienen un solo bloque, que es el "principal" y por tanto todos llevan cero espacios de indentación. En las secciones siguientes en las que usaremos bloques dentro de otras estructuras de control podremos ver ejemplos en los que la indentación juega su papel fundamental.

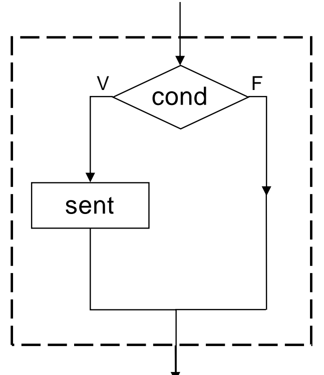
Se puede usar "punto y coma" para separar sentencias si están en la misma línea, pero en esta asignatura no usaremos esa caracerística y pondremos siempre cada sentencia en una línea separada. Haciéndolo así, no es necesario poner un punto y coma para separarlas, y en particular no es necesario poner punto y coma al final de cada línea (decimos esto porque en otros lenguajes como C o Java sí es obligatorio terminar cada línea con punto y coma).

2.4.2. Estructura alternativa

Permite elegir entre dos alternativas, según sea verdadera o falsa, la condición que se evalúa. Existen dos subtipos

- Aternativa simple (`if`)
- Aternativa doble (`if-else`)

En la **estructura alternativa simple** se proporciona una condición y una sentencia (o un bloque de ellas). Si la "condición" es verdadera se ejecuta la "sentencia". Si no, no se ejecuta ninguna acción.



Implementación en python:

```
1 if cond:
2     sent
```

Figura 2.3: Estructura alternativa simple (`if`)

Esta estructura de control se puede representar gráficamente como se muestra en la figura 2.3. En esta misma figura se muestra cómo traducir el gráfico a su implementación en el lenguaje Python.

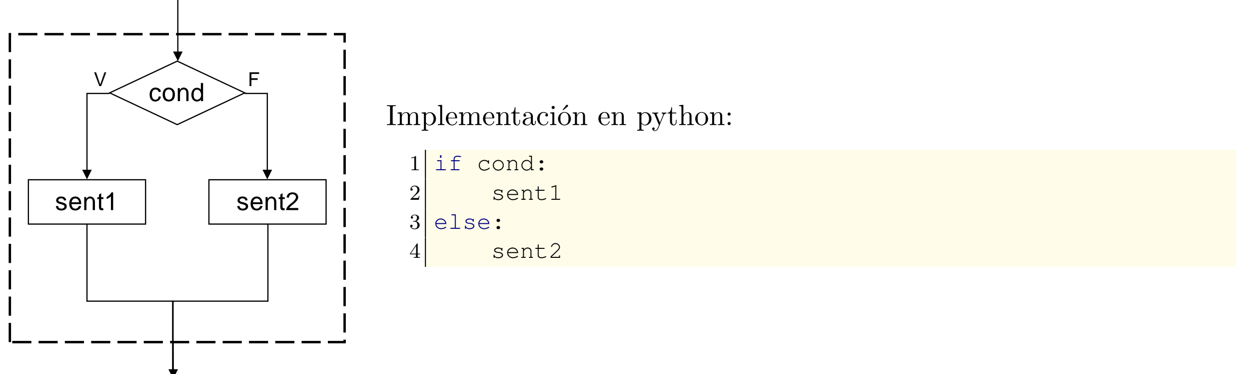
Sobre la implementación, observar los siguientes detalles:

- La condición (que se representa por `cond` en la figura) será una expresión booleana cuyo resultado será `True` o `False`. La condición no necesita ir encerrada entre paréntesis (a diferencia de otros lenguajes como C o Java).
- Tras la condición, se ponen dos puntos (`:`), finalizando así la línea.
- La sentencia a ejecutar (representada por `sent` en la figura) debe ir indentada, habitualmente cuatro espacios como ya hemos dicho antes. Si se trata de un bloque en lugar de una sola sentencia, todo el bloque irá indentado la misma cantidad de espacios.

Ejemplo Suponiendo que la variable `nota` contiene la calificación de un alumno, el siguiente fragmento de programa determina si ha superado la prueba.

```
1 # El programa obtiene la nota por algún medio
2 if nota >= 5.0:
3     print "Prueba superada"
```

En la **estructura alternativa doble** (`if-else`) se proporcionan dos posibles sentencias (o bloques de sentencias), a elegir según el resultado de la condición, como se muestra en la figura 2.4. Si la "condición" es verdadera se ejecuta la "sentencia1". Si no, se ejecuta la "sentencia2".



Implementación en python:

```
1 if cond:
2     sent1
3 else:
4     sent2
```

Figura 2.4: Estructura alternativa doble (`if-else`)

Sobre la implementación, observar los siguientes detalles:

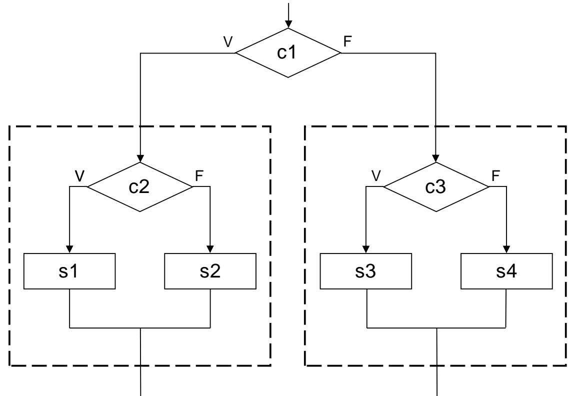
- La condición sigue las mismas observaciones que para el caso anterior.
- La sentencia a ejecutar para el caso `True` (representada por `sent1` en la figura) debe ir indentada, habitualmente cuatro espacios como ya hemos dicho antes. Si se trata de un bloque en lugar de una sola sentencia, todo el bloque irá indentado la misma cantidad de espacios.

  Lo mismo cabe decir sobre la sentencia (o bloque) para el caso `False`, representada por `sent2` en la figura.
- La palabra `else` ha de finalizarse con dos puntos y ha de tener el mismo nivel de indentación que la palabra `if`, de este modo se "ve" a qué `if` corresponde el `else`.

Ejemplo El siguiente fragmento de programa determina si la variable `num` de tipo entero contiene un número par o impar.

```
1 # El programa inicializa num por algún medio
2 if num%2 == 0:
3     print "Es par"
4 else:
5     print "Es impar"
```

Anidamiento de estructuras alternativas Las estructuras alternativas pueden "anidarse", esto es, donde debería ir `sent1` o `sent2` en el diagrama anterior, podemos poner otra estructura alternativa. La figura 2.5 muestra un ejemplo de esto. El anidamiento puede complicarse más, si en lugar de `s1`, `s2`, `s3` o `s4` en dicha figura, incluímos otra estructura alternativa. El lenguaje no pone límite al nivel de anidamiento, que puede ser tan profundo como se quiera. En la práctica, sin embargo, utilizar más de dos niveles resulta difícil de leer, y suele ser un síntoma de que se podía haber diseñado el algoritmo de otra forma, o de que parte de él puede ser extraído a una función (concepto que veremos más adelante).



Implementación en python:

```
1 if c1:
2     if c2:
3         s1
4     else:
5         s2
6 else:
7     if c3:
8         s3
9     else:
10        s4
```

Figura 2.5: Estructuras alternativas anidadas

Como se ve en la figura, la implementación en python hace visible el anidamiento de estas estructuras mediante el nivel de indentación de las líneas. Las líneas que usan el mismo nivel de indentación, están al mismo nivel de anidamiento. Es importante que cada `else` vaya anidado con su `if`.

Finalmente, python proporciona una estructura **multialternativa** (`if-elif-else`), que permite elegir entre varias alternativas según el resultado de diferentes expresiones booleanas. En realidad, puede implementarse mediante una serie de `if-else` anidados "en cascada", pero la sintaxis con `if-elif-else` resulta más legible, al reducir el anidamiento.

La idea general se leería como "si se cumple `cond1` haz `sent1`, si no, si se cumple `cond2` haz `sent2`, si no, si se cumple `cond3` haz `sent3`, etc.. y si no se cumple ninguna, haz `sentencia`", y la implementación en Python sería la siguiente:

```
1 if cond1:
2     sent1
3 elif cond2:
4     sent2
5 elif cond3:
6     sent3
7 ...
8 elif condN:
9     sentN
10 else:
11     sentencia    #cualquier otro caso no contemplado
```

Observaciones sobre la implementación:

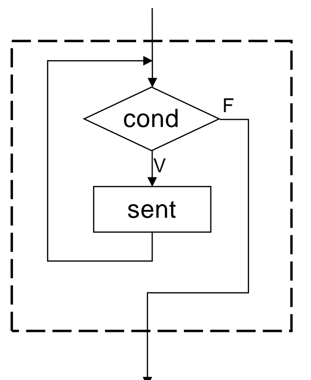
- La palabra `elif` es una contracción de `else if`.
- Cada uno de los `elif` ha de ir forzosamente alineado con el `if` inicial, así como el `else` final.
- Se pueden poner tantos `elif` como se necesiten.
- El `else` final es opcional. Si no se pone, y ninguna de las condiciones se ha cumplido, entonces no se ejecutará ninguna sentencia de las incluidas en el `if-elif`.
- Como siempre, cualquiera de las `sent` del código anterior puede ser un bloque de sentencias. Basta escribir cada una en una línea y todas con el mismo nivel de indentación.

Ejemplo El siguiente ejemplo pide al usuario una nota numérica y escribe la correspondiente nota cualitativa. Otro uso típico de la estructura multialternativa es para crear "menús" en los que el usuario puede elegir una opción entre varias que se le presentan.

```
1 nota = int(raw_input("Introduzca la nota del examen: "))
2 if (nota >= 0) and (nota < 5):
3     print "Suspenso"
4 elif (nota >= 5) and (nota < 7):
5     print "Aprobado"
6 elif (nota >= 7) and (nota < 9):
7     print "Notable"
8 elif (nota >= 9) and (nota <= 10):
9     print "Sobresaliente"
10 else:
11     print "Nota no válida"
```

2.4.3. Estructura repetitiva `while`

Una estructura repetitiva es un bloque de sentencias (denominado *cuerpo* del bucle) que se va a ejecutar varias veces. El bucle incluye una condición, que regula si se seguirá repitiendo o no. Las instrucciones dentro del bloque modificarán los valores de algunas de las variables que forman parte de la condición, haciendo que en algún momento la evaluación de esa condición cambie y por tanto el bucle ya no se repita más.



Implementación en python:

```
1 while cond:
2     sent
```

Figura 2.6: Estructura repetitiva MIENTRAS (`while`)

El bucle `while` se puede representar gráficamente como se muestra en la figura 2.6. La condición es lo primero que se evalúa y si el resultado es `False`, se abandona el bucle sin ejecutar su sentencia `sent`. Si la condición es `True`, entonces se ejecuta `sent` y tras ello se vuelve otra vez a la condición, para evaluarla de nuevo. Mientras la condición siga siendo cierta, la sentencia `sent` se ejecutará una y otra vez. Se entiende que `sent` modifica alguna variable, de modo que `cond` pueda pasar a ser falsa, y de este modo se terminaría el bucle.

Observar que si la condición es inicialmente falsa el cuerpo del bucle no se ejecutará ni siquiera una vez.

Si hubiera más de una "sentencia" dentro del bucle, se deberá formar una única sentencia compuesta usando el mismo nivel de indentación, como se ve en el siguiente ejemplo.

Ejemplo El siguiente bucle calcula el número de dígitos de un entero n dado (introducido por teclado).

```
1 n = int(raw_input("Numero entero: "))
2 a = n
3 digitos = 1
```

```
4 while a / 10 != 0:
5     digitos = digitos + 1
6     a = a / 10
7 print n, "tiene", digitos, "digitos"
```

Cómo construir un bucle `while`

En este apartado se verá cómo abordar la construcción de programas que requieren una estructura repetitiva para obtener la solución del problema a resolver. Para ello es necesario dar una definición de programa (o algoritmo) que complemente la dada inicialmente en el apartado 2.1.3 y que esté basada en el concepto de **estado** de un programa. Este estado es el conjunto de valores de las variables del programa en un instante dado de su ejecución.

Definición

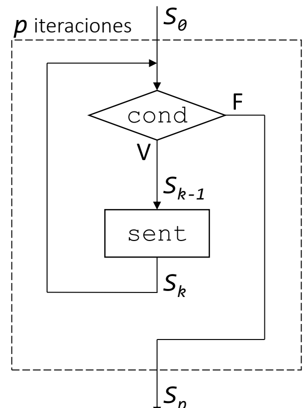
Un programa (o algoritmo) es una secuencia de instrucciones tal que, seguidas paso a paso, proporciona una secuencia ordenada de estados finita en la que el último estado (o *estado final*) contiene la solución del problema a resolver para cada entrada válida.

Para el ejemplo de bucle `while` del apartado previo, los estados que se obtienen para la entrada $n = 62574$ son los que se muestran en la tabla adjunta. Para una mejor comprensión de la ejecución paso a paso del programa, la tabla se ha complementado con los valores de las expresiones del bucle, en concreto con la condición de continuación de éste.

Estado	n	*digitos*	a	$a/10 \neq 0$
S_0	62574	1	62574	V
S_1	62574	2	6257	V
S_2	62574	3	625	V
S_3	62574	4	62	V
S_4	62574	5	6	F

Se observa que para la entrada $n = 62574$, la ejecución del programa genera una secuencia de cuatro estados: S_0, S_1, S_2, S_3 y S_4, donde S_4 es el último estado (o *estado final*) porque para éste se cumple la condición $a/10 = 0$ y el bucle `while` ya no se ejecutará más, además, este último estado contiene la solución al problema (62574 tiene 5 dígitos).

La reiterada ejecución de la sentencia interna (`sent`) del bucle es la que produce el cambio de estado de las variables, de forma que en la primera iteración S_0 es el estado antes de ejecutarse la sentencia interna y S_1 es el estado después de su ejecución. A su vez, este último también es el estado de entrada para la siguiente iteración (la segunda) y así sucesivamente.



Implementación en python:

```
# Estado inicial, S0
while cond: # iteración k  (k = 1, 2, ..., p)
    # Estado Sk-1
    sent
    # Estado Sk
# Estado final, Sp
```

Figura 2.7: Secuencia de estados de un bucle (`while`)

En general, si el número de iteraciones del bucle `while` es p ($p \geq 0$), la secuencia de estados que resulta de su ejecución (S_0, S_1, ..., S_p) es tal que cumple lo siguiente (ver figura 2.7):

- El *estado inicial*, S_0, es el estado de las variables antes de entrar al bucle `while`.
  Para el ejemplo este estado es $S_0 = \{n = 62574, digitos = 1, a = 62574\}$.
- El *estado final*, S_p, es el estado de las variables al finalizar el bucle `while`. Este estado, que se alcanza cuando la condición `cond` del bucle es falsa, tiene que contener el resultado que se espera obtener tras la ejecución del bucle. Si este estado es a su vez la salida del programa, entonces tiene que dar la solución del problema a resolver.
  Para el ejemplo este estado es $S_4 = \{n = 62574, digitos = 5, a = 6\}$.
- El *cambio de estado*, o forma de pasar de un estado al siguiente en cada iteración $k = 1, 2, ..., p$, está establecido por la sentencia interna `sent` del bucle. Para cada iteración k, S_{k-1} es el estado a la entrada de la sentencia interna y S_k es el estado a la salida de ésta (este estado será la entrada para la siguiente iteración).

Hasta aquí se ha visto como es la secuencia de estados que resulta de la ejecución del bucle y se ha caracterizado ésta mediante el *estado inicial*, el *estado final* y el *cambio de de estado*. Ahora se verá el proceso inverso; es decir, cómo construir el bucle `while` partiendo de la secuencia de estados que resuelve el problema para alguna entrada válida. Para ello se deben aplicar las siguientes reglas:

Reglas de construcción del `while`

1. Inicializar todas las variables para obtener el *estado inicial*.
2. Determinar la condición de parada del bucle `while`. Para establecer esta condición se tendrá en cuenta que el *estado final* tiene que contener la salida que se espera del bucle (si ésta es también la salida del programa, la solución del problema). La negación de la condición de parada del bucle es su condición, `cond`, de continuación.
3. Obtener la sentencia interna, `sent`, del bucle para producir el *cambio de estado*. Para ello hay que determinar cómo cambia cada una de las variables del estado.

Observar que las reglas dadas se aplican a alguna secuencia de estados que soluciona el problema para alguna entrada y, por tanto, es necesario determinar esta secuencia a priori (el programa, o bucle, todavía no se ha realizado). La determinación de esta secuencia de estados pasa, ineludiblemente, por **entender** y **analizar** el problema que se quiere resolver. Un análisis detallado del problema facilitará la identificación de la información que requiere el estado del programa (sus variables), o bien permitirá reconocer un *problema tipo* que sirva de guía en la obtención de la solución.

Una vez identificado el estado del programa (conjunto de variables requeridas), es sencillo obtener una tabla de estados (o parte de ésta) para alguna entrada concreta o genérica. Si fuera el caso, esta tabla siempre se deberá completar con los valores de expresiones o condiciones que influyan en el estado de alguna de las variables.

A continuación se aplicará el proceso de construcción del bucle `while` a un ejemplo concreto. Pero antes una última consideración que evitará que se cometan errores en la codificación de los programas:

Importante

Como norma general, un programa nunca debería modificar las variables que contienen la información (o datos) de entrada al mismo. Si para avanzar hacia la solución del problema hubiera que realizar alguna modificación a los valores de éstas, previamente se asignarán estos valores a otras variables y serán éstas últimas las que se modifiquen.

Observar que para el ejemplo que se ha estado tratando desde el principio, efectivamente así se ha hecho. La entrada al programa, el valor de la variable `n`, no se modifica, en su lugar, como es necesario utilizar y modificar el valor proporcionado se utiliza una variable `a`.

Ejemplo Escribir un programa que permita obtener en pantalla la secuencia de enteros 1, 3, 6, 10, 15, ... que no superen un valor *umbral* (≥0) dado.

Antes comenzar con el proceso de construcción del bucle, observar que la generación de cualquier secuencia para la que se pueda establecer las reglas de construcción de su término general es intrínsecamente iterativa. Además, tal y como se enuncia el problema, se desconoce de antemano cuantos términos se van a generar; es decir, cuántas iteraciones se van a realizar. Por tanto, este es un problema adecuado para solucionar mediante un bucle `while`.

Para poder resolver el problema es necesario saber cómo se genera la secuencia de enteros solicitada: 1, 3, 6, 10, 15, ... Seguramente el lector ya se habrá dado cuenta de que el siguiente elemento de la secuencia es el 21 y que la forma de obtener esta secuencia de enteros es la siguiente: 1, $1+2=3$, $3+3=6$, $6+4=10$, $10+5=15$, ...

Simplemente con determinar cómo se genera la secuencia de enteros, se identifica el estado del programa: se necesitan tres datos enteros (tres variables).

Una variable, `umbral`, para el valor umbral de entrada y para poder cambiar el estado (generar cada término de la secuencia de salida): el último término obtenido (variable `termino`) y una cantidad a sumar que se va incrementando (variable `cantidad`). Ahora se puede realizar la tabla de estados, por ejemplo para la entrada $umbral = 25$, y aplicar los tres reglas que permiten construir el bucle `while`.

Estado	*umbral*	*termino*	*cantidad*
S_0	25	1	2
S_1	25	$1+2=3$	3
S_2	25	$3+3=6$	4
S_3	25	$6+4=10$	5
S_4	25	$10+5=15$	6
S_5	25	$15+6=21$	7
S_6	25	$21+7=28$	8

1. Establecer el estado inicial:

```
umbral = int(raw_input("Umbral: "))
termino = 1
cantidad = 2
```

2. Determinar la condición de parada del bucle. Éste tiene que finalizar cuando se alcance un término de la secuencia de salida (valor de `termino`) que supere el valor de `umbral`.
   - Condición de terminación del `while`: `termino > umbral`
   - Condición de continuación del `while`: `termino` ≤ `umbral`
3. Cambio de estado:

```
print termino,  #escribe en pantalla el término actual
termino = termino + cantidad
cantidad = cantidad + 1
```

Observar que en la aplicación de la tercera regla, lo más frecuente es que el orden en que se cambia el estado de las variables (y por tanto el orden de las sentencias en el bloque `sent`) sea importante. En particular, para este ejemplo, la variable `termino` ya tiene el valor del primer término de la secuencia de salida antes de entrar al bucle y, por tanto, se debe escribir en pantalla su valor antes de cambiar su estado. Por otra parte, en cada paso k la variable `termino` depende del valor de la variable `cantidad` en el paso $k-1$ (la iteración previa), por tanto antes de cambiar el estado de la variable `cantidad` se debe cambiar el estado de la variable `termino`.

Una vez aplicadas las reglas la construcción del bucle `while`, la escritura del programa es inmediata:

```
1 umbral = int(raw_input("Umbral: "))
2 termino = 1
3 cantidad = 2
4 while termino <= umbral:
5     print termino,  # escribe en pantalla el término actual
6     termino = termino + cantidad
7     cantidad = cantidad + 1
```

```
8 print
```

2.4.4. Estructura repetitiva `for`

Aunque con el bucle `while` se puede implementar ya cualquier algoritmo, es muy frecuente que aparezca la necesidad de realizar un bucle para el que se conoce de antemano el número de iteraciones a realizar. En este caso, el bucle normalmente está controlado por un contador que parte de un valor inicial que se va incrementando (o decrementando) hasta superar un valor final.

Estado	i	var	...
S_0	0	var_0	...
S_1	1	var_1	...
S_2	2	var_2	...
...	...	...	...
S_n	n	var_n	...

Supongamos por ejemplo que el bucle se realiza n veces, se podría resolver el problema mediante un bucle `while` como se vio en el apartado previo. El estado estaría formado por una serie de variables, una de ellas el contador i del número de iteraciones, tal y como se muestra en la tabla de estados adjunta. Y aplicando las reglas de construcción del `while` se obtendría:

Estado inicial: `i = 0`, `var = ...`
Condición de parada: `i > n`

```
1 i = 0
2 var = ...
3 while i <= n:
4     i = i + 1
5     #cambio de estado de var
6     #cambio de estado de otras variables
```

Observar que el valor inicial de la `i` no tiene por qué ser necesariamente 0. Asimismo, el valor que sumamos a `i` en cada iteración del bucle no tiene por qué ser `1`. Podríamos querer contar de 2 en 2, o de 3 en 3, etc. A veces es incluso necesario contar “hacia atrás”, de modo que la `i` se va decrementando en cada iteración del bucle.

Todas estas variantes se pueden implementar con facilidad modificando ligeramente la estructura `while` que acabamos de presentar. Sin embargo, ya que este tipo de bucles aparece muy a menudo en todos los programas, la mayoría de los lenguajes de programación incluyen una sintaxis específica para implementarlos, que permite simplificar ligeramente el código y reducir el número de líneas necesarias. La mayoría de los lenguajes (por ejemplo, C, Java) utilizan la palabra reservada `for` para este tipo de bucles.

En Python también existe el bucle `for` y puede ser usado para implementar esta idea de “bucle controlado por contador”. El `for` de Python es mucho más versátil, y puede usarse no solo para hacer que la `i` vaya tomando el valor de una serie de enteros sucesivos, sino también para obtener los caracteres sucesivos de una cadena o para que vaya tomando una secuencia de valores arbitraria, a través de las listas (esto se verá más adelante). Pero no nos adelantemos. En esta sección veremos únicamente cómo `for` puede usarse para implementar el bucle controlado por contador.

Para este cometido, Python proporciona la función `range()` que genera una secuencia de enteros entre un valor inicial y final que se le suministre y separados entre sí una cantidad que también se le puede suministrar. Es sobre esta secuencia de enteros sobre la que la variable `i` irá tomando valores.

La sintaxis general de `range` es `range(inicial, final, paso)`, siendo:

- *inicial* el valor del primer entero de la secuencia. Puede omitirse, y entonces recibirá automáticamente el valor cero.
- *final* el valor del primer entero “fuera” de la secuencia. Este valor ya no formará parte de la secuencia de enteros generada, que se detendrá en el entero anterior a éste.
- *paso* es la distancia entre los enteros que se van generando. Puede omitirse y entonces recibirá automáticamente el valor uno.

Importante

Aunque pueden omitirse algunos de sus parámetros, en esta asignatura utilizaremos el `range` siempre con sus tres parámetros

Veamos un ejemplo de rango ascendente, que recorre los impares menores de 10:

```
>>> range(1,10,2)
[1, 3, 5, 7, 9]
```

En el bucle ascendente, el último número que se incluye en el rango es el mayor que cumpla ser menor que el valor *final*.

¿Qué parámetros tendríamos que pasarle a `range` para generar un rango que incluya los números pares comprendidos entre 1 y 10, incluyendo al 10? Se deja como ejercicio para el lector.

Finalmente, si el paso es negativo, el valor *inicial* tendrá que ser mayor que el valor *final*, de lo contrario se nos generaría un rango "vacío". Un par de ejemplos:

```
>>> range(1,10,-1)
[]
>>> range(10,1,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2]
```

Observa como en el primer caso el rango que se obtiene está vacío, debido a que hemos puesto un valor *inicial* que es menor que el valor *final*. En el segundo caso el rango se genera correctamente, pero observa que, al igual que en los rangos ascendentes, el valor *final* especificado (el 1) no forma parte de la secuencia que se genera. Es decir, en este caso `range` va generando números hasta que encuentra uno que es *menor o igual* que el final especificado y entonces se detiene y este último no forma parte del resultado.

¿Qué parámetros habría que pasarle a `range` para que genere todos los múltiplos de 3 positivos y menores de 100? Se deja como ejercicio para el lector.

Ahora que ya se sabe como utilizar el `range` para generar una lista de números, se dará a continuación la sintaxis del bucle `for` para hacer que una variable `i` vaya tomando valores sobre esa lista. La sintaxis es la siguiente:

```
for i in range(inicial,final,paso):
    sentencia1
    sentencia2
```

La variable `i` del `for`, es la variable de control del bucle. El estado de esta variable está determinado por el `range` y su cambio de estado por el `paso` de éste. Además, siempre es la primera variable que cambia de estado.

Cómo construir un bucle `for`

Un bucle `for` es una estructura repetitiva como lo es el `while`, y al igual que para éste, se obtendrá el bucle `for` partiendo de una secuencia de estados que solucione el problema para alguna entrada.

Por tanto, la forma de proceder para obtener las reglas de construcción del bucle `for` es la misma que la ya vista para el bucle `while` y las reglas son similares a las ya dadas para este último. Las diferencias que hay en las reglas de construcción del `for` se deben a las características propias de este tipo de bucle. Estas diferencias son las siguientes:

- El bucle `for` sólo se puede utilizar si se conoce a priori el número de iteraciones a realizar y, precisamente, es el número de iteraciones el que establece su terminación. Si el paso del `range` es 1 (`range(inicial, final, 1)`), entonces la diferencia entre el valor `final` e `inicial` del `range` es el número de iteraciones.

- Siempre hay una variable, la variable de control del bucle, cuyo estado es el número de iteración o bien un valor directamente relacionado con éste.
- La inicialización de las variables y el cambio de estado de éstas se realiza igual que para el bucle `while`, con una salvedad. La variable de control del bucle se inicializa con el `range`, a su vez, el paso de éste establece su cambio de estado.

Reglas de construcción del `for`

1. Inicializar todas las variables, excepto la variable de control del bucle, para obtener el *estado inicial*.
2. Obtener el `range` del `for`, `range(inicial, final, paso)`, a partir del número de iteraciones n (≥ 0) del bucle. Si $paso = 1$, entonces se tiene que cumplir $final - inicial = n$. Además, si el cambio de estado de alguna otra variable depende de la variable que controla el bucle, entonces `inicial` tiene que ser el valor correspondiente al primer cambio de estado de la variable de control.
3. Obtener la sentencia interna del bucle para producir el *cambio de estado*. Para ello hay que determinar cómo cambia cada una de las variables del estado, excepto la variable de control del bucle.

A continuación se verá un ejemplo de construcción del bucle `for`. El ejemplo va a ser el mismo que el que se vió para el `while`, pero se cambiará el enunciado de forma conveniente para que se pueda solucionar con un `for`.

Ejemplo Dado un entero n (≥ 0), escribir un programa que permita obtener en pantalla los n primeros términos de la secuencia de enteros 1, 3, 6, 10, 15, ...

Como el problema ya se ha tratado en el apartado previo se partirá directamente de la secuencia de estados para la entrada $n = 5$, éstos se muestran en la tabla adjunta.
La tabla es análoga a la ya utilizada en el caso del `while` sólo cambia la variable de entrada (`n` por `umbral`) y se añade la variable adicional `k` que será la variable de control del bucle.

Estado	n	*termino*	*cantidad*	k
S_0	5	1	1	0
S_1	5	$1+2=3$	2	1
S_2	5	$3+3=6$	3	2
S_3	5	$6+4=10$	4	3
S_4	5	$10+5=15$	5	4
S_5	5	$15+6=21$	6	5

El cambio más significativo que se ha hecho en la tabla, ha sido desplazar el valor de cantidad un lugar hacia abajo. Más adelante se comentará porque se ha hecho este cambio y también se verá que éste no es un cambio trascendental.

Se aplicaran a continuación las reglas de construcción del bucle `for` con `k` como variable de control. Antes una observación, el valor de la variable `cantidad` siempre es uno más que el valor de la variable `k`, por tanto, se puede prescindir de la primera para el estado del programa, y cuando se necesite su valor se utilizará el valor de `k+1`.

1. Establecer el estado inicial:

```
n = int(raw_input("Entero (>=0): "))
termino = 1
```

2. Determinar la condición de parada del bucle. Éste tiene que finalizar cuando se obtengan `n` términos (`n` iteraciones).
   El valor inicial de `k` es el del primer cambio de estado (`k=1`) y como el bucle se ejecuta `n` veces y el paso es 1, el valor final del `range` tiene que ser `n+1`.
3. Cambio de estado:

```
print termino,  #escribe en pantalla el término actual
termino = termino + k + 1
```

Una vez aplicadas las reglas la construcción del bucle `for`, la escritura del programa es inmediata:

```
n = int(raw_input("Entero (>=0): "))
termino = 1
for k in range(1, n+1, 1):
    print termino,  # escribe en pantalla el término actual
    termino = termino + k + 1
print
```

Estado	n	$termino$	$cantidad$
S_0	5	1	1
S_1	5	$1+2=3$	2
S_2	5	$3+3=6$	3
S_3	5	$6+4=10$	4
S_4	5	$10+5=15$	5
S_5	5	$15+6=21$	6

Antes de escribir el bucle previo se había visto que sobraba una de las variables en el estado inicialmente planteado, o `k` o `cantidad`. Otra opción, que es la que ahora se plantea, hubiera sido tomar la variable `cantidad` directamente como variable de control del bucle en lugar de introducir una variable nueva.

En este caso, como la variable de control (`cantidad`) es la primera que cambia de estado, lo más lógico es que el valor a sumar a `termino` en cada iteración sea el valor de la variable `cantidad` en esa misma iteración (en el mismo estado). Esto explica porque se desplazó hacia abajo cada valor de la variable `cantidad`. Y ahora aplicando las reglas de construcción del bucle `for` se obtiene:

```
n = int(raw_input("Entero (>=0): "))
termino = 1
for cantidad in range(2, n+2, 1):
    print termino,  # escribe en pantalla el término actual
    termino = termino + cantidad
print
```

El desplazamiento realizado en la variable `cantidad` es una consecuencia de ser ésta la primera variable que cambia de estado. Este problema no se produce con un bucle `while` porque la actualización de la variable se puede realizar cuando se necesite, sea al principio o al final del bucle. En cualquier caso, tampoco pasaría nada si no se hubiera realizado el desplazamiento, aunque la aplicación directa de las reglas de construcción del `for` habrían llevado a otro `range` y a otro cambio de estado de la variable `termino`.

2.5. Funciones

Nota

El contenido de esta sección está basado en el libro *Introducción a la programación con Python* de la Universidad Jaume I. Podéis encontrar explicaciones más detalladas en el mismo

Podemos pensar que un programa se compone de varias partes, como por ejemplo, obtener los datos de entrada por parte del usuario, realizar ciertos cálculos con los mismos y mostrar el resultado de esos cálculos en la pantalla. Un buen plan de ataque para realizar un programa consiste en descomponer la tarea a realizar en unas cuantas subtareas, que a su vez pueden descomponerse en subtareas más pequeñas o más simples y así sucesivamente. Llegará un momento en que dichas subtareas serán lo suficientemente pequeñas como para que sea sencillo programarlas. Este método se conoce como *diseño descendente* y da lugar a la *programación modular*. La mayoría de los lenguajes de programación cuentan con recursos que les permiten dividir un programa en partes más pequeñas (subprogramas). En el caso del Python estos subprogramas se conocen como *funciones*.

Importante

Una *función* es un fragmento de código de un programa que resuelve un subproblema con entidad propia.

Aunque no nos hayamos dado cuenta, en secciones anteriores ya hemos hecho uso de funciones. Para la entrada por teclado hemos utilizado la función `raw_input` y para los bucles `for` la función `range`.

Lo que vamos a ver en esta sección es cómo definir nuestras propias funciones, de forma que enseñaremos a Python a hacer cálculos que inicialmente no sabía realizar y que sirven para adaptar el lenguaje de programación al tipo de problemas que estamos intentando resolver.

2.5.1. Uso de funciones

Denominaremos activar, invocar o *llamar a una función* a la acción de usarla. Las funciones que hemos aprendido a invocar reciben cero, uno o más argumentos separados por comas y encerrados entre un par de paréntesis y que, habitualmente, devuelven un valor.

```
>>> abs(-3)
3
>>> abs(round(2.45,1))
2.5
```

Podemos llamar a una función desde una expresión. Como el resultado tiene un tipo determinado, hemos de estar atentos a que éste sea compatible con la operación y tipo de los operandos con los que se combina:

```
>>> 1+(abs(-3)*2)
7
>>> 2.5 / abs(round(2.45,1))
1.0
>>> 3+str(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: number coercion failed
```

En el último caso se ha producido un error de tipos porque se ha intentado sumar una cadena, que es el tipo de dato del valor devuelto por str, a un entero.

Observa que los argumentos de una función también pueden ser expresiones:

```
>>> abs(round(1.0/9, 4/(1+1)))
0.11
```

Eso sí, los argumentos deben coincidir con el tipo del parámetro que se espera. En el siguiente ejemplo vemos cómo la función `round` funciona correctamente cuando se le pasa un argumento del tipo adecuado (`float`) y cómo se produce un error cuando se le pasa un `str` en lugar de un `float`.

```
>>> round(7.3)
7.0
>>> round('7.3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a float is required
```

2.5.2. Funciones que ya existen

Python proporciona funciones trigonométricas, logaritmos, etc., pero no están directamente disponibles cuando iniciamos una sesión. Antes de utilizarlas hemos de indicar a Python que vamos a hacerlo. Para ello, importamos cada función de un módulo.

Empezaremos por importar la función seno (*sin*, del inglés *sine*) del módulo matemático (*math*):

```
>>> from math import sin
```

Ahora podemos utilizar la función (indicándole los radianes) en nuestros cálculos:

```
>>> sin(0)
0.0
>>> sin(1)
0.841470984808
```

Inicialmente Python no *sabe* calcular la función seno. Cuando importamos una función, Python *aprende* su definición y nos permite utilizarla. Las definiciones de funciones residen en módulos. Las funciones trigonométricas residen en el módulo matemático. Por ejemplo, la función coseno, en este momento, es desconocida para Python.

```
>>> cos(0)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: cos
```

Antes de usarla, es necesario importarla del módulo matemático:

```
>>> from math import cos
>>> cos(0)
1.0
```

Son muchos los módulos existentes para trabajar en Python. Por ejemplo, el módulo `math` contiene las funciones matemáticas clásicas y `random` funciones que generan números pseudoaleatorios. Hay otros módulos que son más específicos y se centran en problemas concretos, como por ejemplo:

Interfaces gráficas `wxPython`, `pyGtk`, `pyQT`, ...

Bases de datos `MySQLdb`, `pySQLite`, `cx_Oracle`, ...

Imagen `PIL`, `gdmodule`, `VideoCapture`, ...

Ciencias `scipy`, `numarray`, `NumPy`, ...

Videojuegos `Pygame`, `Soya 3D`, `pyOpenGL`, ...

Sonido `pySonic`, `pyMedia`, `pyMIDI`, ...

Existen también módulos de geolocalización, de puertos (USB, serie, paralelo, ...), para programación de dispositivos móviles, Web, programas de mensajería y casi para cualquier cosa que podamos imaginarnos.

2.5.3. Definición de nuevas funciones

Vamos a estudiar el modo en que podemos *definir* nuestras propias funciones Python para *invocarlas* posteriormente.

Definición y uso de funciones con un solo parámetro

Empezaremos definiendo una función muy sencilla, una que recibe un número y devuelve el cuadrado de dicho número. El nombre que daremos a la función es cuadrado. Observa este fragmento de programa:

```
1 def cuadrado(x):
2     return x**2
```

Importante:

Observa que la sentencia perteneciente a la función se encuentra indentada. Todo el código perteneciente a la función debe estar indentado

Acabamos de definir la función `cuadrado` que se aplica sobre un valor al que llamamos `x` y devuelve un número: el resultado de elevar `x` al cuadrado. En el programa aparecen dos nuevas palabras reservadas: `def` y `return`. La palabra `def` es abreviatura de *define* y `return` significa *devuelve* en inglés. Podríamos leer el programa anterior como "define cuadrado de x como el valor que resulta de elevar x al cuadrado".

Importante:

Una vez que hayamos definido la función podremos usarla (invocarla o llamarla) de la misma forma que la funciones ya definidas en Python

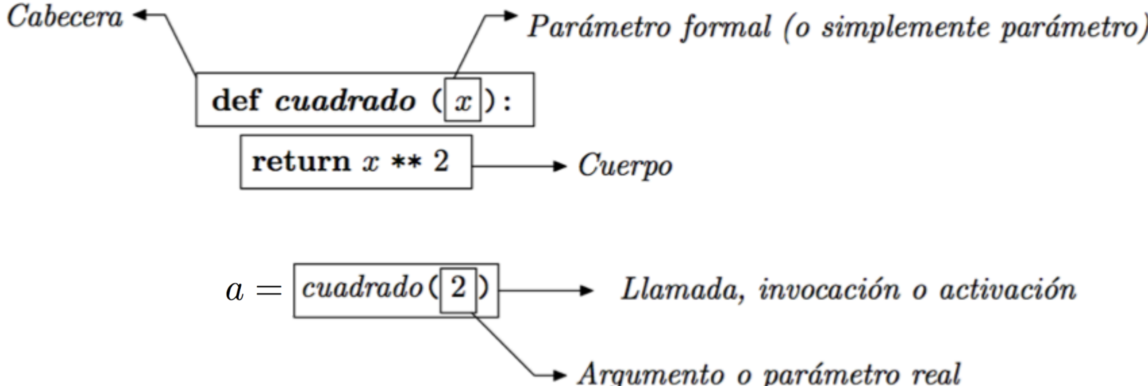
En las líneas que siguen a su definición, la función `cuadrado` puede utilizarse del mismo modo que las funciones predefinidas:

```
1 def cuadrado(x):
2     return x**2
3
4 a = cuadrado(2)
5 print a
6 b = 1 + cuadrado(3)
7 print cuadrado(b * 3)
```

En cada caso, el resultado de la expresión que sigue entre paréntesis al nombre de la función es utilizado como valor de `x` durante la ejecución de `cuadrado`. En la primera llamada (línea 4) el valor es 2, en la siguiente llamada línea 6) es 3 y en la última, 30.

Detengámonos un momento para recalcar algunos conceptos:

- La línea que empieza con `def` es la cabecera de la función y el fragmento de programa que contiene los cálculos que debe efectuar la función se denomina cuerpo de la función.
- Cuando estamos definiendo una función, su parámetro se denomina parámetro formal (aunque, por abreviar, normalmente usaremos el término parámetro, sin más).
- El valor que pasamos a una función cuando la invocamos se denomina parámetro real o argumento.
- Las porciones de un programa que no son cuerpo de funciones forman parte del programa principal: son las sentencias que se ejecutarán cuando el programa entre en acción.
- El cuerpo de las funciones sólo se ejecutará si se producen las correspondientes llamadas.
- El cuerpo de las funciones *debe* estar indentado.



Importante: ¡Definir no es invocar!

Si intentamos ejecutar este programa:

```
1 def cuadrado(x):
2     return x**2
```

no ocurrirá nada en absoluto; bueno, al menos nada que aparezca por pantalla. La definición de una función sólo hace que Python *aprenda* silenciosamente un método de cálculo asociado al identificador `cuadrado`. Nada más. **Definir una función sólo asocia un método de cálculo a un identificador y no supone ejecutar dicho método de cálculo**.

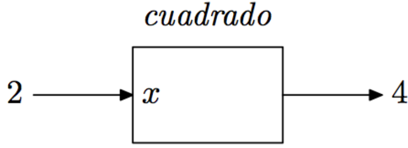
Este otro programa sí muestra algo por pantalla:

```
1 def cuadrado(x):
2     return x**2
3
4 a = cuadrado(2)
5 print a
```

Al invocar la función `cuadrado` (línea 4) se ejecuta ésta. En el programa, la invocación de la última línea provoca la ejecución de la línea 2 con un valor de `x` igual a 2 (argumento de la llamada). El valor devuelto con `return` es almacenado en la variable `a` como efecto de la asignación y, posteriormente mostrado en patalla a causa del `print` de la línea 5.

Las reglas para dar nombre a las funciones y a sus parámetros son las mismas que seguimos para dar nombre a las variables: sólo se pueden usar letras (del alfabeto inglés), dígitos y el carácter de subrayado; la primera letra del nombre no puede ser un número; y no se pueden usar palabras reservadas. Pero, ¡cuidado!: **no debes dar el mismo nombre a una función y a una variable**.

Al definir una función cuadrado es como si hubiésemos creado una *máquina de calcular cuadrados*. Desde la óptica de su uso, podemos representar la función como una caja que transforma un dato de entrada en un dato de salida:



Vamos con un ejemplo más: una función que calcula el valor de x por el seno de x:

```
1 from math import sin
2
3 def xsin (x):
4     return x * sin(x)
```

Lo interesante de este ejemplo es que la función definida, xsin, contiene una llamada a otra función (sin). No hay problema: desde una función puedes invocar a cualquier otra.

Importante: Una confusión frecuente

Supongamos que definimos una función con un parámetro x como esta:

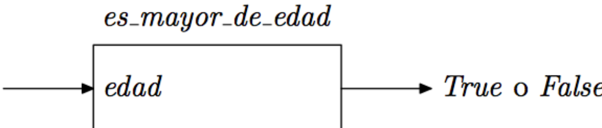
```
1 def cubo (x):
2     return x**3
```

Es frecuente en los aprendices confundir el parámetro x con una variable x. Por tanto, les parece extraño que podamos invocar así a la función:

```
1 def cubo (x):
2     return x**3
3
4 y=1
5 el_cubo = cubo(y)
6 print el_cubo
```

¿Cómo es que ahora llamamos y a lo que se llamaba x? No hay problema alguno. Al definir una función, usamos un identificador cualquiera para referirnos al parámetro. Tanto da que se llame x como y.

En el cuerpo de una función no sólo pueden aparecer sentencias return, también podemos usar estructuras de control: sentencias condicionales, bucles, etc. Lo podemos comprobar diseñando una función que recibe un número y devuelve un booleano. El valor de entrada es la edad de una persona y la función devuelve True si la persona es mayor de edad y False en caso contrario.



Si invocásemos la función con la edad 23 devolvería True, mientras que si la invocásemos con la edad 16 devolvería False.

La función podría tener la siguiente implementación:

```
1 def es_mayor_de_edad(edad):
2     if edad < 18:
3         resultado = False
4     else:
5         resultado = True
6     return resultado
```

Anque también podría tener esta otra:

```
1 def es_mayor_de_edad(edad):
2     if edad < 18:
3         return False
4     else:
5         return True
```

Aparecen dos sentencias `return`: cuando la ejecución llega a cualquiera de ellas, finaliza inmediatamente la llamada a la función y se devuelve el valor que sigue al `return`.

Importante

Una sentencia `return` fuerza a terminar la ejecución de una llamada a función.

Definición y uso de funciones con varios parámetros

No todas las funciones tienen un sólo parámetro. Vamos a definir ahora una con dos parámetros: una función que devuelve el valor del área de un rectángulo dadas su altura y su anchura:

area_rectangulo

→ *altura*
→ *anchura* → *altura* × *anchura*

```
1 def area_rectangulo(altura, anchura):
2     return altura*anchura
```

Observa que los diferentes parámetros en la definición de una función deben separarse por comas. Al usar la función, en su llamada o invocación, los argumentos también deben separarse por comas:

```
1 def area_rectangulo(altura, anchura):
2     return altura*anchura
3
4 area = area_rectangulo(3,4)
5 print area
```

Definición y uso de funciones sin parámetros

Vamos a considerar ahora cómo definir e invocar funciones sin parámetros. En realidad hay poco que decir: lo único que debes tener presente es que es obligatorio poner paréntesis a continuación del identificador, tanto al definir la función como al invocarla.

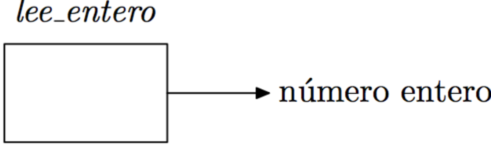
En el siguiente ejemplo se define y usa una función que lee de teclado un número entero:

```
1 def lee_entero():
2     return int(raw_input())
3
4 a = lee_entero()
5 print a
```

Importante

Recuerda: al llamar a una función los paréntesis **son obligatorios**

Podemos representar esta función como una caja que proporciona un dato de salida sin ningún dato de entrada:



Veamos otro ejemplo: una función que lee un número por teclado asegurándose de que éste sea positivo:

```
1 def lee_entero_positivo():
2     numero = int(raw_input())
3     while numero < 0:
4         print "El numero debe ser positivo"
5         numero = int(raw_input())
6     return numero
7
8 a = lee_entero_positivo()
9 print a
```

Una posible aplicación de la definición de funciones sin argumentos es la presentación de menús con selección de opción por teclado. Esta función, por ejemplo, muestra un menú con tres opciones, pide al usuario que seleccione una y se asegura de que la opción seleccionada es válida. Si el usuario se equivoca, se le informa por pantalla del error:

```
1 def menu():
2     opcion = ''
3     while not ('a' <= opcion <= 'c'):
4         print "Cajero automatico:"
5         print "a) Ingresar dinero"
6         print "b) Sacar dinero"
7         print "c) Consultar saldo"
8         opcion = raw_input("Elige una opcion:")
9         if not (opcion >= 'a' and opcion <= 'c'):
10            print "Elige una opcion valida: a, b o c:"
11    return opcion
12
13 accion = menu()
14 print accion
```

2.5.4. Variables locales y variables globales

Observa que en el cuerpo de las funciones es posible definir y usar variables. Vamos a estudiar con detenimiento algunas propiedades de las variables definidas en el cuerpo de una función y en qué se diferencian de las variables que definimos fuera de cualquier función, es decir, en el denominado programa principal.

Veamos un programa que calcula el área de un triángulo conocida la longitud de sus lados. Para ello utilizaremos la siguiente fórmula $\sqrt{s*(s-a)*(s-b)*(s-c)}$ donde $s = (a+b+c)/2$ y a, b y c son la longitud de sus lados:

```
1 from math import sqrt
2
3 def area_triangulo(a,b,c):
4     s = (a+b+c)/2.0
5     return sqrt(s*(s-a)*(s-b)*(s-c))
6
7 area = area_triangulo(1,3,2.5)
8 print area
9 print s
```

Ahora viene lo importante: la variable `s` sólo existe en el cuerpo de la función. Fuera de dicho cuerpo, `s` no está definida. El programa provoca un error al ejecutarse porque intenta acceder a `s` desde el programa principal. Cuando se ejecuta, aparece esto por pantalla:

```
1.1709371247
Traceback (innermost last):
  File "area_triangulo.py", line 9, in ?
    print s
NameError: s
```

La primera línea mostrada en pantalla es el resultado de ejecutar la línea 8 del programa. La línea 8Python incluye la impresión en pantalla del resultado de la llamada a `area_triangulo`, así que el flujo de ejecución ha pasado por la línea 4 y `s` se ha creado correctamente. De hecho, se ha accedido a su valor en la línea 5 y no se ha producido error alguno. Sin embargo, al ejecutar la línea 9 se ha producido un error por intentar mostrar el valor de una variable inexistente: `s`. La razón es que `s` se ha creado en la línea 4 y se ha destruido tan pronto ha finalizado la ejecución de `area_triangulo`.

Importante

Las variables que sólo existen en el cuerpo de una función se denominan *variables locales*. En contraposición, el resto de variables se llaman *variables globales*.

También los parámetros formales de una función se consideran variables locales, así que no puedes acceder a su valor fuera del cuerpo de la función.

¿Y cuándo se crean `a`, `b` y `c`? ¿Con qué valores? Cuando llamamos a la función con, por ejemplo, `area_triangulo(1, 3, 2.5)`, ocurre lo siguiente: los parámetros `a`, `b` y `c` se crean como variables locales en la función y apuntan a los valores 1, 3 y 2.5, respectivamente. Se inicia entonces la ejecución del cuerpo de `area_triangulo` hasta llegar a la línea que contiene el `return`. El valor que resulta de evaluar la expresión que sigue al `return` se devuelve como resultado de la llamada a la función. Al acabar la ejecución de la función, las variables locales `a`, `b` y `c` dejan de existir (del mismo modo que deja de existir la variable local `s`).

2.5.5. Procedimientos: funciones sin devolución de valor

Un procedimiento es un mecanismo proporcionado por los lenguajes de programación para describir una tarea (o una acción) que no produce un resultado y que, de forma análoga a cómo se utilizan las funciones, se puede activar, invocar o llamar para realizar dicha tarea (o acción).

Por la definición dada de procedimiento y lo que hemos visto de funciones, es fácil darse cuenta de que no puede haber grandes diferencias entre ambos, al menos en lo que a definición y uso se refiere. Tanto es así, que en varios lenguajes de programación (entre los que se encuentra el Python) *un procedimiento es directamente una función sin valor de retorno*. De forma que para definir e invocar un procedimiento se define e invoca una función de este tipo.

Importante

Al contrario de lo que ocurre con las funciones que retornan un valor, la invocación a un procedimiento (función sin retorno) nunca se puede realizar dentro de una expresión

Una función siempre retorna un valor y, por tanto, dispone de un mecanismo simple para transferir resultados hacia el programa u otras funciones (el valor retornado). Sin embargo, un procedimiento es una función sin retorno y, por tanto, se requiere que haya alguna otra alternativa de intercambio de información. Así, para que un procedimiento pueda trasladar resultados al programa tras su ejecución (o a otros procedimientos o funciones) debe modificar los valores de uno o más de sus parámetros reales durante su ejecución.

En Python los números y las cadenas de caracteres son objetos inmutables; mientras que las listas y los ficheros (que se verán más adelante en la asignatura) son objetos mutables (se pueden modificar). Y así, los únicos parámetros reales que la ejecución de un procedimiento puede modificar son los correspondientes a objetos mutables (listas y ficheros).

De lo dicho anteriormente, pudiera parecer que en Python no tiene sentido alguno definir e invocar procedimientos sin parámetros, o donde todos ellos se sustituyan por objetos inmutables

(números o cadenas de caracteres). Desde luego con el objetivo de tratar de hacer alguna modificación a sus argumentos así es, porque no se puede. Sin embargo, hay un caso muy particular en el que sí puede tener sentido: cuando únicamente se pretende mostrar información en la pantalla. Por ejemplo, procedimiento que muestra en pantalla la tabla de multiplicar de un número que recibe como parámetro:

```
1 # definicion del procedimiento
2 def tabla_del(n):
3     for i in range(1,11):
4         print n,"x",i,"=",n*i
5
6 # programa principal
7 tabla_del(7)
```

Debe quedar claro que mostrar información mediante un print (o varios) nada tiene que ver con que la función retorne un resultado. Es más, la función no retorna nada (no hay sentencia return); es decir, es un procedimiento.

2.6. Tipos y estructuras de datos básicas: listas y cadenas

La mayor parte de los lenguajes de programación aparte de trabajar con los tipos básicos de datos tales como los números, los caracteres, etc., implementan otras estructuras que permiten trabajar con cantidades de información más amplias. En matemáticas o en física, estamos acostumbrados a tratar con vectores o matrices, que son una colección de valores agrupados bajo un solo nombre. Por ejemplo, para representar las coordenadas de un punto en el espacio se suelen usar vectores.

$$\bar{v} = (3, 4, 0)$$

o para representar los términos de un sistema de ecuaciones podemos hacer uso de las matrices.

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 7 \\ 2 & 1 & 0 \\ 1 & 3 & 2 \end{pmatrix}$$

El vector v o la matriz $\mathbf{A}$ representan a todo el conjunto de datos, pero también se puede referenciar a cada uno de los datos particulares, en este caso, mediante índices.

$$v_2 = 4\,, \quad A_{1,3} = 7$$

Los lenguajes de programación modernos implementan varios tipos de estructuras que agrupan colecciones de datos bajo un solo nombre de variable. Las *listas*, los *arrays*, las *cadenas de texto*, los *diccionarios* o los *conjuntos* son algunos de los tipos estructurados más usuales.

Las listas de Python engloban conjuntos de datos que pueden ser de distintos tipos (y en este sentido pueden ser usadas de forma similar al tipo de datos denominado *estructura* o *registro* en otros lenguajes, y del que Python carece), mientras que los arrays agrupan datos que necesariamente han de ser todos del mismo tipo, y a los que se da el nombre de *componentes*. Las *cadenas*, de las que ya hemos visto numerosos ejemplos, por su uso habitual tienen un tratamiento especial en todos los lenguajes. En cuanto a los tipos *diccionario* y *conjunto* no se verán en esta asignatura.

En Python, el tipo *lista* se denomina `list` y permite implementar directamente agrupaciones de datos en los que cada dato ocupa una posición dentro de la agrupación. Su longitud es variable y de hecho, se pueden añadir o quitar elementos en cualquier momento. Cada elemento de la lista puede ser de cualquier tipo (incluso otras listas). El acceso a cada elemento particular se realiza a través de índices (que comienzan en la posición 0 y no en 1 como es tradicional en física o matemáticas).

Las *cadenas* tienen su propio tipo asociado `str` y en Python se implementan como un tipo especial de lista, por lo que comparten la mayor parte de sus características.

Veamos con detenimiento estos tipos estructurados de datos.

2.6.1. El tipo list

El tipo list se introduce simplemente encerrando entre corchetes la lista de datos, separados por comas. Por ejemplo

```
>>> lista = ['velocidad', 17, 3.1416]
```

Esto crea un "recipiente" al cual se le pone el nombre lista, y en dicho recipiente se reserva espacio para tres datos. En cada uno de los "huecos" reservados se pone cada uno de los datos, en este caso la cadena 'velocidad' y los números 17 y 3.1416.

También es posible representar el vector v anterior mediante una lista. En este caso todos los elementos son del mismo tipo.

```
>>> v = [3, 4, 0]
```

Una vez creada ¿qué podemos hacer con una lista? Algunas de las operaciones básicas que Python trae "de fábrica" son:

- Contar cuántos elementos tiene la lista. Basta usar la función len

```
>>> len(lista)
3
```

- Calcular la suma de todos los elementos, usando la función sum, si todos los elementos son numéricos (enteros o reales):

```
>>> sum(v)
7
```

- Encontrar el máximo o el mínimo de los elementos (igualmente tiene sentido si todos son numéricos). Para ello usamos las funciones max y min:

```
>>> max(v)
4
>>> min(v)
0
```

- Imprimir la lista completa, con print:

```
>>> print lista
['velocidad', 17, 3.1416]
>>> print v
[3, 4, 0]
```

- Ordenar la lista, devolviendo como resultado una nueva lista, que podemos imprimir o almacenar en otra variable (la lista original no se modifica, mantendrá el orden que tenía):

```
>>> print v
[3, 4, 0]
>>> print sorted(v)
[0, 3, 4]
>>> print v
[3, 4, 0]
>>> v_ordenada = sorted(v)
>>> print v_ordenada
[0, 3, 4]
```

- Concatenar listas. Si usamos el operador + entre dos listas, lo que ocurre es que se crea una nueva lista concatenando ambas. Esta nueva lista se puede imprimir o asignar a otra variable. Las listas originales no son modificadas. Ejemplo:

```
>>> print v
[3, 4, 0]
>>> print v+v
[3, 4, 0, 3, 4, 0]
>>> otra = v + [100]
>>> print otra
[3, 4, 0, 100]
>>> print v
[3, 4, 0]
```

  Observa que en el ejemplo anterior, el valor `[100]` es otra lista, con un solo elemento, por lo que el operador + concatenará ambas listas. Si hubiéramos puesto en cambio `v+100` estaríamos intentando sumar un tipo lista (la variable `v`) con un tipo entero (el dato `100`). El operador + no funciona si sus dos operandos no son del mismo tipo, por lo que tendíamos un error.

- Asignar otro nombre a la lista, mediante el operador de asignación =. Al hacer por ejemplo `q=v`, tendremos dos nombres (`q` y `v`) para referirnos a la misma lista:

```
>>> print v
[3, 4, 0]
>>> q = v
>>> print q
[3, 4, 0]
```

  Más adelante en este tema veremos qué ocurre exactamente con el operador de asignación y qué consecuencias tiene al tratar con listas.

Hay más operaciones predefinidas para listas, pero estas son suficientes. La cuestión es ¿y si queremos hacer algo con los datos de una lista que no puede hacerse con ninguna de las funciones predefinidas? Por ejemplo, ¿y si queremos cambiar cada elemento de la lista por su triple? o bien ¿y si queremos calcular la suma de los cuadrados de los elementos? etcétera...

La respuesta es que en realidad podemos hacer cualquier cosa con los datos que hay en una lista porque cada uno de esos datos sigue siendo accesible por separado, además de poder accederse a la lista "como un conjunto" a través de su nombre. Veamos esto con detalle.

2.6.2. Acceso a los elementos individuales de la lista

Cada uno de los elementos de la lista `p` está accesible a través de una variable llamada `p[i]`, donde `i` es el denominado *índice* del elemento. Es como cuando en matemáticas decimos que un vector **v** tiene por componentes $v_1, v_2, \ldots, v_i, \ldots, v_n$, siendo n el número de componentes del vector.

En Python, al igual que en la mayoría de los lenguajes de programación (Java, C, etc.), y a diferencia de lo habitual en matemáticas, los índices comienzan a numerarse desde cero, y por tanto los n elementos de la lista `p` serían accesibles mediante los nombres `p[0]` (primer elemento), `p[1]` (segundo elemento), etc... hasta `p[n-1]` (último elemento).

Por ejemplo, considera la asignación `p=[100, 150, 30, 15]`, que crea una lista con cuatro elementos. La variable `p` es un nombre para esa lista. Representaremos la situación mediante la siguiente figura:

p → | 100 | 150 | 30 | 15 |

Cada uno de los datos a los que apunta `p` puede ser accedido individualmente mediante la sintaxis `p[0]`, `p[1]`, etc, como muestra la siguiente figura:

	p[0]	p[1]	p[2]	p[3]
p →	100	150	30	15

La expresión `p[i]` puede usarse como parte de cualquier expresión, y así por ejemplo `p[0]+p[1]*2` produciría el resultado `400`, la expresión `a=p[2]` asignaría el entero `30` a la variable `a`, etc.

El tipo de `p[i]` es el tipo del dato allí contenido, de modo que por ejemplo `type(p[0])` sería `int`. Recuerda que en Python cada elemento de la lista podría ser de un tipo diferente, aunque en esta asignatura no usaremos mucho esta característica.

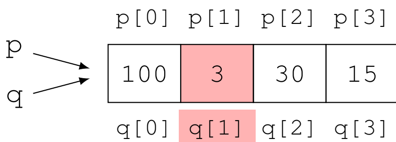
Observa que si tenemos dos nombres diferentes para una misma lista (por ejemplo, si hacemos `q=p`, `q` se convierte en un nuevo nombre para la lista `p`), entonces podremos usar cualquiera de esos nombres para acceder a los elementos que hay en ella. Es decir, tras la asignación `q=p`, tendríamos la situación de la figura:

	p[0]	p[1]	p[2]	p[3]
p, q →	100	150	30	15
	q[0]	q[1]	q[2]	q[3]

Por lo que `p[1]` y `q[1]` ambos se refieren al mismo dato, el entero `150`.

Las listas son objetos mutables

Esto es muy importante si usamos la expresión `p[i]` al lado izquierdo de una asignación, puesto que en este caso *estaremos modificando* el dato almacenado en esa posición de la lista. Es decir, si hacemos `q[1]=3`, estaremos cambiando lo que había en `q[1]` (el `150`) por un `3`. La nueva situación sería:

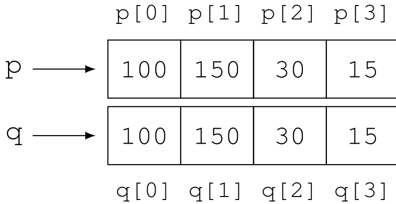


¡Indirectamente hemos cambiado el valor de `p`! Si ahora usamos `p[1]` en cualquier expresión (por ejemplo `print p[1]`) encontraremos que vale 3.

Si lo que queremos es crear una nueva lista `q` que inicialmente contenga los mismos valores que `p`, pero que sea un "objeto" independiente, de modo que podamos modificar `q` sin afectar a `p`, una posibilidad para lograrlo es la siguiente:

```
>>> p = [100, 150, 30, 15]
>>> q = list(p)
>>> print p
[100, 150, 30, 15]
>>> print q
[100, 150, 30, 15]
```

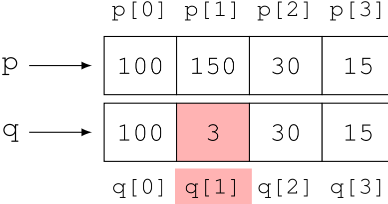
La función `list()` crea un nuevo "objeto" del tipo list, y carga en él los valores iniciales que le pasemos como parámetro, que ha de ser de tipo secuencia. En este caso le estamos pasando una lista `p`. A la nueva lista que resulta, que es una copia de la que había en `p`, le damos el nombre `q`. Al imprimir ambas vemos que tienen los mismos valores, sin embargo se trata de dos objetos diferentes. La situación sería la de la figura siguiente:



Si ahora modificamos cualquier elemento de q, esto no afecta a los elementos de p, como podemos ver:

```
>>> q[1] = 3
>>> print p
[100, 150, 30, 15]
>>> print q
[100, 3, 30, 15]
```

ya que p y q se refieren a diferentes listas.



Comparación de listas

De la explicación anterior se deduce que no es la misma situación tener dos listas almacenadas en distintas posiciones de memoria con los mismos elementos que tener una única lista con dos nombres.

La primera situación se resuelve mediante el operador de igualdad ==. Este comparador lo que hará será en primer lugar comparar la longitud de las listas, y si resultan iguales comparar entonces uno a uno cada elemento. Si todos son iguales, el resultado será True. Así, si hubiéramos mirado si p==q antes de cambiar q[1], el resultado habría sido True, mientras que si lo miramos de nuevo tras cambiarlo (siendo q una copia independiente de q), el resultado sería False.

El comparador == por tanto nos dice si dos listas son *iguales* elemento a elemento, pero no nos dice si son *la misma*. Puede ser que las dos listas sean iguales porque son en realidad la misma (como ocurría con p y q tras hacer q=p), pero pueden ser iguales también si, aún siendo diferentes objetos, contienen la misma secuencia de datos.

Si queremos descubrir si las listas son en realidad la misma, podemos usar el comparador is, en la expresión p is q. Este operador dará como resultado True sólo si ambas variables apuntan al mismo objeto. Si apuntan a objetos diferentes, el resultado será False, aunque pueda darse el caso de que ambos objetos sean iguales elemento a elemento.

Es decir:

```
>>> p = [100, 150, 30, 15]
>>> q = p
>>> r = list(p)
>>> p == q
True
>>> p == r
True
>>> p is q
True
>>> p is r
False
```

La situación creada por el código anterior se muestra en la siguiente figura:

```
p ─┐    ┌─────┬─────┬────┬────┐
   ├──▶ │ 100 │ 150 │ 30 │ 15 │
q ─┘    └─────┴─────┴────┴────┘
        ┌─────┬─────┬────┬────┐
r ────▶ │ 100 │ 150 │ 30 │ 15 │
        └─────┴─────┴────┴────┘
```

Si en este momento hiciéramos la asignación `p[1]=3`, intenta adivinar qué resultado darían las comparaciones `p is q`, `p is r`, `p==q` y `p==r`.

Añadir y quitar datos en una lista

Supongamos que a la lista `p` de los ejemplos anteriores le queremos añadir otro dato, por ejemplo el número 12. La sintaxis para lograrlo es:

```
>>> p.append(12)
```

Importante

Esta sintaxis es diferente a lo que vimos hasta ahora. Ocurre que Python es un lenguaje *orientado a objetos* y si bien no vamos a ahondar en esta característica del lenguaje en esta asignatura, sí que nos encontraremos en algunos lugares (por ejemplo ahora), con la sintaxis relacionada con el manejo de objetos.

En pocas palabras, un objeto es un tipo de dato que agrupa bajo una misma estructura datos y funciones para trabajar sobre esos datos. A los datos se les denomina comunmente *atributos*, y a las funciones *métodos*. Los *métodos* son las únicas funciones que tienen vía libre para modificar, si se requiere, los *atributos*. Aunque no lo habíamos comentado hasta ahora, todos los datos en Python son objetos (el tipo entero, el float, las cadenas, las listas, etc.) Para llamar a un método un objeto se usa la sintaxis que acabamos de ver en el último ejemplo, es decir, primero el nombre de la variable que apunta al objeto, después un punto y después el nombre del método, seguido de unos paréntesis, dentro de los cuales van los parámetros si es que los necesita.

Como vemos la sintaxis es igual a la vista para llamar a una función, sólo que delante del nombre de la función, separada por un punto, va el nombre de una variable que señala al objeto.

En este caso, el objeto de tipo `list` tiene el método `append(12)` como una forma de modificar los contenidos de la lista, añadiendo un 12 al final de los que ya había. El resultado será que la lista a la que `p` se refiere habrá credido y tendrá ahora 5 elementos en lugar de 4. Podemos imprimirla y comprobar su nueva longitud:

```
>>> print p
[100,150,30,15,12]
>>> len(p)
5
```

La situación, gráficamente, es:

```
p ─┐    ┌─────┬─────┬────┬────┬────┐
   ├──▶ │ 100 │ 150 │ 30 │ 15 │ 12 │
q ─┘    └─────┴─────┴────┴────┴────┘
        ┌─────┬─────┬────┬────┐
r ────▶ │ 100 │ 150 │ 30 │ 15 │
        └─────┴─────┴────┴────┘
```

Observa que, ya que q se refería a la misma lista que p, cualquier cambio que hagamos en la lista a través de p se verá reflejado también cuando la manejemos a través de q. En concreto len(q) también será 5. En cambio r se refería a una lista diferente (aunque inicialmente tenía los mismos datos). El añadido de un dato a través de p no ha afectado a r, como se ve en la figura.

Otro método útil que un objeto tipo list tiene es extend(). A través de este método podemos hacer crecer la lista añadiéndole varios elementos en lugar de uno solo (como hacía append()). Para ello, a extend() se le pasa como parámetro otra lista (o secuencia), conteniendo los elementos que queremos añadir. Por ejemplo:

```
>>> p.extend([1, 2, 3])
>>> len(p)
8
>>> print p
[100, 150, 30, 15, 12, 1, 2, 3]
```

Además de poder pasarle una lista de valores directamente, como en el ejemplo anterior, podríamos haberlo hecho a través de otra variable. Por ejemplo, si quisiéramos añadir a p los elementos que hay en la lista r, podríamos poner p.extend(r). En este caso se creará una copia de la lista r y esa se añadirá a p.

Si se quieren añadir datos a una lista, pero no en el final sino en cualquier punto intermedio, se puede conseguir haciendo uso del método insert, que tiene como parámetros la posición y el elemento que queremos *insertar* en la lista. Existen otras posibilidades, como el uso de "slices", pero su explicación queda fuera de las pretensiones de esta asignatura.

Para **eliminar** datos de una lista, tenemos dos posibilidades:

- Si conocemos en qué posición de la lista está el dato que queremos borrar, usamos el método pop(posicion) de la lista. Recuerda que las posiciones comienzan a numerarse desde 0 y llegan hasta len(lista)-1. Si no se especifica la posición (es decir, usamos simplemente pop() sin parámetros), se eliminará el último de la lista.

  Además de eliminar el dato, su valor es retornado, de modo que lo leemos a la vez que lo eliminamos. Por ejemplo:

```
>>> p = [1, 3, 5, 7, 11, 13]
>>> n = p.pop()
>>> print n
13
>>> print p
[1, 3, 5, 7, 11]
>>> n = p.pop(0)
>>> print n
1
>>> print p
[3, 5, 7, 11]
```

- Si conocemos el valor del dato, pero no en qué posición está en la lista, podemos usar el método remove(dato). Si el dato se encuentra, es eliminado (si aparece varias veces, sólo se elimina la primera). No obstante el uso de este método es un poco arriesgado, ya que si el dato no estuviera en la lista, se generaría una excepción. Por ejemplo:

```
>>> p = [1, 3, 5, 7, 11, 13]
>>> p.remove(5)
>>> print p
[1, 3, 7, 11, 13]
>>> p.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ValueError: list.remove(x): x not in list
```

Para evitar el error, podríamos comprobar si el dato efectivamente está, antes de intentar eliminarlo. Para ello Python tiene el operador `in` que se usa en la expresión booleana: `dato in lista`, y que devuelve `True` si el dato está en la lista y `False` si no. Así que si hacemos `if 2 in p:` antes de intentar `p.remove(2)`, evitaríamos que se generase la excepción.

2.6.3. El operador de corte

En Python es posible obtener un subconjunto de los elementos de una lista o cadena y almacenarlos en otra variable. Para ello se utiliza el operador de corte, el signo de puntuación ':' , junto con varias expresiones de tipo entero y los corchetes. Dado se trata de un operador, se crea un objeto nuevo en memoria, esto es importante si el resultado se va a asignar a otra variable. Su asociatividad es de izquierda a derecha. La sintaxis más general es `nombreLista[inicio:fin:paso]`, en donde inicio es la posición del primer elemento de `nombreLista` que estará en el resultado, fin la posición del primer elemento que *no estará* en el resultado y paso es la diferencia entre los índices de dos elementos consecutivos del resultado en su ubicación original en `nombreLista`. Conviene tener en cuanta las siguientes "reglas" que permiten escribir y entender fácilmente expresiones en las que interviene el operador de corte:

- Por defecto (si se omiten) inicio es cero, fin es igual a `len(nombreLista)` y paso es uno.
- Si se omiten inicio y fin pero no paso y este es menor que cero, inicio es la longitud menos uno y fin es cero.
- Si solo hay un ':', entonces las expresiones a ambos lados del mismo son inicio y fin.
- En Python listas y cadenas se pueden indexar usando índices negativos. En ese caso se entiende que se comienza a contar por el final y en uno. Por ejemplo `nombreLista[len(nombreLista)-1]` y `nombreLista[-1]` representan el último elemento de la lista, -2 sería el índice del penúltimo (dos menos que la longitud), etc.

De esta manera, ciertas selecciones de elementos de una lista o cadena se pueden escribir de una forma muy compacta. Ejemplos:

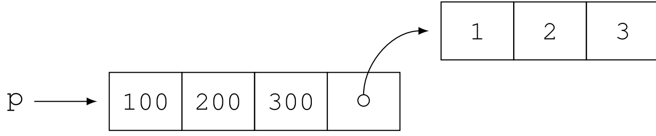
- `nombreLista[:]` es una *copia* de toda la lista: por defecto inicio es 0, fin es len(nombreLista), paso es 1.
- `nombreLista[:-1]` es una lista que contiene todos los elementos de `nombreLista` menos el último. En general `nombreLista[:-n]`, con `n >= 1` es una lista con los elementos de `nombreLista` menos los `n` últimos.
- `nombreLista[-n:]` por el contrario es una lista formada por los `n` últimos de `nombreLista`.
- `nombreLista[:n]` es la lista formada por los n primeros elementos de `nombreLista`.
- `nombreLista[::-1]` es una lista en la que están todos los elementos de `nombreLista` en orden inverso, del final al principio. Este caso es menos claro: como el paso es negativo y se omite inicio y fin, estos son respectivamente -1 y 0. Por lo tanto son todos los elementos de `nombreLista` del final al principio recorridos en orden inverso.

2.6.4. Listas que contienen listas

Una lista es una serie de datos, y cada uno de estos datos puede ser de cualquiera de los tipos que ya hemos visto. Por tanto, un dato contenido en una lista podría ser otra lista. Por ejemplo:

```
>>> p = [100, 200, 300, [1, 2, 3]]
>>> print p
[100, 200, 300, [1, 2, 3]]
>>> print p[0]
100
>>> print p[2]
300
>>> print p[3]
[1, 2, 3]
```

La estructura de datos que hemos creado en el código anterior se puede representar gráficamente así:



¿Qué valor crees que obtendremos al hacer `len(p)`? Compruébalo en el intérprete.

Como vemos, el último elemento de `p` no contiene un dato propiamente dicho, sino una "referencia" hacia otro dato que es la lista `[1, 2, 3]`. Esta lista de alguna forma es "anónima" en el sentido de que no tiene nombre (no hay una variable que se refiera a ella). Si quisiéramos entonces modificar el `1` que hay en ella y cambiarlo por un `5` ¿cómo podríamos lograrlo? Piénsalo un poco antes de continuar leyendo.

Hemos dicho que no hay una variable que se refiera a la lista `[1,2,3]`, pero esto no es del todo cierto. En realidad `p[3]` es un nombre que se refiere a esa lista, como hemos visto al hacer `print p[3]`. Y si `p[3]` es una lista, podremos aplicarle las mismas operaciones que a cualquier otra lista. Por ejemplo, podemos hacer `len(p[3])` (¿qué valor devolvería?). Y en particular, también podemos poner unos corchetes tras su nombre y acceder así a cualquiera de sus elementos.

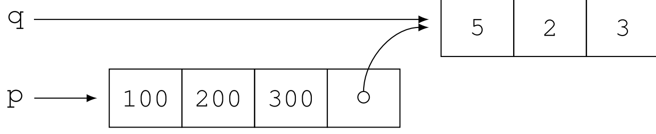
De modo que para cambiar el `1` por un `5` la sintaxis sería:

```
>>> p[3][0] = 5
>>> print p
[100, 200, 300, [5, 2, 3]]
```

Observa que en el ejemplo anterior, la lista `[1, 2, 3]` sólo puede ser accedida a través de `p[3]`, ya que no tenemos ninguna otra variable que se refiera a esa lista. Pero podríamos tenerla. Considera el siguiente código:

```
>>> q = p[3]
```

Tras la asignación anterior, `q` se convierte en otro nombre para la lista `[1, 2, 3]`, como refleja la siguiente figura:



A partir de esta figura, se comprende que si hacemos por ejemplo `q[1]=27` estaremos modificando indirectamente el valor de `p[3][1]`, lo cual se manifestará al imprimir `p`.

```
>>> q[1] = 27
>>> print q
[5, 27, 3]
>>> print p
[100, 200, 300, [5, 27, 3]]
```

A la misma situación de la figura anterior habríamos llegado también si hubieramos inicializado p y q de este otro modo:

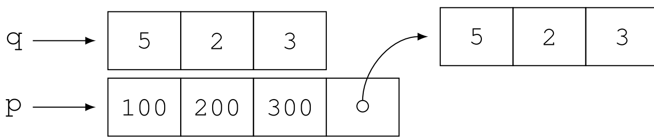
```
>>> q = [5, 2, 3]
>>> p = [100, 200, 300, q]
>>> print p
[100, 200, 300, [5, 2, 3]]
```

Fíjate que al dar la lista de valores de p hemos usado q como uno de ellos. El efecto es que dentro de p[3] se almacena una referencia a la misma lista a la que se refiere q. Es decir, la misma situación de la figura anterior. Las modificaciones que hagamos a través de q tendrán efecto en p.

Si hubiéramos querido tener en p[3] una referencia a una copia de q, en lugar de una referencia a la misma lista que q) una forma de lograrlo habría sido:

```
>>> q = [5, 2, 3]
>>> p = [100, 200, 300]
>>> p.append(q)
>>> print p
[100, 200, 300, [5, 2, 3]]
```

Aunque el resultado parece el mismo, la situación ahora sería la de la figura siguiente:



2.6.5. Bucles para recorrer listas

Como hemos dicho antes, Python trae algunas funciones útiles para realizar cálculos elementales sobre listas. Por ejemplo, la función sum(lista) nos devuelve la suma de todos los elementos de una lista. Sin embargo, hay otras muchas operaciones que podríamos necesitar, y que no vienen con el lenguaje. Por ejemplo, ¿y si quisiéramos calcular la suma de los cuadrados? ¿O crear una nueva lista cuyos elementos sean las raíces cuadradas de los elementos de la lista dada? etcétera...

Obviamente la solución consiste en implementar nosotros mismos los cálculos requeridos en un bucle. En cada iteración (repetición) del bucle, obtendremos un elemento de la lista a procesar, con el que realizaremos nuestros cálculos.

Ya hemos visto que cada elemento de la lista es accesible a través de un índice, que va desde 0 hasta len(lista)-1, por lo que una estrategia directa es utilizar un bucle for sobre una variable i que recorra el rango 0 a len(lista)-1, y dentro de él usar esa variable como índice para acceder a lista[i]. Precisamente este rango es el que obtenemos mediante range(len(lista)), por lo que una implementación directa de estas ideas sería la siguiente:

```
1 # Inicializamos la lista p con una serie de numeros
2 p = [4, 9, 27, 100, 110]
3
4 # Calculemos la suma de los cuadrados
5 suma_2 = 0      # de momento vale cero
6 for i in range(len(p)):
7     suma_2 = suma_2 + p[i]**2
8 print "Lista p =", p
```

```
 9 print "La suma de los cuadrados vale", suma_2
10
11 # Creemos ahora una nueva lista que contenga las raices cuadradas de cada
12 # uno de los datos de la lista p
13 # Para calcular raices cuadradas necesitamos la funcion sqrt del modulo math
14 import math
15
16 # La nueva lista con los resultados, inicialmente vacia:
17 raices_cuadradas = []
18
19 # Rellenemos la lista anterior, mediante un bucle
20 for i in range(len(p)):
21     raices_cuadradas.append(math.sqrt(p[i]))
22 print "Raices cuadradas:", raices_cuadradas
```

El resultado de ejecutar el código anterior sería:

```
Lista p = [4, 9, 27, 100, 110]
La suma de los cuadrados vale 22926
Raices cuadradas: [2.0, 3.0, 5.196152422706632, 10.0, 10.488088481701515]
```

Esta forma de acceder a cada elemento de una lista, a través de su índice, es la forma que permiten casi todos los lenguajes de programación (C, C++, java, etc.).

Importante

Cuando se accede a los elementos de una lista usando un índice, se puede cambiar el valor de estos.

Python proporciona otra forma aún más breve para recorrer los elementos de una lista. Se trata de usar la sintaxis siguiente:

```
1 for variable in lista:
2   sentencia1
3   sentencia2
4   etc...
```

Lo que ocurre es que la variable `variable` va pasando por todos los valores de la lista, por orden. En la primera iteración del bucle `variable` toma el valor de `lista[0]`, en la siguiente iteración toma el valor de `lista[1]` y así sucesivamente hasta agotar todos los valores de la lista. Observa que con esta sintaxis no se requieren índices ni corchetes.

Usemos esta sintaxis para implementar de nuevo el cálculo de la suma de los cuadrados y la lista con las raíces cuadradas:

```
 1 # Inicializamos la lista p con una serie de numeros
 2 p = [4, 9, 27, 100, 110]
 3
 4 # Calculemos la suma de los cuadrados
 5 suma_2 = 0     # de momento vale cero
 6 for dato in p:
 7     suma_2 = suma_2 + dato**2
 8 print "Lista p =", p
 9 print "La suma de los cuadrados vale", suma_2
10
11 # Creemos ahora una nueva lista que contenga las raices cuadradas de cada
12 # uno de los datos de la lista p
13 # Para calcular raices cuadradas necesitamos la funcion sqrt del modulo math
14 import math
15
16 # La nueva lista con los resultados, inicialmente vacia:
17 raices_cuadradas = []
```

```
18
19 # Rellenemos la lista anterior, mediante un bucle
20 for dato in p:
21     raices_cuadradas.append(math.sqrt(dato))
22 print "Raices cuadradas:", raices_cuadradas
```

Como ves, esta sintaxis es más clara, pero ten en cuenta que es una característica que en otros lenguajes puede no estar presente (el ejemplo más notable es el C).

Importante

Cuando se accede a los elementos de una lista de esta forma, **no** se puede cambiar el valor de estos.

2.6.6. Listas y funciones

Una función puede recibir como parámetro una lista, y también devolver una lista como resultado. Así, por ejemplo, podemos convertir en una función uno de los ejemplos antes vistos, el que calcula la suma de los cuadrados de los elementos de una lista.

```
1 def suma_de_los_cuadrados(lista):
2     """Dada una lista como parametro, que contiene una serie de
3     numeros, la funcion retorna la suma de los cuadrados"""
4     suma_2 = 0 # De momento la suma es cero
5     for dato in lista:
6         suma_2 = suma_2 + dato**2
7     # Al salir del bucle tenemos el resultado, que retornamos
8     return suma_2
9
10 # Para probar el ejemplo anterior, creare una lista
11 p = [1, 2, 3, 4]
12 resultado = suma_de_los_cuadrados(p)
13 print "La lista es:", p
14 print "La suma de los cuadrados es", resultado
```

Al ejecutar el programa anterior, el resultado será 30. Si necesitáramos una función que en lugar de la suma de los cuadrados hiciera la suma de los cubos, bastaría cambiar el `**2` de la línea 6 por un `**3`. Esto nos lleva a plantearnos la siguiente generalización: Escribe una función llamada `suma_de_las_potencias` que, dada una lista $\{x_i\}$ calcule $\sum x_i^n$, siendo n otro valor que se le pasa como parámetro. Se deja este problema como ejercicio para el lector.

Como hemos dicho antes, una función también puede retornar una lista como resultado. Gracias a esto también podemos convertir en función el otro ejemplo antes visto, que a partir de una lista dada crea otra lista que contiene las raíces cuadradas de cada elemento de la lista original. La solución sería la siguiente:

```
1 import math    # Para poder usar la funcion sqrt
2
3 def raices_cuadradas(lista):
4     """Dada una lista como parametro, que contiene una serie de
5     numeros, la funcion retorna otra lista que contiene las
6     raices cuadradas de esos numeros"""
7     lista_resultado = []   # De momento esta vacia
8     for dato in lista:
9         lista_resultado.append(math.sqrt(dato))
10     # Al salir del bucle tenemos la lista_resultado rellena
11     return lista_resultado
12
13
14 # Para probar el ejemplo anterior, creare una lista
15 p = [4, 25, 16, 9]
```

```
16 resultado = raices_cuadradas(p)
17 print "La lista es:", p
18 print "Las raices cuadradas son:", resultado
```

Retornar una lista como resultado puede usarse para crear funciones que retornen varios resultados, aunque existen otros mecanismos en Python que no se verán en este curso. Por ejemplo, una función que dado un par de enteros retorna el cociente y el resto de su división. Podemos retornar una lista cuyo primer elemento sea el cociente, y el segundo sea el resto, como se muestra en el listado siguiente

```
1  def cociente_y_resto(a, b):
2      """Dados dos enteros, a y b, la funcion retorna una lista
3      cuyo primer elemento es el cociente entero a/b y cuyo segundo
4      elemento es el resto de dicha division"""
5      cociente=a/b
6      resto=a %b
7      return [cociente, resto]
8
9  # Probemos la funcion con un par de numeros que nos de el usuario
10 numero1=int(raw_input("Dame un entero: "))
11 numero2=int(raw_input("Dame otro: "))
12 resultado=cociente_y_resto(numero1, numero2)
13 print "El cociente es", resultado[0], "y el resto", resultado[1]
```

Un par de observaciones sobre el programa anterior. En primer lugar, la función es tan simple que su código podría haberse acortado hasta dejarla en una sola línea, simplemente si no hacemos uso de las variables intermedias `cociente` y `resto`, sino que ponemos directamente las expresiones `a/b` y `a %b` en los valores a retornar. Así (he omitido aquí la documentación de la función, que sería la misma que en el listado anterior):

```
1 def cociente_y_resto(a, b):
2     return [a/b, a %b]
```

En segundo lugar, cuando llamamos a la función y recogemos el valor retornado, lo hacemos sobre una variable llamada `resultado`, y después necesitamos acceder a `resultado[0]` para obtener el cociente y a `resultado[1]` para obtener el resto. Esto es poco legible y puede mejorarse gracias a una característica del lenguaje Python, que nos permite asignar las listas del mismo modo que se mostró en la para las tuplas. Si al lado izquierdo de una asignación, en lugar de una variable hay una lista de variables, y al lado derecho de la asignación hay una lista de valores o expresiones, entonces primero se evalúan las expresiones del lado de la derecha, y después cada una de las variables de la izquierda recibirá cada uno de los resultados. Esto es:

```
>>> [x, y, z] = [8, 3*5, 10/2]
>>> x
8
>>> y
15
>>> z
5
>>> [a, b, z] = [z, y, x]
```

¿Puedes adivinar qué valor tomarán las variables `a`, `b` y `z` tras esta última asignación?

Para que este tipo de asignación funcione, la lista de la izquierda ha de tener la misma longitud que la lista de la derecha, y la lista de la izquierda ha de estar compuesta exclusivamente por nombres de variables. La lista de la derecha podría ser cualquier cosa. ¡Incluso el resultado de llamar a una función! Esto nos permitiría hacer, por ejemplo:

```
1 [cociente, resto] = cociente_y_resto(10, 3)
2 print "Cociente=", cociente, "Resto=", resto
```

Es decir, hemos usado `[cociente, resto]` en vez de `resultado`, por lo que después tenemos nombres para el cociente y para el resto en lugar de tener que acceder a ellos con la sintaxis menos legible `resultado[0]`, `resultado[1]`.

La función puede cambiar elementos en la lista que recibe como parámetro

Una consecuencia directa de que las listas son objetos **mutables**, y del funcionamiento del paso de parámetros y la asignación en Python, es que, a diferencia de lo visto hasta ahora, una función puede recibir un parámetro de tipo lista y devolver la misma modificada.

Ya hemos visto en el tema de funciones que el paso de parámetros no es más que una asignación, en la que se asigna a los *parámetros formales* (los declarados en la implementación de la función) los valores que tienen los *parámetros reales* (los que se ponen en la llamada a la función).

Teniendo esta idea clara, veamos qué pasa cuando una función intenta modificar el valor del parámetro que ha recibido. Considera el siguiente ejemplo:

```
 1 def intenta_modificar(x):
 2     print "    x vale", x
 3     x = [0, 0, 0]
 4     # Hemos cambiado el valor de x?
 5     print "    Ahora x vale", x
 6 
 7 # Vamos a crear una variable a con una lista
 8 a = [1, 2, 3]
 9 print "a vale", a
10 
11 # Llamamos a la funcion intenta_modificar, pasandole a
12 intenta_modificar(a)
13 
14 # Habra cambiado el valor de a?
15 print "Ahora a vale", a
```

¿Qué crees que saldrá por pantalla al ejecutar este programa? Piénsalo un instante antes de leer la respuesta.

```
a vale [1, 2, 3]
    x vale [1, 2, 3]
    Ahora x vale [0, 0, 0]
Ahora a vale [1, 2, 3]
```

Como puedes ver, la función ha cambiado el valor de `x`, pero eso no ha afectado al valor de `a`. Si no comprendes por qué, los siguientes esquemas pueden ayudarte a clarificar la situación.

Instrucción	Explicación	Diagrama
`a = [1, 2, 3]`	Crea una nueva lista que contiene los datos 1, 2 y 3, y hace que la variable `a` "apunte" (se refiera) a dicha lista.	a → [1 | 2 | 3]
`print a`	Se imprime el valor de `a` y obviamente sale `[1, 2, 3]`	
`intenta_modificar(a)`	Durante la llamada a la función, se realiza la asignación al parámetro formal, es decir `x=a`, lo cual crea un nuevo "nombre" para la misma lista	x → [1 | 2 | 3] ← a
`print x`	Se imprime el valor de `x` que obviamente es el mismo que el de `a`.	

`x = [0, 0, 0]`	Se crea una nueva lista que contiene tres ceros, y se modifica x para que ahora apunte a esta nueva lista. Eso no afecta a a que sigue apuntando a la antigua.	x → [0, 0, 0]; a → [1, 2, 3]
`print x`	Se imprime el valor de x y ahora salen tres ceros	
(return implícito)	La función retorna, esto hace que x deje de existir. Ya que niguna otra variable apunta a la lista `[0,0,0]`, esta lista también será destruida por Python en algún momento, de forma automática.	[0, 0, 0]; a → [1, 2, 3]
`print a`	Se imprime el valor de a y como se ve del último diagrama, sigue apuntando a la lista original, por lo que de nuevo sale `[1, 2, 3]`	a → [1, 2, 3]

Cabe destacar que todo lo explicado en este ejemplo en el que a y x son variables de tipo `list`, es igualmente cierto y aplicable a cualquier otro tipo de datos en Python. En particular, esto mismo sucede si a y x son de tipo entero. En general, sea cual sea el tipo del parámetro de una función, una asignación que se haga a ese parámetro causa que se modifique sólo esa variable (haciendola apuntar a otro lugar) y sin afectar a la variable que era el parámetro real.

Sin embargo... ¿qué pasa si dentro de la función hacemos `x[0]=0`, por ejemplo? Es decir, ¿qué imprimiría el código siguiente?

```
1  def intenta_modificar2(x):
2      print "    x vale", x
3      x[0] = 0
4      # Hemos cambiado el valor de x?
5      print "    Ahora x vale", x
6
7  # Vamos a crear una variable a con una lista
8  a = [1, 2, 3]
9  print "a vale", a
10
11 # Llamamos a la funcion intenta_modificar, pasandole a
12 intenta_modificar2(a)
13
14 # Habra cambiado el valor de a?
15 print "Ahora a vale", a
```

Si te fijas, este ejemplo es idéntico al anterior, salvo por la línea 3, que antes hacía `x=[0,0,0]` y ahora hace `x[0]=0` (bueno, y también que hemos renombrado la función a `intenta_modificar2`, pero esto es irrelevante).

Si esperabas que al ejecutar este código la variable a no se viera afectada, es que no lo has pensado bien. Mira el resultado:

```
a vale [1, 2, 3]
    x vale [1, 2, 3]
    Ahora x vale [0, 2, 3]
Ahora a vale [0, 2, 3]
```

Al cambiar el primer elemento de `x` y poner allí un cero, ¡hemos afectado indirectamente a la variable `a` del programa principal, que ahora también tiene un cero en su primer elemento! En realidad, si has comprendido el caso del ejemplo anterior, este tiene también perfecto sentido. La siguiente tabla lo explica paso a paso:

Instrucción	**Explicación**	**Diagrama**
`a = [1, 2, 3]`	Crea una nueva lista que contiene los datos 1, 2 y 3, y hace que la variable `a` "apunte" (se refiera) a dicha lista.	a → 1 2 3
`print a`	Se imprime el valor de `a` y obviamente sale `[1, 2, 3]`	
`intenta_modificar2(a)`	Durante la llamada a la función, se realiza la asignación al parámetro formal, es decir `x=a`, lo cual crea un nuevo "nombre" para la misma lista	x, a → 1 2 3
`print x`	Se imprime el valor de `x` que obviamente es el mismo que el de `a`.	
`x[0] = 0`	Se modifica el primer elemento de la lista apuntada por `x`. Ya que es la misma lista a la que apuntaba `a`, ¡estamos modificando `a`!	x, a → 0 2 3
`print x`	Se imprime el valor de `x` y ahora sale `[0,2,3]`	
(return implícito)	La función retorna, esto hace que `x` deje de existir. La lista a la que apuntaba `x` no se destruye, porque todavía tenemos a `a` apuntando a ella.	a → 0 2 3
`print a`	Se imprime el valor de `a` y evidentemente sale `[0, 2, 3]`	a → 0 2 3

Por tanto cuando se pasa una lista a una función, esta función puede modificar los elementos almacenados en esa lista, si la función usa los corchetes para acceder a un elemento de la lista y le asigna otro valor.

Esto es posible sólo con listas (en general, con cualquier objeto en Python que sea *mutable*). Si `x` fuese un entero, no hay forma de hacer `x[0]=0` (si lo intentas dará un error, porque el operador `[]` sólo tiene sentido sobre listas). Y si lo que hacemos es `x=0` ya hemos visto que eso simplemente cambia la `x` para que "apunte" a un cero, sin modificar el valor al que apuntaba `a`.

De modo que es posible escribir funciones que modifiquen una lista "in situ", es decir, sin crear una lista nueva a partir de ella. Así, podríamos hacer una función `raices_cuadradas_lista` que en lugar de crear una nueva lista con los valores de las raices cuadradas y retornar esa nueva lista, lo que haga sea ir modificando la lista original sustituyendo cada elemento por su raíz cuadrada. Sería como sigue:

```
1 import math    # Para poder usar la funcion sqrt
```

```
2
3 def raices_cuadradas_in_situ(lista):
4     """Dada una lista como parametro, que contiene una serie de
5     numeros, la funcion MODIFICA esa lista cambiando cada elemento
6     por su raiz cuadrada"""
7     for i in range(len(lista)):
8         lista[i] = math.sqrt(lista[i])
9     # No es necesario retornar ningun valor, ya estan todos en la lista
10
11 # Para probar el ejemplo anterior, creare una lista
12 p = [4, 25, 16, 9]
13 print "Antes de llamar, la lista es:", p
14 resultado = raices_cuadradas_in_situ(p)
15 print "Despues de llamar, la lista es:", p
16 print "El valor retornado por la funcion es", resultado
```

Observa que la función no contiene sentencia `return` (aunque de todas formas hará un `return` implícito cuando salga del bucle, ya que no hay más instrucciones para ejecutar en la función). La forma de llamar a esta función por tanto es distinta a la del ejemplo con `raices_cuadradas`, donde hacíamos `resultado=raices_cuadradas(p)`. En aquel ejemplo la lista con las raíces cuadradas era diferente de la lista `p`, y la variable `resultado` servía para apuntar a la nueva lista. En cambio ahora la lista con las raíces es la propia `p` (sus valores originales se pierden), y la función no retorna nada, por lo que asignar a `resultado` como hace el programa anterior no tiene sentido. Bastaba llamar a la función sin asignar su resultado a ninguna variable. No obstante, si lo asignamos como se ve en el ejemplo anterior, encontraremos que la variable `resultado` toma el valor especial `None`, que es el valor que Python usa para representar "nada". La función no ha retornado nada.

Al ejecutar ese código veremos en pantalla:

```
Antes de llamar, la lista es: [4, 25, 16, 9]
Despues de llamar, la lista es: [2.0, 5.0, 4.0, 3.0]
El valor retornado por la funcion es None
```

Para terminar, indicar que es posible escribir programas que *nunca hagan uso de esta característica*. Es decir, cualquier problema computacional se puede resolver haciendo uso de funciones que nunca modifiquen los valores de los parámetros que reciben, sino que en vez de ello creen otros datos nuevos y los retornen.

2.6.7. Listas y cadenas

La cadena es un tipo especial de lista

Una cadena de caracteres es un tipo especial de lista, en la que cada elemento es un carácter, esto es, una letra, un dígito, un signo de puntuación o un carácter de control. Por tanto, podemos usar los mecanismos que conocemos de las listas para iterar sobre sus elementos, calcular su longitud, concatenarlas, etc.

Por ejemplo, el siguiente código cuenta cuántas letras vocales hay en la frase escrita por el usuario:

```
1 # Pedir al usuario un texto
2 texto = raw_input("Escribe una frase: ")
3
4 vocales = 0 # Contador de vocales halladas
5 # Recorrer letra a letra el texto
6 for letra in texto:
7   # Comprobar si es una vocal
8   # Es sencillo gracias al operador "in" ya visto para listas
9   # que tambien es aplicable a cadenas
10  if letra in "aeiouAEIOU":
11      vocales = vocales + 1
```

```
12
13 print "Tu frase tiene", len(texto), "letras"
14 print "de las cuales", vocales, "son vocales."
```

Sin embargo hay una diferencia fundamental entre las cadenas y las listas. Las cadenas son *inmutables*, lo que significa que no podemos alterar ninguno de sus caracteres. Si bien podemos consultar el valor de `texto[i]` para ver qué letra es, no podemos en cambio asignar nada a la variable `texto[i]`. Si lo intentamos, obtendremos un error en tiempo de ejecución.

Imagina que queremos programar una función que recibe como parámetro una cadena y queremos que se ocupe de cambiar todas las vocales que aparezcan en dicha cadena por el signo _. Es decir, si la cadena que recibe contiene el mensaje `"Hola Mundo"`, el resultado debería ser `"H_l_ m_nd_"`. Si intentamos una versión de la función que modifique la cadena "in situ", fracasará:

```
1  def cambiar_vocales(texto):
2      """Esta funcion recibe una cadena de texto como parametro
3      e intenta cambiar todas las vocales que contiene por el
4      signo _. Sin embargo, ya que las cadenas son inmutables,
5      producira un error."""
6      # Recorremos todos los indices para acceder a cada letra
7      for i in range(len(texto)):
8          # Miramos si es una vocal
9          if texto[i] in "aeiouAEIOU":
10             # Si lo es, la cambiamos. Esto es lo que dara error
11             texto[i] = "_"
12
13 # Para probar la funcion anterior creo un mensaje
14 mensaje = "Mensaje de prueba"
15
16 # Intento "borrar" sus vocales
17 cambiar_vocales(mensaje)
18
19 # Imprimo el resultado (en realidad esto nunca llegara a ejecutarse
20 # porque el programa "rompe" antes con un error)
21 print mensaje
```

Al ejecutar el programa anterior obtendremos un error:

```
Traceback (most recent call last):
  File "funcion-sustituir-vocales-mal.py", line 17, in <module>
    cambiar_vocales(mensaje)
  File "funcion-sustituir-vocales-mal.py", line 11, in cambiar_vocales
    texto[i] = "_"
TypeError: 'str' object does not support item assignment
```

La solución consiste en hacer una función que, en lugar de modificar la cadena, crea una cadena nueva que va construyendo letra a letra, copiando cada letra de la cadena original, salvo si es una vocal en cuyo caso inserta el carácter `"_"`. Pero aparentemente esto tampoco puede hacerse, ya que al ser las cadenas datos inmutables, carecen también del método `.append()` para poder ir añadiendo caracteres. ¿Entonces?

La respuesta es que, aunque efectivamente no puedo añadir caracteres a una cadena dada, sí puedo crear una nueva cadena como la suma de dos (concatenación), y asignar el resultado de esta concatenación a la misma variable. Por ejemplo:

```
>>> texto="Prueba"
>>> texto
'Prueba'
>>> texto = texto + "s"
>>> texto
'Pruebas'
```

Puede parecer que hemos añadido una letra "s" a la cadena. En realidad lo que ha ocurrido es que hemos creado una cadena nueva, que contiene "Pruebas", pero sin afectar a la cadena original que sigue conteniendo "Prueba". Después hacemos que la variable texto se refiera a esta nueva cadena. La cadena original ya no tiene variable que se refiera a ella y Python la destruirá.

Usando este enfoque, la función que cambia vocales por subrayados quedaría así:

```
def cambiar_vocales(texto):
    """Esta funcion recibe una cadena de texto como parametro
    y retorna otra cadena que es una copia de la recibida, salvo
    por que todas las vocales han sido sustituidas por el
    signo _"""
    resultado = ""   # De momento no tenemos letras en el resultado
    # Recorremos todos los indices para acceder a cada letra
    for letra in texto:
        # Miramos si es una vocal
        if letra in "aeiouAEIOU":
            # Si lo es, concatenamos "_" al resultado
            resultado = resultado + "_"
        else:
            # Si no, concatenamos la letra en cuestion
            resultado = resultado + letra
    # Una vez salimos del bucle, tenemos la cadena formada
    # No olvidarse de retornarla!
    return resultado

# Para probar la funcion anterior creo un mensaje
mensaje = "Mensaje de prueba"

# Borramos sus vocales, pero debo recoger el resultado en otra variable
cambiado = cambiar_vocales(mensaje)

# Imprimir cadena original y cambiada:
print mensaje
print cambiado
```

Convertir una cadena en una lista

A menudo se tiene una cadena de texto que contiene una serie de palabras, separadas mediante algún símbolo separador especial. Por ejemplo, al exportar una hoja de cálculo Excel en el formato denominado csv (*comma separated values*) lo que hace Excel es escribir un fichero en el que cada línea representa una fila de la tabla, y los contenidos de cada celda se escriben todos en la misma línea, usando el punto y coma (;) como separador de columnas.

Por ejemplo, considera una tabla Excel que contenga la siguiente información, sobre los apellidos más frecuentes:

Apellido	**Frecuencia**	**Por 1000**
GARCIA	1.483.939	31,6
GONZALEZ	935.135	19,9
RODRIGUEZ	932.924	19,8
FERNANDEZ	928.618	19,7
LOPEZ	879.145	18,7

Si exportamos esta hoja al formato csv, el fichero resultante se podría abrir en el bloc de notas (contiene solo texto) y veríamos lo siguiente:

```
Apellido;Frecuencia;Por 1000
GARCIA;1.483.939;31,6
GONZALEZ;935.135;19,9
RODRIGUEZ;932.924;19,8
```

```
FERNANDEZ;928.618;19,7
LOPEZ;879.145;18,7
```

Si leyéramos ese fichero desde un programa en Python, la función `readline` nos daría una línea completa, y sería interesante poder "romper" esa línea en tres trozos y así poder recuperar el apellido, la frecuencia y el tanto por 1000 almacenado en cada línea.

Pues bien, las cadenas de texto tienen entre sus métodos uno llamado `split()` que sirve justamente para este propósito. Este método espera un parámetro que es el carácter (o en realidad la sub-cadena) que se usará como separador. En nuestro ejemplo se trataría del punto y coma. El resultado del método es una lista, en la cual cada elemento es una cadena que ha resultado de "romper" la cadena original. En nuestro ejemplo, la lista tendría tres elementos correspondientes a los tres trozos. Por ejemplo:

```
>>> linea = "GARCIA;1.483.939;31,6"
>>> trozos=linea.split(";")
>>> len(trozos)
3
>>> trozos[0]
'GARCIA'
>>> trozos[1]
'1.483.939'
>>> trozos[2]
'31,6'
```

Otra forma de convertir una cadena en una lista es usar la función `list()` pasándole una cadena. El resultado será una lista en la que cada elemento es una letra de la cadena original. Esto tiene en general poca utilidad, y no lo usaremos en esta asignatura.

Convertir una lista en una cadena

Podemos convertir una lista que contenga valores de cualquier tipo en una cadena, que sería la misma que veríamos en pantalla al hacer un `print` de la lista. Para ello basta pasar dicha lista a la función `str()`. Pero esto en general no es muy útil:

```
>>> lista = [1, 2.3, 5]
>>> lista
[1, 2.2999999999999998, 5]
>>> cadena = str(lista)
>>> cadena
'[1, 2.2999999999999998, 5]'
```

Otro caso mucho más útil es aquel en el que tenemos una lista cuyos elementos son cadenas, y queremos "concatenar todos juntos" estos elementos, posiblemente insertando entre ellos algún tipo de separador.

Por ejemplo, tenemos la lista `semana` que contiene como elementos `["lunes", "martes", "miercoles", "jueves", "viernes", "sabado", "domingo"]` y queremos construir una cadena que contenga el texto `"lunes, martes, miercoles, jueves, viernes, sabado, domingo"`.Como ves, se trata de concatenar las palabras que había en la lista, poniendo una coma y un espacio (`", "`) entre ellas.

Con lo que sabemos hasta ahora, esto podría hacerse de diferentes formas. La más directa (y la peor), podría ser simplemente "sumar" (concatenar) los 7 elementos de la lista "a mano".

```
1 cadena = semana[0] + ", " + semana[1] + ", " + semana[2] + ", " +
2          semana[3] + ", " + semana[4] + ", " + semana[5] + ", " +
3          semana[6]
```

Ni que decir tiene que el código anterior es una barbaridad. Además de lo torpe que resulta tener que repetir tantas veces la misma expresión, ni siquiera es una solución genérica. Habría que cambiarlo si la lista que quiero concatenar tiene más o menos elementos. ¡Para esto están los bucles!

En una solución con bucles, comenzaríamos con una `cadena` vacía (que contenga `""`) e iríamos sumándole a dicha `cadena` cada elemento de `semana`, más la cadena `", "`, hasta llegar al último.

```
cadena=""    # Inicialmente vacía
for texto in semana:
    cadena = cadena + texto + ", "
```

Sin embargo el código anterior no hace exactamente lo que queríamos, ya que aparecería una coma también después del último día de la semana, y en ese caso no la queremos. El ultimo caso es excepcional y para manejar ese caso tenemos que "afear" un poco la solución anterior. Tenemos dos formas, la primera sería comprobar dentro del bucle si estamos en el último elemento, y si no estamos, añadir la coma separadora. Sería así:

```
cadena=""    # Inicialmente vacía
ultimo_indice = len(semana) - 1
for i in range(len(semana)):
    cadena = cadena + semana[i]    # Añadir el dato
    if i != ultimo_indice:         # y si no es el ultimo
        cadena = cadena + ", "     # añadir también la coma
```

La segunda forma es hacer que el bucle se recorra para todos los elementos salvo el último, y tratar este después, fuera del bucle. Sería así:

```
cadena=""    # Inicialmente vacía
ultimo_indice = len(semana) - 1
for i in range(ultimo_indice):
    cadena = cadena + semana[i] + ", "   # Añadir dato con su coma
# Ahora añadir el ultimo elemento
cadena = cadena + semana[ultimo_indice]  # sin coma detras
```

Cualquiera de estas soluciones es buena. Ambas funcionan para listas de cualquier longitud, salvo para listas vacías, en las que la segunda solución fallaría. Si quisiéramos hacerlo totalmente genérico habría que comprobar en el segundo caso si la lista tiene longitud cero, en cuyo caso no habría que hacer nada más sobre `cadena`.

Este es el problema que intentamos resolver, y aunque ya hemos dado con una solución para él, resulta conveniente saber que Python ya venía con una solución pre-programada para este problema, entre los métodos disponibles para cadenas se tiene `.join()`.

La sintaxis de este método es un poco extraña y de alguna forma parece estar "al revés" de lo que uno esperaría. El método se invoca sobre una cadena que contenga el separador a insertar (en nuestro caso, la cadena `", "`), y se le pasa como parámetro la lista que contiene los datos a concatenar. El resultado de la llamada es la cadena concatenada como queríamos. Para el ejemplo anterior esto se traduciría en sustituir la línea 6 por la siguiente:

```
cadena = ", ".join(semana)
```

En casos excepcionales podemos querer concatenar todos los elementos de la lista sin ningún tipo de separador intermedio. Por ejemplo, en el caso en que cada elemento de la lista sea un solo carácter y queramos juntarlos todos para formar una palabra o cadena. Esto también puede hacerlo `join()`, basta usar una cadena vacía (`""`) en la llamada. Por ejemplo:

```
>>> letras = ["H", "o", "l", "a"]
>>> palabra = "".join(letras)
>>> palabra
'Hola'
```

2.6.8. Un caso frecuente en ingeniería: listas de listas rectangulares, matrices

Muchas magnitudes usadas en ingeniería se representan como matrices porque se corresponden con una estructura bidimensional sobre una rejilla con un cierto número de filas y columnas. Por

ejemplo, medidas tomadas sobre un terreno, tensiones e intensidades en un circuito eléctrico, esfuerzos en la barras de una estructura, colores de pixels de una imagen, etc. Python dispone de un tipo específico de dato para realizar cálculos con matrices, con la mayor parte de las operaciones habituales (producto, inversa, determinante), implementadas. El módulo en donde estas características están implementadas se estudiará más adelante. Sin embargo es posible realizar cálculos con matrices representando estas como listas de listas. El inconveniente estriba en que no existe una función predefinida en Python que permita crear esta estructura de datos, por así decirlo, darle un número de filas y columnas. Sin embargo es muy fácil con lo que hemos visto hasta ahora, de hecho se puede hacer de dos formas, usando `append` y el operador de repetición `*` o bien sólo este último.

La primera idea consiste en ir añadiendo "filas" a una lista inicialmente vacía. Nótese que de esta forma cada elemento de la lista inicialmente vacía es a su vez una lista, de modo que para indexar cada uno de los escalares son necesarios dos índices, el primero indicaría la fila y el segundo el elemento concreto dentro de esa fila, es decir, la columna. A modo de ejemplo se creará una lista de listas para albergar una matriz de tres filas y dos columnas.

```
>>> a=[]
>>> for i in range(3):
...     a.append([None]*2)
...
```

En el caso anterior no se han inicializado los elementos individuales de la matriz con ningún valor, se ha utilizado `None`. Según el caso podría ser interesante inicializarlos todos a 1 o a 0. Por ejemplo:

```
>>> a=[]
>>> for i in range(3):
...     a.append([0]*2)
...
```

En cualquier caso, es posible asignar los valores individuales utilizando los índices correspondienes a la fila y columna de cada elemento:

```
>>> a[0][0]=3.1
>>> a[0][1]=-1.0
>>> a[1][0]=0.0
>>> a[1][1]=-0.1
>>> a[2][0]=7.0
>>> a[2][1]=5.3
```

Así, la lista de listas quedaría finalmente:

```
>>> a
[[3.1, -1.0], [0.0, -0.1], [7.0, 5.3]]
```

Si se utiliza un objeto de este tipo con uno o más índices fuera del rango definido para filas o columnas, se produce un error:

```
>>> a[1][3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

La segunda de las formas de crear una lista de listas para almacenar una matriz es similar a la anterior, pero la lista inicial se crea del tamaño de las filas (con tantos elementos como filas tenga la matriz) y después se asigna a cada uno de esos elementos una lista con tantos elementos como columnas tenga la matriz. Para ello se usa el operador de repetición `*`. La clave de que así se consiga asignar espacio para los distintos elementos está en que (como se dijo en su momento) los operadores sobre listas producen *nuevos* objetos, no referencias a objetos existentes. Al igual que en

el caso anterior, se puede inicializar cada uno de los elementos de la matriz en este punto. En este caso se ha inicializado a 1.0, aprovechando para recordar la importancia del tipo de datos empleado a la hora de realizar operaciones aritméticas y que Python es un lenguaje de tipado dinámico.

```
>>> a=[None]*3
>>> for i in range(3):
...     a[i]=[1.0]*2
...
```

El resto de operaciones serían idénticas al caso anterior.

Es evidente que la salida por pantalla de las matrices así representadas no tiene el aspecto que se espera, una organización bidimensional de los números que contiene. Podemos aproximarnos de forma sucesiva a este aspecto recordando distintos temas que se han visto en el tema de listas. En primer lugar vamos a iterar con un sólo bucle directamente sobre la lista, es decir, cada elemento que vamos a obtener es a su vez una lista:

```
>>> for fila in a:
...     print fila
...
[3.1, -1.0]
[0.0, -0.1]
[7.0, 5.3]
```

Las instrucciones anteriores hacen que, la primera vez, el objeto `fila` sea la primera fila de la matriz, una lista, que al ser mostrada aparece entre corchetes con los elementos separados por comas. La segunda vez la siguiente y así sucesivamente. Ahora bien, se puede hacer esto mismo con cada elemento de cada fila, usando un bucle `for` anidado más:

```
>>> for fila in a:
...     for elemento in fila:
...         print elemento,
...     print
...
3.1 -1.0
0.0 -0.1
7.0 5.3
```

En el código de ejemplo anterior, el primer bucle asigna al objeto `fila` cada una de las filas de `a`, el segundo asigna a `elemento` cada uno de las componentes de la fila actual. Un par de comentarios sobre como se usa `print`. El primero de ellos está dentro del bucle más interno y por lo tanto muestra cada uno de los elementos individuales de la matriz. Como queremos que salgan en la misma línea, lleva una coma al final. El segundo `print` es para mostrar en distintas líneas de la pantalla cada una de las filas, está dentro del primer `for` pero fuera del segundo.

Alternativamente, se pueden usar secuencias de índices dentro del rango válido para filas y columnas para acceder a los elementos de la matriz. En este caso el código que la mostraría por la pantalla sería:

```
>>> for i in range(3):
...     for j in range(2):
...         print a[i][j],
...     print
...
```

Como se explicó anteriormente, `range(3)` produce la lista `[0,1,2]` y `range(2)` produce la lista `[0,1]`, por este motivo, con los dos bucles anidados anteriores, se generan todas las posibles parejas de índices válidos (en realidad el producto cartesiano de estos) para `a`.

El acceso a los elementos usando índices también permite cambiar el valor de los elementos de la matriz, cosa que no se podría hacer iterando sobre filas y elementos de filas. Por ejemplo, el siguiente fragmento de código asigna cero a los elementos de la matriz anterior:

```
>>> for i in range(3):
...     for j in range(2):
...         a[i][j]=0
...
```

De los ejemplos anteriores podría deducirse que es necesario usar de forma explícita el número de filas o columnas para poder recorrer una lista que represente una matriz. Como se puede deducir fácilmente, estos valores se pueden obtener de forma casi trivial usando la función `len`. De esta forma, por ejemplo, se pueden reescribir algunos de los fragmentos de código anteriores como sigue:

```
>>> for i in range(len(a)):
...     for j in range(len(a[i])):
...         print a[i][j],
...     print
...
```

La longitud de la lista "mas externa", `len(a)`, es el número de filas. La longitud de cada fila `len(a[i])` es el número de columnas. En el caso concreto que nos ocupa, matrices rectangulares, se podría haber usado `len(a[0])` ya que todas las filas tienen la misma longitud. La alternativa usada sirve para todos los casos, incluso aquellos en los que cada fila tenga un tamaño distinto y nos permite abstraernos de las dimensiones de la matriz que estemos manejando. Esto es más recomendable que usar de forma explícita el número de filas y columnas.

Como es evidente, las matrices representadas como listas se manejan con funciones de la misma forma que cualquier otra lista. Así por ejemplo podríamos convertir en función el código usado para crear una matriz.

```
1  def zeros(filas,columnas):
2      """Devuelve una matriz de ceros de alto filas y ancho columnas"""
3      #Lista vacia
4      a = []
5      #Tantas veces como filas
6      for i in range(filas):
7          #anadir una fila con tantas columnas como ancho
8          a.append([0]*columnas)
9          #retornar la matriz
10     return a
```

Esto nos permite reutilizar código en nuestros programas, como se indicó cuando se trató el tema de funciones. Por ejemplo, usando la función anterior se puede escribir otra para pedir una matriz de reales por el teclado. Primero se crea la matriz, después se recorre y en cada iteración se pide el elemento indexado por i y j.

```
1  def pide_matriz_float(filas,columnas):
2      """Pide una matriz de reales por el teclado"""
3      #se crea la matriz
4      a=zeros(filas,columnas)
5      #se recorre, en cada iteracion se pide uno de los elementos
6      for i in range(filas):
7          for j in range(columnas):
8              #se muestran los indices de los elementos que se piden
9              print "[",i,"][",j,"]=",
10             a[i][j]=float(raw_input())
11     return a
```

Y podríamos hacer lo mismo con el fragmento de código que muestra una matriz por la pantalla.

```
1  def muestraMatriz(a):
2      """Muestra una matriz, representada la lista de listas a, por la
3      pantalla, una fila por linea de la pantalla"""
4      #se recorre la lista de listas usando indices y dos for anidados
```

```
5       for i in range(len(a)):
6           for j in range(len(a[i])):
7               #la coma hace que los elementos de una fila vayan seguidos
8               print a[i][j],
9           #siguiente linea
10          print
```

Al final de esta sección se incluyen una serie de ejercicios resueltos en donde se ha intentado cubrir un abanico lo más amplio posible casos de uso de matrices representadas como listas de listas. Siempre se han usado funciones, dado que es una práctica aconsejable proceder de esta forma.

2.6.9. Ejercicios resueltos.

[**Ejercicio 1**] Escribir una función que reciba como parámetros una lista y un valor escalar. La función devuelve el número de veces que se repite el valor escalar dentro de la lista.

```
1  #En esta solucion se itera sobre la lista
2  #de valores directamente
3  #elemento toma el valor de cada uno de los
4  #elementos de la lista.
5  def veces1(lista,valor):
6      """Devuelve el numero de veces que se repite
7      valor dentro de lista"""
8      #se inicializa el contador
9      veces=0
10     for elemento in lista:
11         #si el elemento actual es el valor que se contabiliza
12         if elemento==valor:
13             #se incrementa el contador
14             veces=veces+1
15     #se retorna el valor al acabar el bucle
16     return veces
17
18 #En esta solucion se itera sobre la lista
19 #de los indices validos para el parametro
20 #lista para acceder al valor de cada elemento de
21 #la lista se usan los [] y el valor del
22 #indice.
23 def veces2(lista,valor):
24     """Devuelve el numero de veces que se repite
25     valor dentro de lista"""
26     veces=0
27     for i in range(len(lista)):
28         if lista[i]==valor:
29             veces=veces+1
30     return veces
```

[**Ejercicio 2**] Escribir una función que reciba como parámetros una lista y un valor escalar. La función devuelve el índice que ocupa la primera ocurrencia del valor escalar dentro de la lista o `None` si no se encuentra.

```
1  #En este caso es mas conveniente usar una lista
2  #de indices validos porque se necesita saber en
3  #donde esta el valor buscado.
4  def donde_esta(lista,valor):
5      """Devuelve el indice de lista en donde esta la primera ocurrencia de
6      valor, si es que esta, None en caso contrario"""
7      #ya que python dispone del operador in, nos permite devolver
8      #None directamente si valor no esta en lista, sin tener que
9      #recorrerla de forma explicita
10     #si el valor esta en la lista
11     if valor in lista:
12         #recorremos la lista
```

```
13         for i in range(len(lista)):
14             #si se encuentra el valor
15             if lista[i]==valor:
16                 #se retorna su indice
17                 return i
18     #en caso contrario se retorna None
19     else:
20         return None
```

[Ejercicio 3] Escribir una función que reciba como parámetros dos listas, representando dos vectores numéricos. La función devuelve el producto escalar de ambos vectores.

```
1 #En este caso no se puede iterar sobre las
2 #listas (con lo que se ha visto en teorÃa)
3 #porque es necesario recorrer en
4 #paralelo las dos, que codifican dos vectores.
5 def producto_escalar(v1,v2):
6     """Devuelve el producto escalar de v1 y v2"""
7     #como se va a hacer un sumatorio se inicializa resultado a 0
8     resultado=0
9     #se supone que v1 y v2 tienen la misma longitud
10     for i in range(len(v1)):
11         #se acumula en resultado el producto de las  componentes
12         #que ocupan la misma posicion
13         resultado=resultado+v1[i]*v2[i]
14     return resultado
```

[Ejercicio 4] Escribir una función que reciba como parámetros dos listas, representando dos vectores numéricos. La función devuelve la suma vectorial de ambos vectores.

```
1 #En este caso no se puede iterar sobre las
2 #listas (con lo que se ha visto en teorÃa)
3 #porque es necesario recorrer en
4 #paralelo las dos, que codifican dos vectores.
5 #Se da tamano a resultado repitiendo 0
6 #tantas veces como elementos hay en v1.
7 def suma_vectorial1(v1,v2):
8     """Devuelve la suma vectorial de v1 y v2"""
9     #dar tamano al raesultado
10     resultado=[0]*len(v1)
11     #da igual usar len de v1, v2 o resultado
12     for i in range(len(resultado)):
13         #la componente i esima del resultado es la suma de las
14         #componentes que ocupan el mismo lugar
15         resultado[i]=v1[i]+v2[i]
16     return resultado
17
18 #En este caso no se puede iterar sobre las
19 #listas (con lo que se ha visto en teorÃa)
20 #porque es necesario recorrer en
21 #paralelo las dos, que codifican dos vectores.
22 #En esta version no se da tamano al resultado
23 #se utiliza append para ir anadiendo elementos
24 #segun se van calculando
25 def suma_vectorial2(v1,v2):
26     #no se da tamano porque se usa append en el for
27     resultado=[]
28     for i in range(len(v1)):
29         #anadir la componente i-esima al resultado
30         resultado.append(v1[i]+v2[i])
31     return resultado
```

[**Ejercicio 5**] Escribir una función que reciba como parámetros una lista de números y un número. La función devolverá el producto del número y del vector que está representado como lista de números.

```
1 #similar a los anteriores, con una lista
2 #y un escalar como parametros.
3 def producto_por_escalar(v,a):
4     """Devuelve el producto del vector v por el escalar a"""
5     resultado=[0]*len(v)
6     #en cada iteracion se almacena en la componente i esima del resultado
7     #el producto de la componente i esima por el escalar
8     for i in range(len(resultado)):
9         resultado[i]=v[i]*a
10    return resultado
```

[**Ejercicio 6**] Escribir una función que reciba como parámetro una lista representando un vector numérico y que devuelva su módulo.

```
1 def modulo(a):
2     """Devuelve el modulo del vector a, representado como lista"""
3     #la variable que se utiliza para hacer el sumatorio se inicializa
4     #a cero
5     suma_cuadrados=0
6     for i in range(len(a)):
7         #se acumula el cuadrado de la componente actual
8         suma_cuadrados=suma_cuadrados+a[i]**2
9     #se puede discrepar de poner esto aqui
10    import math
11    #se devuelve la raiz cuadrada de la suma de los cuadrados
12    return math.sqrt(suma_cuadrados)
```

[**Ejercicio 7**] Escribir una función que reciba una lista y que devuelva otra lista con copias únicas de los elementos de la lista original, es decir, eliminando las repeticiones.

```
1 def copiasUnicas(a):
2     """Devuelve un vector con copias unicas de los elementos de a,
3     es ddecir, elimina las repeticiones"""
4     #vector de booleanos para marcar los repetidos, inicialmente False
5     esta_repetido=[False]*len(a)
6     #comparar cada elemento con todos los que siguen
7     for i in range(len(a)-1):
8         for j in range(i+1,len(a)):
9             #si coincide el valor
10            if a[i]==a[j]:
11                #esta repetido
12                esta_repetido[j]=True
13    #inicializacion del resultado
14    b=[]
15    #recorrer otra vez a
16    for i in range(len(a)):
17        #si el elemento no esta repetido
18        if not esta_repetido[i]:
19            #anadir al resultado
20            b.append(a[i])
21    #devolver
22    return b
```

[**Ejercicio 8**] Escribir funciones para calcular la media, varianza y moda de un vector de datos numéricos. Comprobar el resultado usando unos valores de prueba adecuados.

```
1 def indice_maximo(lista):
2     """Devuelve el indice en donde esta el maximo de una lista de
3     numeros"""
```

```
4      #se inicializa el indice a cero, el primero
5      i_max=0
6      #se recorren los restantes
7      for i in range(1,len(lista)):
8          #si el i esimo es mayor, se actualiza
9          if lista[i]>lista[i_max]:
10             i_max=i
11     #se retorna al posicion en donde esta el maximo
12     return i_max
13
14 def media(lista):
15     """Calcula la media de una lista de numeros, como real."""
16     #suma dividido entre el numero de elementos. Se convierte a float
17     #la suma para obtener decimales.
18     return float(sum(lista))/len(lista)
19
20 def varianza1(x):
21     """Calcula la varianza de una lista de numeros, como real. Se usa
22     la expresion Var(x)=E(x**2)-(E(x))**2."""
23     #media de la lista
24     m=media(x)
25     #inicializacion de la lista de x**2
26     #alternativamente se podria ir anadiendo con append
27     #dentro del bucle
28     x_2=[0]*len(x)
29     #para cada elemento de la lista
30     #se calcula el elemento i esimo de la lista de x**2
31     for i in range(0,len(x)):
32         x_2[i]=x[i]**2
33     #se devuelve el valor de la varianza
34     return media(x_2)-media(x)**2
35
36 def varianza2(x):
37     """Calcula la varianza de una lista de numeros, como real. Se usa
38     la expresion Var(x)=E((x-E(x))**2). Por lo tanto primero se calcula
39     E(x), despues (x-E(x))**2 y despues la media de esta."""
40     #media de los valores de la lista
41     m=media(x)
42     #inicializacion de la lista con (x-media(x))**2
43     x_e_x_2=[0]*len(x)
44     #se calcula el elemento i esimo de la lista con (x-media(x))**2
45     for i in range(0,len(x)):
46         x_e_x_2[i]=(x[i]-m)**2
47     #se retorna la varianza
48     return media(x_e_x_2)
49
50
51 #La implementacion de la funcion moda no es especialmente ortodoxa,
52 #he primado que sea pedagogica, y tampoco es eficiente.
53 #El ultimo elemento, si es distinto no se cuenta, lo cual da igual:
54 #Si la lista es de longitud uno, ese elemento es la moda.
55 #Si la longitud es mayor que uno, o todos son distintos y entonces
56 #cualquiera es la moda o este no se repite y entonces no es la moda
57 #o se ha repetido antes y ya se ha contado. Se puede mejorar
58 #facilmente anadiendo un vector de booleanos marcando los visitados.
59 def moda(lista_numeros):
60     """Devuelve la moda de una lista_numeros, es decir, el numero
61     que mas se repite.
62     NOTA: no valido para muestras de valores continuos."""
63     #caso especial, un solo elemento
64     if len(lista_numeros)==1:
```

```
65          return lista_numeros[0]
66      #inicializacion lista de contadores
67      lista_contadores=[0]*len(lista_numeros)
68      #para cada elemento excepto el ultimo
69      for i in range(0,len(lista_numeros)-1):
70          lista_contadores[i]=1
71          #para todos los siguientes
72          for j in range(i+1,len(lista_numeros)):
73              #si son iguales se incrementa el contador asociado a i
74              if lista_numeros[i]==lista_numeros[j]:
75                  lista_contadores[i]=lista_contadores[i]+1
76      #alternativamente se puede usar sorted y devolver el
77      #primer valor de la lista ordenada
78      return lista_numeros[indice_maximo(lista_contadores)]
79
80  print moda([2,3,4,3,2,1,2,1,2,3,4,5])
81  print varianza1([2,3,4,3,2,1,2,1,2,3,4,5])
82  print varianza2([2,3,4,3,2,1,2,1,2,3,4,5])
```

[**Ejercicio 9**] Escribir una función que devuelva `True` si el número que se pasa como parámetro es capicúa, `False` es caso contrario. Usando esa función escribir un programa que muestre los n primeros números capicúa. El número de capicúas se pide por el teclado.

```
1  def es_capicua(n):
2      """devuelve true si es capicua, false en caso contrario"""
3      #si el numero convertido en cadena es igual que el numero
4      #con las cifras en orden inverso, usando str
5      return str(n)==str(n)[::-1]
6
7  #cuantos capicua se van a mostrar
8  cuantos=int(raw_input('cuantos?'))
9  #contador capicuas
10 conta=0
11 #cada uno de los numeros que se prueban
12 num=0
13 #mientras no encuentre el numero de capicuas pedido
14 while conta < cuantos:
15     #si es capicua
16     if es_capicua(num):
17         #se muestra
18         print num
19         #se cuenta
20         conta+=1
21     #siguiente numero
22     num+=1
```

[**Ejercicio 10**] Escribir una función que devuelva `True` si el número que se pasa como parámetro es automórfico, `False` en caso contrario. Un número es automórfico si aparece al final de su cuadrado. Por ejemplo, 25 al cuadrado es 625, luego 25 es automórfico. Usando esa función escribir un programa que muestre los n primeros números capicúa. El número de automórficos se pide por el teclado.

```
1  #Usa cadenas y slicing.
2  def es_automorfico(n):
3      """devuelve True si es automorfico, False en caso contrario"""
4      #numero al cuadrado convertido a cadena
5      cuadrado_str=str(n**2)
6      #longitud del numero original
7      lon_n=len(str(n))
8      #longitud del cuadrado
9      lon_cuadrado=len(cuadrado_str)
10     #parte final del cuadrado de la longitud del numero original
```

```
    p_final_cuadrado=cuadrado_str[lon_cuadrado-lon_n:lon_cuadrado]
    #si coincide con el numero original es automorfico
    return p_final_cuadrado==str(n)

#cuantos automorficos se van a mostrar
cuantos=int(raw_input('cuantos?'))
#contador automorficos
conta=0
#cada uno de los numeros que se prueban
num=0
#mientras no encuentre el numero de automorficos pedido
while conta < cuantos:
    #si es automorfico
    if es_automorfico(num):
        #se muestra
        print num
        #se cuenta
        conta+=1
    #siguiente numero
    num+=1
```

[**Ejercicio 11**] Escribir una función que reciba tres parámetros. Los dos primeros representan el número de filas y columnas de una matriz. El tercero el valor inicial de los elementos de la matriz. La función devuelve una lista de listas que representa esa matriz, cada elemento de la matriz igual al tercer parámetro de la función.

```
def inicializaMatriz(fil,col,val):
    """Inicializa una matriz al numero de filas y columnas indicado,
    cada elemento de la matriz a val"""
    #numero de filas
    a = [None]*fil
    #cada filal longitud igual a columnas, cada elemento igual a val
    for i in range(fil):
        a[i]=[val]*col
    return a
```

[**Ejercicio 12**] Escribir una función que reciba una matriz de números como parámetro, representada como una lista de listas. La función devolverá la suma de todos los valores de la matriz.

```
#version clasica recorriendo la matriz elemento a elemento
def suma_elementos_matriz(a):
    """Calcula la suma de todos los elementos de una matriz"""
    #variable para hacer el sumatorio
    suma=0
    #para el numero de filas de a
    for i in range(len(a)):
        #para el numero de columnas de a
        #si a es rectangular, da igual usar len(a[i]) o len(a[0])
        for j in range(len(a[i])):
            suma=suma+a[i][j]
    return suma

#version usando sum para calcular una lista con la suma de las filas
#despues se usa sum para sumar esa lista.
def suma_elementos_matriz_sum(a):
    """Calcula la suma de todos los elementos de una matriz"""
    #lista de suma de cada fila
    suma_f=[]
    #para cada fila
    for i in range(len(a)):
        #anadir su suma a la lista
        suma_f.append(sum(a[i]))
```

```
24      #retornar la suma de la lista de sumas por filas
25      return sum(suma_f)
```

[**Ejercicio 13**] Escribir una función que reciba dos matrices y devuelva la suma matricial de ambas.

```
1  def suma_matricial(a,b):
2      """Devuelve la suma de las matrices a y b"""
3      #se crea la matriz de alguna de las formas vistas
4      #el tamano del resultado es igual que el de los parametros de entrada
5      c=crea_matriz(len(a),len(a[0]))
6      #cada elemento i,j  del resultado es la suma de los elementos i,j
7      #de a y b
8      for i in range(len(a)):
9          for j in range(len(a[0])):
10             c[i][j]=a[i][j]+b[i][j]
11     return c
```

[**Ejercicio 14**] Escribir una función que reciba una matriz y devuelva una lista con la suma de cada fila por separado.

```
1  def suma_filas_matriz(a):
2      """Devuelve la suma de los elementos de cada fila de a por separado"""
3      #alternativamente se podria crear vacia y anadir con append, cuidado
4      #con la variable usada entonces, reinicializar a cero cada vez
5      suma_filas=[0]*len(a)
6      #para cada fila
7      for i in range(len(a)):
8          #sumar cada elemento
9          for j in range(len(a[0])):
10             suma_filas[i]=suma_filas[i]+a[i][j]
11     return suma_filas
12
13 #version usando la funcion sum por filas
14 def suma_filas_matrizSum(a):
15     """Devuelve la suma de los elementos de cada fila de a por separado"""
16     #en realidad en este caso no hace falta inicializar a cero
17     suma_filas=[0]*len(a)
18     #para cada fila
19     for i in range(len(a)):
20         #se suman los elementos de la fila con sum
21         suma_filas[i]=sum(a[i])
22     return suma_filas
```

[**Ejercicio 15**] Escribir una función que reciba una matriz y devuelva una lista con la suma de cada columna por separado.

```
1  #lo mismo por columnas. No se puede hacer usando sum, a no ser
2  #que se trasponga antes.
3  def suma_columnas_matriz(a):
4      """Devuelve una lista con la suma de cada columna de a por separado"""
5     suma_columnas=[0]*len(a[0])
6     #basicamente, para cada columna
7     for j in range(len(a[0])):
8         #recorrerla y sumar sus elementos
9         for i in range(len(a)):
10            suma_columnas[j]=suma_columnas[j]+a[i][j]
11    return suma_columnas
```

[**Ejercicio 16**] Escribir una función que reciba una matriz y que devuelva el mayor de sus elementos.

```
1  #max devuelve el valor de una lista pero no el de una lista de listas
2  #por eso tiene sentido implementar esta funcion
```

```
3 #version clasica recorriendo la matriz.
4 def max_matriz(a):
5     """Devuelve el mayor elemento de una matriz representada como lista
6     de listas"""
7     #tambien se podria haber inicializado al maximo de la primera fila y
8     #comenzar i en 1
9     maximo=a[0][0]
10    #se recorre la matriz
11    for i in range(len(a)):
12        for j in range(len(a[i])):
13            #si se encuentra uno mayor que el maximo actual, es el
14            #nuevo maximo
15            if a[i][j]>maximo:
16                maximo=a[i][j]
17    return maximo
18
19 #max devuelve el valor de una lista pero no el de una lista de listas
20 #por eso tiene sentido implementar esta funcion
21 #version calculando el maximo de los maximos por filas.
22 def max_matriz_max(a):
23     """Devuelve el mayor elemento de una matriz representada como lista
24     de listas"""
25     #Inicializo maximo al maximo de la primera fila
26     maximo=max(a[0])
27     #se recorren las filas posteriores a la 0
28     for i in range(1,len(a)):
29         #maximo de la fila actual
30         max_fila=max(a[i])
31         #si es mayor que el maximo actual, es el nuevo maximo
32         if max_fila>maximo:
33                 maximo=max_fila
34     return maximo
```

[**Ejercicio 17**] Escribir una función que reciba una matriz y que devuelva su traspuesta.

```
1 #evidentemente si no es rectangular no tiene sentido.
2 def traspuesta(a):
3     """Devuelve la traspuesta de a"""
4     #el resultado tiene de numero de filas las columnas de a y de numero
5     #de columnas las filas de a
6     b=zeros(len(a[0]),len(a))
7     #para cada elemento de a
8     for i in range(len(a)):
9         for j in range(len(a[i])):
10            #se asigna al elemento j,i de b el i,j de a
11            b[j][i]=a[i][j]
12    return b
```

[**Ejercicio 18**] Escribir una función que reciba dos matrices y devuelva su producto matricial.

```
1 def producto_matricial(a,b):
2     """Devuelve el producto matricial de a y b"""
3     #en este caso, como se va a sumar repetidas veces sobre cada c[i][j]
4     #es obligatorio inicializar c toda a ceros
5     #el tamano del resultado es filas de a y columnas de b
6     c=zeros(len(a),len(b[0]))
7     #para cada elemento del resultado
8     for i in range(len(a)):
9         for j in range(len(b[0])):
10            #acumular en c[i][j] el producto de a[i][k] y b[k][j]
11            for k in range(len(a[0])):
12                c[i][j]=c[i][j]+a[i][k]*b[k][j]
13    return c
```

2.6.10. Ejercicios Propuestos

[**Ejercicio 1**] Escribir una función que reciba como parámetros una lista y un valor escalar. La función devuelve el índice que ocupa la *última* repetición del valor escalar dentro de la lista o `None` si no se encuentra.

[**Ejercicio 2**] Escribir una función que reciba como parámetros una lista y un valor escalar. La función devuelve una lista con los índices de cada repetición del valor escalar dentro de la lista o `None` si no se encuentra.

[**Ejercicio 3**] Repetir el ejercicio que expurga las repeticiones de los elementos de una lista, usando el método `append` y el operador `in`.

[**Ejercicio 4**] Escribir una función que reciba un parámetro, representando el tamaño de una matriz cuadrada. La función devuelve una lista de listas que contiene la matriz identidad.

[**Ejercicio 5**] Escribir una función que reciba un parámetro, representando una matriz cuadrada. La función devuelve `True` si la matriz es diagonal, `False` en caso contrario.

[**Ejercicio 6**] Escribir una función que reciba un parámetro, representando una matriz cuadrada. La función devuelve `True` si la matriz es simétrica, `False` en caso contrario.

[**Ejercicio 7**] Usando `append`, Escribir una función que reciba una matriz y devuelva una lista con la suma de cada columna por separado.

[**Ejercicio 8**] Escribir una función que reciba una lista de listas de números representando una matriz y que devuelva una lista con el máximo de cada fila por separado

[**Ejercicio 9**] Escribir una función que reciba una lista de listas de números representando una matriz y que devuelva una lista con el máximo de cada columna por separado

[**Ejercicio 10**] Repetir el anterior usando la traspuesta del parámetro y la función que calcula el máximo de cada fila por separado.

2.7. Almacenamiento permanente

Los datos utilizados por los programas que hemos hecho hasta el momento no se conservan indefinidamente, sino únicamente durante el tiempo de ejecución. Cuando termina la ejecución del intérprete que lee y ejecuta nuestro programa esos datos dejan de existir desde un punto de vista práctico así que, si necesitamos usarlos más tarde, tendremos que volver a calcularlos de nuevo. Sin embargo, existe la posibilidad de conservar esos datos de forma permanente, incluso tras apagar el ordenador. Para ello los programas pueden hacer uso de **ficheros** que permiten guardar los datos fuera de la memoria del ordenador, típicamente en un dispositivo como un disco duro, un lápiz de memoria, etc. De esta manera un programa puede salvar los datos que se deban conservar entre ejecuciones, e incluso puede hacer uso de los datos guardados por otros programas; el uso de ficheros es una de las formas más obvias y sencillas de intercambio de información entre programas.

Los datos pueden almacenarse en ficheros utilizando su representación interna (en binario), aunque en estos apuntes nos limitaremos a presentar el manejo de ficheros de texto, es decir, aquellos que contienen la información en forma de cadenas de caracteres y que, por tanto, pueden leerse perfectamente en cualquier editor simple de textos, como Textedit, gEdit, o el bloc de notas.

2.7.1. Apertura y cierre de ficheros

Podemos pensar en los ficheros como si fuesen cuadernos de notas. Así, para poder usar un cuaderno primero lo debemos *abrir*, a continuación podemos *leer* y/o *escribir* en él y, cuando hayamos terminado, lo debemos *cerrar*.

De igual forma, para que un programa pueda acceder al contenido de un fichero, éste debe ser previamente abierto. Para ello se debe utilizar la función `open()`, a la que hay que suministrar dos parámetros, que serán dos cadenas de caracteres indicando:

- El *nombre* del fichero al que queremos acceder
- El *modo* en el que vamos a abrir el fichero. Esta cadena puede tener los siguientes valores

- 'r' para abrir un fichero con permiso de lectura. Si el fichero no existe se produce un error.
- 'w' para abrir un fichero con permiso de escritura. Si el fichero no existe se crea uno vacío y si ya existe se reescribe su contenido.
- 'a' para abrir un fichero con permiso de escritura, de forma que lo que escriba el programa se añade al contenido previo del fichero.

La función `open()` puede fallar en ocasiones, por ejemplo, si intentamos abrir en modo lectura un fichero que no exista obtendremos un mensaje de error y el programa detendrá su ejecución, como se puede ver en el ejemplo siguiente:

```
>>> f = open('informes.txt','r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'informes.txt'
```

En este caso el intérprete muestra un error puesto que el fichero que pretendíamos abrir para leer no existe. Para poder presentar las operaciones de lectura

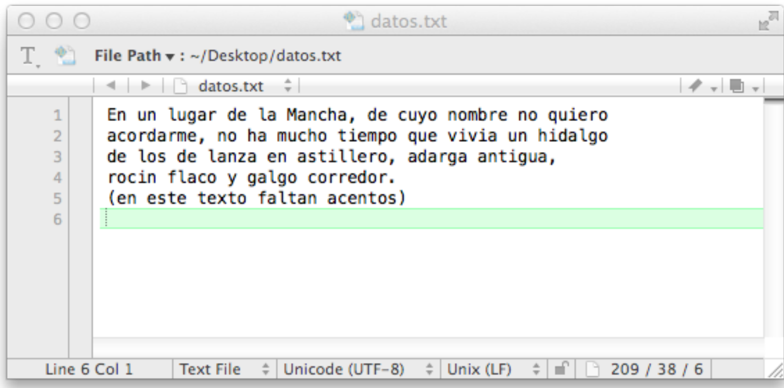
Lo que retorna la llamada a la función `open()` debe asociarse a una variable que, en caso de que la operación de apertura tenga éxito, hace referencia al fichero recién abierto. Con esa variable operaremos para acceder al contenido del fichero, tal como veremos más adelante. Una vez hayamos concluido nuestras operaciones de acceso al contenido debemos cerrar el fichero con `close()`.

IMPORTANTE

Es muy importante que una vez que cerremos los ficheros una vez que hayamos terminado de acceder a su contenido, de igual forma que cerraríamos un cuaderno de notas tras haberlo utilizado. De lo contrario, podrían producirse pérdidas de información.

2.7.2. Lectura de líneas a lista

Supongamos que hemos creado con algún editor simple un fichero con el nombre `'datos.txt'`, que contiene el texto siguiente[5]:



Con el código que se muestra a continuación abrimos el fichero en modo lectura, leemos su contenido, cerramos el fichero y, finalmente, mostramos en la consola el texto leído, que ahora ya está cargado en memoria en la *lista* `datos`:

[5] Para evitar posibles problemas con la codificación hemos prescindido del uso de cualquier carácter que no pertenezca a la tabla de códigos ASCII, como por ejemplo las letras acentuadas.

```
1 f = open('datos.txt', 'r')
2 datos = f.readlines()
3 f.close()
4 print 'Contenido del fichero:'
5 for lin in datos:
6     print lin
```

La ejecución del programa produce esta salida en la consola:

```
Contenido del fichero:
En un lugar de la Mancha, de cuyo nombre no quiero

acordarme, no ha mucho tiempo que vivia un hidalgo

de los de lanza en astillero, adarga antigua,

rocin flaco y galgo corredor.

(en este texto faltan acentos)

>>>
```

Con `readlines()` se leen todas las líneas del fichero y se guardan en la lista `datos`. Cada elemento de `datos` es una cadena de caracteres que se corresponde con cada línea del fichero de texto, incluyendo el carácter de salto de línea final ('\n'), si existe. Por esa razón al imprimir en consola la cadena `lin` aparece a continuación del texto una línea en blanco, debida a la impresión del carácter de salto de línea. Si desde el intérprete imprimimos el valor de la lista `datos` podemos comprobar que, efectivamente, cada línea de texto incluye el carácter especial '\n' al final.

```
>>> print datos
['En un lugar de la Mancha, de cuyo nombre no quiero\n',
'acordarme, no ha mucho tiempo que vivia un hidalgo\n',
'de los de lanza en astillero, adarga antigua,\n',
'rocin flaco y galgo corredor.\n',
'(en este texto faltan acentos)\n']
```

El bucle `for` para leer ficheros por líneas

Una forma más compacta de leer un fichero línea a línea es mediante la iteración sobre variable del fichero con un bucle `for`. El código que se muestra a continuación define una función que, dado un nombre de fichero, cuenta y retorna el número de líneas en blanco que contiene:

```
1 def cuenta_lineas_blanco(nombre_fichero):
2     contador = 0
3     f = open(nombre_fichero, 'r')
4     for lin in f:
5         if len(lin.strip()) == 0:
6             contador = contador + 1
7     return contador
8
9 print 'Lineas en blanco:', cuenta_lineas_blanco('datos.txt')
```

El bucle `for` asigna en cada iteración una línea del fichero a la variable `lin`. La porción de código de las líneas 4 a 6 pueden interpretarse como:

> *"para cada línea del fichero, si su longitud es cero, entonces aumenta en una unidad el contador, ya que se trata de una línea en blanco".*

Como curiosidad cabe destacar que esta función contará como líneas en blanco aquellas cuyos únicos caracteres sean espacios, aunque realmente su longitud no sea 0. Esta funcionalidad se consigue utilizando `strip()` sobre la variable `lin` (línea 5), ya que así eliminamos los espacios iniciales

y finales, así como el salto de línea de la cadena. Una vez eliminados, si la longitud es 0 significa que no había ningún otro carácter y, efectivamente, era una línea en blanco que debe ser contabilizada.

Lectura en bloque de todos los caracteres

Otra forma de leer fácilmente un fichero de texto consiste en utilizar una llamada a `read()`, de forma que se obtiene una única cadena con todo el contenido del fichero, tal como se muestra en el siguiente ejemplo:

```
1 f = open('datos.txt', 'r')
2 datos = f.read()
3 f.close()
```

En el ejemplo esa cadena se asocia a la variable `datos` que, como se puede ver al mostrar su contenido, tiene saltos de línea tras las palabras ‘quiero’, ‘hidalgo’, ‘antigua,’, ’corredor.’ y ‘acentos)’.

```
>>> datos
'En un lugar de la Mancha, de cuyo nombre no quiero\nacordarme, no ha mucho
tiempo que vivia un hidalgo\nde los de lanza en astillero, adarga antigua,\nrocin
flaco y galgo corredor.\n(en este texto faltan acentos)\n'
>>>
```

Si se imprime esta cadena con `print` el salto de linea se hace efectivo en la consola y podemos ver las líneas que contiene el fichero.

```
>>> print datos
En un lugar de la Mancha, de cuyo nombre no quiero
acordarme, no ha mucho tiempo que vivia un hidalgo
de los de lanza en astillero, adarga antigua,
rocin flaco y galgo corredor.
(en este texto faltan acentos)

>>>
```

ATENCIÓN

La posibilidad de cargar todo un fichero en una sola operación de lectura debe utilizarse con precaución ya que, si éste es de gran tamaño, se podría sobrepasar la capacidad de almacenamiento en memoria y el programa no funcionaría.

2.7.3. Escritura en ficheros

Para escribir texto en un fichero necesitamos, en primer lugar, abrir el fichero en *modo escritura*. Si el fichero no existe, se crea, y si ya existe entonces se sobrescribe (y, por tanto, se pierde) su contenido. A continuación podemos efectuar operaciones de escritura de cadenas de caracteres con `write()`, que lleva como parámetro la cadena de caracteres que se desea escribir. Nótese que la operación de escritura no añade separador de líneas, así que si queremos escribir una línea completa debemos incluir al final de la cadena el salto de línea de forma explícita, tal como se muestra en este ejemplo:

```
>>> f = open('texto.txt','w')
>>> f.write('Esta es la primera\n')
>>> f.write('Esta es la segunda\n')
>>> f.close()
```

Dado que los mecanismos de manejo de ficheros en Python están orientados a ficheros de cadenas de caracteres, si necesitamos almacenar valores de otro tipo debemos convertirlos previamente a cadenas. Por ejemplo, el programa siguiente pide por teclado números enteros positivos hasta que se escriba un valor negativo, y almacena en el fichero `'pares.txt'` los que son pares y en otro denominado `'impares.txt'` los que son impares:

```
def es_par(n):
    return n % 2 == 0


def pide_numero():
    print 'Introduce un número positivo (un negativo para terminar):',
    n = raw_input()
    return int(n)

# Programa principal
fpares = open('pares.txt', 'w')
fimpares = open('impares.txt', 'w')
n = pide_numero()
while (n >= 0):
    numero_y_salto = str(n)+'\n'
    if es_par(n):
        fpares.write(numero_y_salto)
    else:
        fimpares.write(numero_y_salto)
    n = pide_numero()
fpares.close()
fimpares.close()
```

En la linea 8, dentro de la función `pide_numero()`, se realiza la conversión a entero de la cadena tecleada por el usuario. Esta conversión es necesaria para luego poder comprobar si el número es par o impar. Posteriormente, a la hora de escribir el número en el fichero correspondiente, tenemos que convertirlo de nuevo en una cadena de caracteres y añadirle el salto de línea. Esa conversión se realiza en la línea 15, asignando el resultado a la variable `numero_y_salto`.

Si ahora queremos hacer un programa que lea los dos ficheros creados por el programa anterior y que calcule el valor medio de los números pares y de los impares entonces podríamos programar lo siguiente:

```
def calcula_media(nombre):
    suma = 0
    contador = 0
    fichero = open(nombre, 'r')
    lineas = fichero.readlines()
    fichero.close()
    for lin in lineas:
        n = int(lin)
        suma = suma + n
        contador = contador + 1
    return float(suma)/contador

# Programa principal
print 'La media de los pares es', calcula_media('pares.txt')
print 'La media de los impares es', calcula_media('impares.txt')
```

Observa que tenemos que hacer la conversión a número entero de cada uno de los números en formato cadena leídos del fichero, tal como se hace en la línea 8. Este programa hace además otra conversión necesaria para obtener un resultado correcto, que es la de convertir la variable `suma` a número real, de forma que la división de la línea 11 devuelva un número real y no trunque el resultado a un número entero. La ejecución del programa anterior produciría una salida como la siguiente, que dependerá, obviamente, del contenido de los ficheros:

```
La media de los pares es 18.0
La media de los impares es 44.5
```

2.7.4. Ejercicios resueltos

[**Ejercicio 1**] Haz una función que tome como parámetros un nombre de fichero y una palabra y retorne una lista con los números de línea en los que aparece dicha palabra. Haz un programa en el que se utilice dicha función.

```
# Programa: Imprimir en consola todas las línea de un fichero que contengan
# una determinada secuencia de caracteres

def caracteres_en_fichero(nombre, caracteres):
    f = open(nombre, 'r')
    lista_de_numeros = []
    contador = 0
    for linea in f:
        contador = contador + 1
        if caracteres in linea:
            lista_de_numeros.append(contador)
    f.close()
    return lista_de_numeros


# Programa principal
nombre_fichero = raw_input('Introduce el nombre del fichero: ')
secuencia = raw_input('Introduce la secuencia de caracteres a buscar: ')
lineas = caracteres_en_fichero(nombre_fichero, secuencia)
if len(lineas) == 0:
    print 'La secuencia', secuencia, 'no aparece en ninguna línea'
else:
    print 'La secuencia', secuencia, 'aparece en las lineas siguientes:'
    print lineas
```

[**Ejercicio 2**] Haz una función `encripta_fichero()` que cifre un fichero grabando el resultado en otro fichero. Los parámetros de la función serán el nombre del fichero origen y del fichero destino. El algoritmo de cifrado será muy simple: cada carácter se sustituirá por el siguiente en la tabla de caracteres ASCII. Para ello necesitarás hacer uso de las funciones `ord()`, que retornan la posición en la tabla de un carácter que se pasa como argumento, y `chr()`, que retorna el carácter de que ocupa la posición que se le pasa como argumento.

```
def encripta_fichero(origen, destino):
    # Cargar fichero original
    f1 = open(origen, 'r')
    texto_original = f1.read()
    f1.close()
    # Crear texto encriptado
    texto_encriptado = ''
    for c in texto_original:
        texto_encriptado = texto_encriptado + chr(ord(c)+1)
    # Guardar texto encriptado en fichero destino
    f2 = open(destino, 'w')
    f2.write(texto_encriptado)
    f2.close()
```

[**Ejercicio 3**] Haz una función `desencripta_fichero()` capaz de descifrar los ficheros encriptados generados por la función del ejercicio anterior.

```
def desencripta_fichero(origen, destino):
    # Cargar fichero encriptado
    f1 = open(origen, 'r')
    texto_encriptado = f1.read()
    f1.close()
```

```
6      # Crear texto desencriptado
7      texto_desencriptado = ''
8      for c in texto_encriptado:
9          texto_desencriptado = texto_desencriptado + chr(ord(c)-1)
10     # Guardar texto desencriptado en fichero destino
11     f2 = open(destino, 'w')
12     f2.write(texto_desencriptado)
13     f2.close()
```

Parte III

Introducción a las bases de datos

CAPÍTULO 3

Introducción a las Bases de Datos

3.1. Conceptos de bases de datos

3.1.1. Definición de Bases de Datos (BD) y de Sistema de Gestión de Bases de Datos (SGBD).

Una **base de datos** se puede definir como un conjunto de información interrelacionada, que se almacena de forma estructurada para ser utilizada posteriormente. Actualmente, cuando hablamos de base de datos nos referimos a aquellas que se almacenan de forma digital, y cuya información se procesa mediante programas específicos. Una base de datos contiene información asociada a un determinado sistema u organización, ya sea de una empresa de transportes, una compañía aérea, una farmacia, la secretaría de la Universidad o de nuestra pequeña biblioteca.

Originalmente esta información se almacenaba en diferentes ficheros (fichero de productos, de empleados, de pedidos, etc.) que eran accedidos por diversos programas (gestión de pedidos, nóminas, contabilidad, etc.).

Esta organización de ficheros y programas independientes para manejarlos, pronto empezó a presentar diversos problemas: se debían organizar muy bien los ficheros y coordinar muy bien los programas para compartir los mismos formatos, para que no se almacenara información redundante, para permitir accesos concurrentes, para establecer permisos a los distintos usuarios, etc.

Un **sistema de gestión de base de datos (SGBD)** es un programa que nos permite manipular la información que conforma una base de datos y que aporta soluciones a los problemas ya mencionados.

Por tanto no es lo mismo una base de datos (BD) que un gestor de bases de datos (SGBD), aunque en muchas ocasiones nos referimos a ambos con el mismo término: Base de Datos. Es decir, la BD del vídeo club es la información asociada al mismo (películas, juegos, clientes, alquileres, etc.) mientras que el SGBD es el programa que utilizamos para mantener dicha información (Postgres, Oracle, SQL Server, MySQl, Access u otros).

3.1.2. Funcionalidades de un SGBD

Hemos comentado que un **gestor de base de datos (SGBD)** es un programa, aunque realmente de trata de un conjunto de programas interrelacionados que nos permiten gestionar diversas bases de datos.

Realmente se puede considerar como un Sistema Operativo de propósito específico, ya que sus funcionalidades son similares.

Cuando hablamos de un SGBD hablamos de un sistema que nos ofrece las siguientes funcionalidades:

- **Gestión de almacenamiento**: Es una de las más importantes funcionalidades de un SGBD,

ya que su principal objetivo es el almacenamiento de información así como la recuperación de la misma de forma eficiente. Los SGBD cuentan con estructuras y mecanismos para poder realizar la manipulación de datos (inserción, modificación y borrado) y su localización de manera muy optimizada. Además de lo que es la organización de la información en disco, los SGBD también disponen de mecanismos para gestionar la información que se mantiene en memoria principal.

- **Gestión de usuarios**: Ya que una BD puede ser accedida por múltiples usuarios se hace necesario llevar una gestión de los mismos, pudiendo asignar diferentes privilegios a cada uno.
- **Gestión de integridad**: Otro aspecto fundamental es mantener una consistencia e integridad entre los datos almacenados. Los SGBD habilitan procedimientos para imponer restricciones a la información de la BD.
- **Gestión de concurrencia**: Como consecuencia del acceso simultáneo a la información de una BD por distintos usuario o programas, los SGBD implementan sistemas de control de la concurrencia de manera que, en la medida de lo posible, cada usuario tenga la sensación de que es el único que está accediendo a la BD en un momento dado.
- **Gestión de transacciones**: Existen muchas operaciones que si se ejecutan *a medias* pueden provocar inconsistencia entre los datos. Uno de los ejemplos más ilustrativos es la transferencia de x euros de una cuenta A a otra cuenta B, este tipo de operaciones se componen de varias *sub-operaciones*: comprobar que A tiene saldo, descontar a A x euros y sumar a B x euros. Este tipo de operaciones se denomina **transacciones**. El SGBD debe garantizar que: o se realizan todas las *sub-operaciones* o no se realiza ninguna. Un sistema así, se denomina *sistema transaccional*.
- **Gestión de recuperaciones**: Los ordenadores, de vez en cuando, *caen* o *se cuelgan* (por muy diversos motivos), y aún así los SGBD deben garantizar la consistencia de los datos. Para ello disponen de mecanismos de recuperación, de tal forma que ante una caída el sistema se recupere en la versión consistente más reciente posible. Estos mecanismos están íntimamente ligados a la Gestión de transacciones.

Todas estas funcionalidades se basan en distintos gestores o módulos que no son independientes sino que trabajan de forma muy interrelacionada.

Por encima de todos estos gestores, que en la mayoría de los casos resultan transparentes para los usuario del SGBD, disponemos de un intérprete que nos permitirá comunicarnos con el propio SGBD utilizando un determinado lenguaje. El lenguaje más común es el SQL. Mediante órdenes SQL podremos crear nuestra BD, insertar, modificar, borrar y consultar datos, definir usuarios, privilegios, restricciones sobre los datos, y muchas cosas más. La mayoría de los SGBD también disponen de diferentes lenguajes de programación con los que podremos trabajar con nuestros datos.

En muchos casos no se hace necesario un lenguaje para manejar nuestro SGBD, ya que disponemos de interfaces gráficas que nos facilitan las operaciones y resultan de gran utilidad cuando el usuario es inexperto. Como es lógico estas interfaces gráficas también presentan importantes deficiencias, *no se puede tener todo*.

Aunque a día de hoy hay muchos tipos de SGBD funcionando (algunos de ellos realmente antiguos), los más estandarizados son los SGBD Relacionales (SGBDR). Estos se caracterizan porque los datos se almacenan en tablas en cuyas filas iremos guardando la información de nuestro sistema. A lo largo de este capítulo, cuando mencionemos un SGBD nos referiremos siempre a los Relacionales, ya que queda fuera de este contexto estudiar otros tipos de gestores.

3.1.3. Aplicaciones sobre Bases de Datos.

Cuando queremos trabajar con una base de datos lo normal es desarrollar una aplicación que se comunica con el SGBD. Las dos arquitecturas más utilizadas son:

Aplicaciones de Escritorio: se trata de programas implementados en un determinado lenguaje de programación (C++, Java, python, etc.) que hace de *host o anfitrión* y que se comunica, utilizando alguna librería de programación, con el SGBD. Este tipo de arquitectura se denomina de dos niveles. Con el lenguaje anfitrión (en él se *aloja* el código para comunicarse con el SGBD) desarrollamos el interface (normalmente gráfico) que utiliza el usuario para manipular la información de la base de datos.

Aplicaciones Web: aquí hablamos de arquitectura de tres niveles: navegador web, servidor web y servidor de bases de datos (SGBD). En este caso es el servidor web el que mediante algún lenguaje de programación (es muy frecuente el uso de PHP) se comunica con el SGBD para atender las peticiones que hace el usuario a través de su navegador. Las siglas LAMP se asocian a este tipo de arquitectura y se corresponden con: **L**inux, sistema operativo en el que se instala el servidor web **A**pache que se comunica con el servidor de bases de datos **M**ySql utilizando el lenguaje **P**HP. Es frecuente que el MySql y el Apache corran sobre el mismo Linux.

3.2. Un ejemplo sencillo

A lo largo de este capítulo vamos a utilizar un ejemplo muy sencillo para ir ilustrando los diferentes apartados. El problema a resolver será crear una base de datos que contenga información sobre los pedidos de nuestra empresa. Los datos y restricciones que tenemos que contemplar son los siguientes:

1. Cada pedido tiene un número, y se almacenará el cliente que lo solicita y la fecha en la que se realiza
2. De cada producto se quiere registrar un código de producto, su descripción, su precio y sus existencias
3. Para los clientes se quiere registrar su nombre, su DNI y su dirección
4. Un pedido incluirá una serie de productos cada cual con su cantidad

3.3. Diseño de Bases de Datos

El diseño de una base datos consiste en definir cómo vamos a almacenar los datos. En nuestro caso, que vamos a utilizar SGBD Relacionales, el objetivo es definir el conjunto de tablas que contendrá toda la información del sistema. Cada tabla estará formada por una serie de columnas, cada una de ellas asociada a un tipo de datos -por ejemplo: identificador de cliente, nombre, DNI, Fecha de nacimiento, etc.- y una serie de filas, cada una de ellas almacenará la información correspondiente, por ejemplo: Pepa, 12345678Z, 1994/05/28, etc.

El proceso de diseño de una base de datos puede llegar a ser muy complejo, y se pueden obtener diferentes soluciones. El diseño es la primera fase en la vida de una base de datos y posiblemente la más importante. Un mal diseño tiene consecuencias nefastas en la implementación y explotación de una base de datos.

Uno de los principios fundamentales que rigen el diseño de una base de datos es evitar la redundancia o repetición de información: cada dato solo debe guardarse en un sitio. No se trata de un problema de ahorro de espacio, sino de un problema de consistencia de la información. Si

almacenamos una misma información en dos sitios distintos, siempre que se modifique en uno de ellos también debe ser modificada en el otro. De no ser así nunca sabríamos cuál es la verdadera información. El mantenimiento de información redundante en bases de datos, aunque no sean muy grandes, se convierte en un serio problema.

Siguiendo nuestro ejemplo, el diseño de una base de datos para almacenar los pedidos, consistirá en definir qué tablas y qué columnas tienen cada una de ellas para poder almacenar nuestra información -pedidos, clientes, productos- de forma lógica y cumpliendo el principio de no repetición de información.

Para ayudarnos en este proceso de diseño se utilizan los modelos de datos que comentamos a continuación.

3.4. Modelos de Datos

Ya que el objetivo de una base de datos es reflejar la información de un determinado sistema, se hace necesario compartir o establecer unas herramientas comunes que nos permitan trabajar con datos.

Se define un **Modelo de Datos** como una colección de herramientas que nos permiten describir los datos con los que vamos a trabajar. Con un modelo de datos podremos representar, en algunos casos de forma gráfica:

- Los propios datos. Por ejemplo un producto, un cliente, un pedido.
- Las relaciones entre ellos. Como que un pedido lo hace un cliente o que en un pedido aparecen una serie de productos.
- Las restricciones de los datos. Podemos establecer que un pedido solo lo realiza un cliente aunque un cliente puede realizar varios pedidos. O que un producto sólo puede aparece en un mismo pedido solo una vez.

Dependiendo del nivel de abstracción con el que trabajemos podemos utilizar distintos Modelos de Datos. En una primera aproximación a nuestra solución del problema intentaremos diferenciar los distintos objetos o entidades que conforman nuestro sistema -el cliente, el pedido, el producto- así como los valores que queremos almacenar de cada uno -por ejemplo del cliente, su nombre, su dni y su dirección-. Además de identificar objetos también definiremos las relaciones que hay entre ellos. A este nivel de profundidad nos abstraemos de pequeños detalles, y el modelo de datos que se utiliza es el denominado Entidad-Relación (E-R). Este modelo nos permite expresar la información de nuestro sistema de forma gráfica, como veremos más adelante. En muchos casos el modelo E-R se hace con lápiz y papel.

Según avanzamos en la solución del problema nos acercamos cada vez más a la implementación de nuestra base de datos en el ordenador. Ahora nos centraremos en cuestiones como el almacenamiento de nuestras entidades y sus relaciones. A este nivel se suele trabajar ya con tablas -también llamadas relaciones-, y el modelo que utilizaremos será el Modelo Relacional.

Existen otros modelos de datos, que nos permiten profundizar aún más en aspectos concretos de nuestra implementación, pero aquí solo veremos los dos anteriormente mencionados.

3.5. Modelo Entidad-Relación

3.5.1. Introducción

El Modelo E-R es un modelo de datos orientado a objetos. En este modelo los objetos se denominan Entidades, de ahí su nombre. Como ya se dijo: el objetivo es identificar los objetos de nuestro sistema y ver cómo se relacionan. No se pretende aquí explicar en profundidad este modelo, pero sí es importante resaltar que el diseño de bases de datos se hace utilizando este modelo. Los usuarios de BD con poca formación diseñan tablas directamente en el modelo relacional, pero cuando el

Figura 3.1: Diagrama Entidad-Relación del sistema de Pedidos

sistema es medianamente complejo los profesionales no lo hacen así, usan este modelo en alguna de sus distintas versiones. El modelo original fue desarrollado por Chen y nos permite crear diagramas que representan la información de nuestro sistema. El aspecto más interesante de este modelo es que nos permite trabajar a un nivel de abstracción alto a la hora de dibujar nuestro diagrama E-R.

Vamos a crear el diagrama E-R asociado al problema de los pedidos. Como ya dijimos lo primero es identificar las entidades y los valores -atributos- que las caracterizan. Podemos identificar las siguientes entidades con sus respectivos atributos:

- Producto: con su código, su nombre, su precio y sus existencias.
- Cliente: con su nombre, su DNI y su dirección.
- Pedido: con su número y la fecha

Aunque a estas alturas aún no hemos hablado de identificadores, ya lo haremos más adelante, necesitamos poder tener algún valor que nos permita identificar de forma única a cada entidad. En el caso de los clientes podríamos pensar que el DNI cumple esa función, y es cierto, pero lo normal es *inventarnos* un valor numérico ara identificarles. Por tanto en el caso de los clientes vamos a añadir el atributo `IdCliente`; para los pedidos podemos utilizar el número de pedido, que renombraremos como IdPedido; por último, para los productos utilizaremos el mismo código aunque no sea numérico y a su vez le denominaremos `IdProducto` Con esto obtendríamos el diagrama de la figura 3.1.

El siguiente paso en el desarrollo de nuestro diagrama E-R será describir las relaciones que se establecen entre ellas. Para simplificar vamos a definir solo dos tipos de relaciones.

1. **Relación 1:N (uno a muchos)**: una entidad de tipo A se puede relacionar con N entidades de tipo B, pero una de tipo B solo se puede relacionar, como mucho, con una de tipo A. Este es el caso de los pedidos y los clientes: 'un cliente puede realizar muchos pedidos, pero un pedido sólo está asociado a un cliente'. Para representar esta restricción debemos colocar una línea que una estas entidades, y en uno de los extremos se coloca lo que se conoce como una *pata de gallo*. Otra alternativa es poner las etiquetas 1 y N o 1 e infinito (∞). En este caso vamos a mostrar las tres opciones. Además vamos a añadir en la entidad B el identificador de la entidad A. Es decir, en cada pedido se guarda el identificador del cliente que lo ha realizado. Añadir el identificador del pedido en el cliente no tendría sentido, ya que un cliente puede realizar más de un pedido. En nuestro diagrama la relación entre pedidos y clientes quedaría tal y como se ve en la figura 3.2.

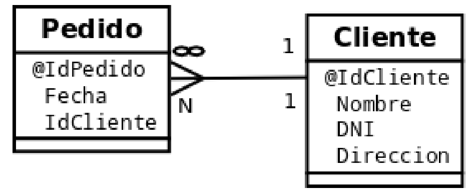


Figura 3.2: Diagrama Entidad-Relación: relación 1:N entre pedidos y clientes

2. **Relación N:M (muchos a muchos)**: una entidad del tipo A se puede relacionar con N del tipo B, y viceversa. Este es el caso de los pedidos y los productos: 'un producto puede aparecer en muchos pedidos, y un pedido puede contener muchos productos'. En este caso la solución pasa por crear una entidad intermedia y establecer dos relaciones 1:N entre esta nueva entidad y las entidades A y B. Siguiendo con los pedidos vamos a crear la entidad *DetallePedido*, e igual que se hizo anteriormente, le añadimos un identificador. Si recordamos el enunciado, de cada producto dentro de un pedido se pide una determinada cantidad, este valor será un atributo de la nueva entidad. De esta forma nuestro diagrama completo quedará como el de la figura 3.3.

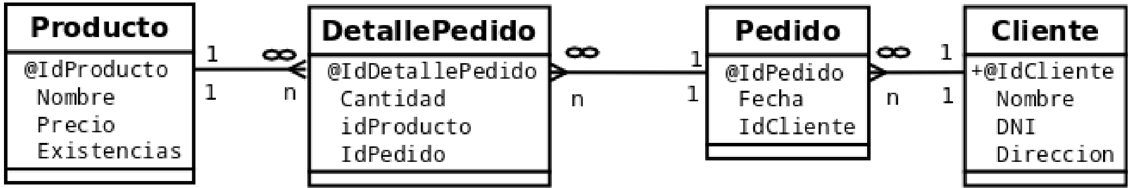


Figura 3.3: Diagrama Entidad-Relación completo

Existe otra alternativa que, aunque genera algunos problemas en los que no vamos a entrar, también es válida y ha sido frecuentemente utilizada. La diferencia estriba en que la clave de la entidad intermedia se forma mediante la unión de las claves de la entidades A y B. Esta alternativa quedaría reflejada en el diagrama de la figura 3.4:

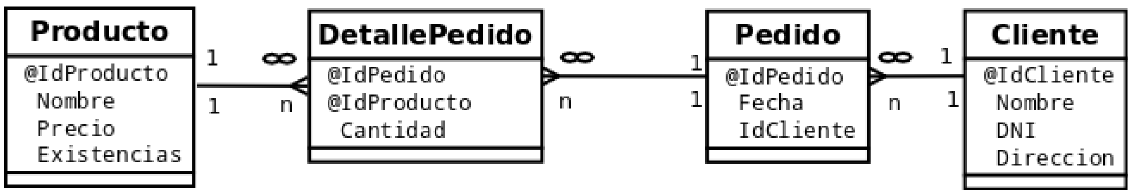


Figura 3.4: Diagrama Entidad-Relación completo alternativo

3.6. Modelo Relacional

Podemos considerar el Modelo Relacional como el más común en la implementación de bases de datos. Tal y como mencionamos anteriormente los SGBD más habituales son los SGBD Relacionales, es decir, los basados en este modelo. El elemento fundamental de este modelo es **la relación** que no es otra cosa que **una tabla**. El origen de este modelo proviene de las matemáticas, más concretamente de la teoría de conjuntos y relaciones, de ahí su nombre. De forma sencilla podemos ver una tabla como una serie de columnas fijas y un número de filas variable. Veamos un ejemplo:

Clientes

IdCliente	Nombre	DNI	Direccion
1	Luis	02345678F	Plaza de los Desahucios, 3
2	Pepa	52555698 D	Av. Los Corruptos, 666
3	Paco		c/ Ere del Hospital, 20
4	Luis	535678X	c/ Los Sobres de Luis, 77

Una *relación* (*tabla*) está formada por un número fijo de *columnas* o *campos* y uno variable de *filas* o *tuplas*.

Diseñar una BD consiste básicamente en diseñar un conjunto de tablas para almacenar nuestros datos; y diseñar una tabla consiste a su vez en definir las columnas que la forman. A cada columna le daremos un nombre, le asociaremos un tipo de datos (entero, cadena, decimal, fecha, etc.) y las restricciones que se consideren necesarias (por ejemplo, que las existencias de un producto no pueden ser negativas).

El proceso de diseño de una base de datos empieza por realizar el diagrama Entidad-Relación que vimos en el apartado anterior. Pero como resultado debemos obtener un conjunto de tablas que se puedan crear en un SGBD generalmente utilizando un lenguaje como el SQL.

El modelo Entidad-Relación y el modelo Relacional están muy bien *relacionados*. Fundamentalmente porque a partir del diagrama E-R se puede generar, de forma automática, el código SQL básico para crear el conjunto de tablas que conforman nuestra base de datos de pedidos.

En el caso de los pedidos el proceso es trivial, basta con crear una tabla por cada entidad. Cada una de estas tablas tendrá como columnas los atributos de la entidad. Luego ya lo tenemos casi todo hecho. Es cierto, que hay casos más complejos, pero en cualquier caso el paso de un diagrama E-R al conjunto de tablas Relacionadas sigue siendo prácticamente automático.

Dicho esto, la base de datos de pedidos quedaría formada por las siguientes tablas:

Clientes

IdCliente	Nombre	DNI	Direccion
1	Luis	02345678F	Plaza de los Desahucios, 3
2	Pepa	52555698 D	Av. Los Corruptos, 666
3	Paco		c/ Ere del Hospital, 20
4	Luis	535678X	c/ Los Sobres de Luis, 77

Productos

IdProducto	Nombre	Precio	Existencias
MA172	Manzana	2.35	50
PE111	Peras	3.14	72
KW001	Kiwis	4.56	61
PL011	Plátanos	1.69	54

Pedidos

IdPedido	*IdCliente*	Fecha
101	1	2011-07-12
102	1	2011-08-21
103	2	2011-08-25
110	3	2011-09-12

DetallesPedido

IdDetallePedido	*IdPedido*	*IdProducto*	Cantidad
1	101	MA172	10
2	101	PE111	12
3	103	KW001	7
4	103	MA172	13
5	102	PE111	2

Aprovechando nuestro ejemplo vamos a seguir explicando aspectos básicos del Modelo Relacional. Una idea que subyace es que la información solo debe guardarse una vez, debemos evitar almacenar información de forma redundante. Alguien, que no hubiera seguido nuestros pasos de diseño, podría haber planteado guardar en la tabla de $PEDIDOS$ la información del cliente que realiza cada pedido, pero en el momento en que un cliente realizara más de un pedido repetiríamos sus datos en distintas filas. Hasta aquí el problema podría ser únicamente el desperdicio de espacio, cosa que al precio del byte no parece muy importante. Pero si esa información del cliente empieza a sufrir modificaciones la cosa empezaría a complicarse, ya que nos obligará a mantener todas las filas asociadas a los pedidos de dicho cliente con la misma información, es decir, que la información sea *consistente*. El análisis de los problemas de repetición de información que presenta un determinado diseño se conoce como Normalización de Bases de Datos, pero está fuera de nuestros objetivos explicarlo aquí. Además si hacemos bien el diagrama E-R estos problemas de repetición de información no suelen presentarse.

Si observamos las tablas vemos que para relacionar la información que está guardada en diferentes tablas, basta con repetir los identificadores en las mismas. Es decir la columna IdCliente de la tabla de Pedidos nos permite relacionar el pedido con el cliente. Ello es posible porque no pueden existir dos clientes con el mismo identificador.

Aquí surgen algunos conceptos fundamentales de este modelo:

- **Clave primaria (primary key, PK)**: En una tabla se denomina clave primaria a la columna (o columnas) que nos permiten identificar de forma única a una fila. Es decir que dado un valor de la clave primaria solo podemos encontrar como mucho una fila. Por tanto los valores de la clave primaria no se pueden repetir dentro de la tabla. Debido a esa capacidad de identificar a una sola fila, en muchas tablas a la columna que forma la clave primaria se le suele llamar identificador (*IdCliente, IdPedido*), aunque también es frecuente utilizar el término *código (CodProducto)*. Es importante ver que *IdCliente* funciona como clave primaria en la tabla de *Clientes* y que no lo hace en la tabla de *Pedidos*, donde podría tomar valores repetidos (tantos como pedidos realice un mismo cliente).

  Nota: en las tablas se han representado mediante negrita las columnas que forman la clave primaria.

- **Clave ajena (foreign key, FK)**: En una tabla se denomina clave ajena a aquella que hace referencia a una clave primaria de otra tabla. Este es el método que tenemos para no tener que repetir la misma información en diferentes tablas. Los valores de una FK se pueden repetir dentro de la tabla, pero deben existir previamente en la tabla a la que hacen referencia. En nuestro caso la columna *IdCliente* de la tabla de *Pedidos* es FK de la tabla de *Clientes* y de esta forma no tenemos que guardar toda la información del cliente que hizo el pedido, sino únicamente su identificador, como ya acabamos de comentar. Además no podremos almacenar un pedido con un valor de *IdCliente* que no exista en la tabla de clientes.

  Nota: en las tablas se han representado mediante cursiva las columnas que son clave ajena.

- **Valores únicos (UNIQUE)**: Evidentemente la columna *DNI* de la tabla de *Clientes* tiene *casi* las mismas características que *IdCliente*, ya que tampoco debería repetirse (en este caso tendríamos dos clientes con el mismo DNI o el mismo cliente guardado dos veces) y también permitiría identificar de forma única a una fila. Por eso, cuando esto ocurre, también se dice que son una clave candidata. Los valores *UNIQUE* tienen otra característica que las diferencian de las claves primarias y es que pueden dejarse sin valor (con valor *null*) como en el DNI de Paco. Por tanto en una tabla podremos tener varias candidatas a ser clave primaria, pero solo una de ellas lo será, normalmente la que nos resulte más funcional.

  Analicemos otro caso interesante que ocurre en la tabla de Detalles_Pedido. Veamos el siguiente ejemplo:

  DetallesPedido

IdDetallePedido	*IdPedido*	*IdProducto*	Cantidad
10	101	MA172	10
11	101	MA172	20

  Vemos que el pedido 101, incluye dos veces las manzanas (MA172) y realmente no sabemos si el cliente quiere 10, 20 ó 30 manzanas. Es decir, deberíamos evitar que un producto aparezca dos veces en el mismo pedido. Para ello si definimos las columnas *IdPedido* y *IdProducto*, de forma conjunta como *UNIQUE*, hemos resuelto nuestro problema: el par (101,MA172) no puede repetirse a lo largo de la tabla.

Ya que hemos mencionado el **valor *null*** vamos a comentar algo sobre él. En principio el valor nulo es un valor válido para cualquier columna a no ser que específicamente lo prohibamos. Los valores nulos son un mal necesario que generan gran número de problemas. Para empezar, en la mayoría de los casos no sabemos qué significan. En nuestro caso Paco tiene un *null* en el campo DNI, pero realmente no sabemos si es debido a que Paco no tiene DNI -por ser un niño o un

extranjero- o que se trata de una persona mayor que cuando le dimos de alta no llevaba el DNI, ni se acordaba de él. La propia representación de valor nulo también nos puede generar problemas. Una alternativa es utilizar la palabra *null* para representarlo, otra es dejarlo en blanco, pero en este caso, si nos referimos a una cadena de texto, la representación del valor nulo coincidirá con una cadena de blancos o tabuladores, incluso con la cadena vacía. Si hablamos de números no es lo mismo un cero que un nulo aunque en este caso su representación no lleva a engaños. Los SGBD nos facilitan mecanismos para poder almacenar y operar con valores nulos.

3.7. Uso básico del lenguaje SQL

Ya hemos hecho referencia al SQL como un lenguaje estándar para trabajar con bases de datos. A menudo se diferencia entre lenguajes de definición de datos (LDD o DDL en inglés) y lenguajes de manipulación de datos (LMD o DML en inglés). Los LDD son aquellos que incluyen todas aquellas órdenes que nos permiten crear, modificar y borrar objetos en nuestros SGBD, ya sea la propia base de datos, tablas, tipos de datos, índices, usuarios, etc. Con el LDD definiremos la estructura en la que guardaremos nuestros datos. Los LMD incluyen aquellas órdenes que nos permiten insertar, modificar, borrar y consultar información en nuestra BD. El SQL (Structured Query Language) es un lenguaje comercial que soporta ambos tipos de órdenes. Es un lenguaje muy antiguo que ha ido soportando el paso del tiempo y se ha ido adaptando en lo posible a algunas novedades de los SGBD, y sigue siendo el más utilizado actualmente e integrado en los todos los SGBD comerciales más populares. Existe un estándar SQL que se va revisando cada cierto tiempo pero cada SGBD dispone de su propia implementación, consecuentemente existen algunas diferencias entre unos y otros.

En muchos casos los SGBD disponen de entornos gráficos que facilitan la creación y mantenimiento de nuestras bases de datos, de tal forma que no se hace necesario conocer SQL. Aunque realmente por debajo del interface gráfico se está ejecutando código SQL. Está fuera del ámbito de este curso el aprender SQL, pero para que el alumno se haga una idea de este lenguaje, vamos a mostrar algunas órdenes sencillas.

3.7.1. Ejemplos de órdenes SQL

Retomando nuestro ejemplo el código SQL que crearía nuestra BD con sus tablas sería:

```
/* ------------------------------------------------*/
/* Cdigo SQL para la creacion de la Base de Datos  */
/* ------------------------------------------------*/

/* Creacion de la base de datos*/
CREATE DATABASE Pedidos;

/* Creación de tablas */
CREATE TABLE Clientes (
IdCliente integer not null primary key ,
Nombre  char(15),
DNI  varchar(10),
Direccion text
);

CREATE TABLE Productos (
IdProducto char(5)  not null primary key ,
Nombre varchar(15),
Precio decimal (5,2) ,
```

```
Existencias integer
);

CREATE TABLE Pedidos (
IdPedido integer not null primary key,
Fecha   date,
IdCliente integer not null references Clientes
);

CREATE TABLE DetallesPedido (
IdPedido integer not null references Pedidos,
IdProducto char(5) not null references Productos,
Cantidad integer,
PRIMARY KEY ( IdPedido,IdProducto )
);
```

Como se puede ver la creación de una tabla se concreta en ir definiendo cada columna con su tipo de dato y sus restricciones. El texto que queda entre los /* y */ son comentarios.

Otras órdenes básicas de manipulación de datos serían:

- **insert**: con esta orden insertaremos filas en una tabla. Ejemplo:

  ```
  insert into Productos values ('MA172', 'Manzana',2.35, 50)
  ```

- **update**: con esta orden modificaremos filas en una tabla. Ejemplo:

  ```
  update Productos
     set Precio = Precio * 1.05
  where IdProducto = 'MA172';
  ```

  De esta forma incrementamos el precio de las manzanas en un 5 por ciento

- **delete**: con esta orden borraremos filas de una tabla. Ejemplo:

  ```
  delete from Productos
  where Precio > 50;
  ```

  Así borramos los productos cuyo precio supere los 50 euros.

- **select**: con esta orden buscamos información en una tabla. Ejemplo:

  ```
  select IdProducto, Nombre
  from Productos
  where Existencias = 0;
  ```

  Así obtendremos los productos sin existencias.

- **select**: con esta orden buscamos información en las tablas de pedidos y clientes.

```
select IdPedido, Fecha
from Pedidos join Clientes on Pedidos.IdCliente = Clientes.IdCliente
where Fecha between '2011-09-04' and '2014-09-30';
```

Estos nos lista los números de pedido, con su fecha y el nombre del clientes que lo realizó de aquellos pedidos de septiembre de 2014.

3.8. SGBD en entornos profesionales de la ingeniería

Cualquier entorno profesional suele tener asociado un sistema de información. Hace años la relación de la ingeniería con la informática se veía más centrada en la capacidad de cálculo de los ordenadores mientras que los bancos eran los que desarrollaban sistemas cuya característica era trabajar con grandes volúmenes de información.

Algunos sistemas de ingeniería, como los sistemas de CAD, han manejado importantes volúmenes de información, pero no gestionados por SGBD de propósito general. Actualmente muchas ingenierías utilizan un SGBD para el propio funcionamiento de la empresa, esto ha supuesto que los SGBD sean conocidos y se empiecen a utilizar en los propios procesos de ingeniería y estén reemplazando a las habituales Hojas de Cálculo.

El uso de Hojas de Cálculo se ha estandarizado ya que no necesitan de unos conocimientos previos tan especializados como las BD. Aún así son muchos los profesionales que cuando descubren las ventajas de una BD, abandonan las Hojas de Cálculo. Está claro que cada herramienta tiene su utilidad y *es para lo que es*.

Hoy en día los SIG (Sistemas de Información Geográficos) han revolucionado el mundo de las bases de datos. Algunos SGBD ya incorporan funcionalidades para manejar este tipo de información espacial como por ejemplo Postgis.

3.8.1. Bases de Datos espaciales y geográficas

El soporte de los datos espaciales en las BD es importante para el almacenamiento y la realización de consultas eficientes de los datos basados en las posiciones espaciales. Dos tipos de datos espaciales son especialmente importantes:

1. Los **datos de diseño asistido por computador (CAD)**, que incluyen información espacial sobre el modo en que los objetos (edificios, coches, aviones, etc.) están construidos. Los sistemas CAD tradicionalmente almacenaban los datos en la memoria durante su edición u otro tipo de procesamiento y los volvían a escribir en ficheros al final de la sesión de edición. Pero un diseño de gran tamaño, como el de un avión, podía resultar imposible guardarlo en memoria. Los objetos almacenados en las BD CAD suelen ser objetos geométricos de los que además se puede almacenar información no espacial por ejemplo el material del que están construidos, color u otras características.

2. Los **datos geográficos** (mapas de carreteras, tierra, mapas topográficos, mapas catastrales, imágenes de satélite, etc.).

   Los sistemas de información geográfica son bases de datos adaptadas tanto para el almacenamiento de los datos geográficos como para el procesamiento de los mismos. Los datos geográficos, como los mapas o imágenes de satélite son de naturaleza espacial, pero se diferencian de los datos de diseño en ciertos aspectos. Los mapas pueden proporcionar no sólo información sobre la ubicación (fronteras, ríos, carreteras, ...) sino también información mucho más detallada asociada con la ubicación (como la elevación del terreno, el tipo de suelo, el uso de la tierra, etc.).

   Se diferencias dos **tipos de datos geográficos**:

Tipo array consisten en mapas de píxeles (picture element) en 2 o más dimensiones. Un ejemplo son las imágenes de satélite, con las que podemos obtener arrays bidimensionales (la cobertura nubosa en la que cada píxel almacena la visibilidad de las nubes en una región concreta) o tridimensionales (la temperatura a distintas altitudes, la temperatura superficial en diferentes momentos, etc.).

Tipo vectorial están formados a partir de objetos geométricos básicos como puntos, segmentos rectilíneos, polilíneas de 2 dimensiones, cilindros, esferas u otros.

Los datos cartográficos suelen representarse en formato vectorial. La representación vectorial es más precisa que la de arrays en algunas aplicaciones.

Aplicaciones de los datos geográficos:

Mapas en línea: existen de muchos tipos aunque los más populares son los mapas de carreteras. Muchos son interactivos y permiten el cálculo de rutas. Disponen de información como el trazado de carreteras, límites de velocidad, condiciones de las vías, servicios, etc.

Sistemas de navegación: proporcionan mapas de carreteras, de rutas de montaña, o cartas de navegación náutica. Se basa en el uso de GPS (Global Position System) La unidad GPS halla la ubicación en términos de: latitud, longitud y elevación. El sistema de navegación puede consultar la BD geográfica para hallar el lugar en que se encuentra y la ruta a seguir.

Sistemas de información de redes de distribución: en ellos se almacena información sobre servicios de telefonía, electricidad, suministro de agua, etc. No solo el trazado sino la descripción de los elementos que constituyen dichas redes.

Parte IV

Componentes hardware y software de un sistema informático

CAPÍTULO 4

Componentes hardware y software de un sistema informático

4.1. Estructura y funcionamiento de un computador

4.1.1. El sistema informático, hardware y sofware

Se denomina **hardware** al conjunto de elementos físicos que componen un ordenador mientras que **software** hace referencia a los programas que un ordenador puede ejecutar.

Un ordenador está formado por un conjunto de componentes, la mayoría de tipo electrónico, que no son capaces de realizar por sí mismos demasiadas funciones. Necesitan de otros componentes no físicos que los pongan en funcionamiento; nos estamos refiriendo a los programas.

Así, la unión de hardware y software es la que permite a un **sistema informático** realizar su función: procesar información. Es conveniente señalar, a pesar de su evidencia, que el hardware y el software son perfectamente inútiles aisladamente: de nada nos sirve un ordenador si no tenemos ningún programa que ejecutar, y de nada nos sirve tener muchos programas si no disponemos de un ordenador que los ejecute.

Figura 4.1: Las dos partes de un sistema informático

4.1.2. Componentes físicos. El hardware

Ya sabemos que el hardware es la parte física del ordenador. Son los elementos tangibles tales como la memoria, la fuente de alimentación, los cables, las impresoras, etc.

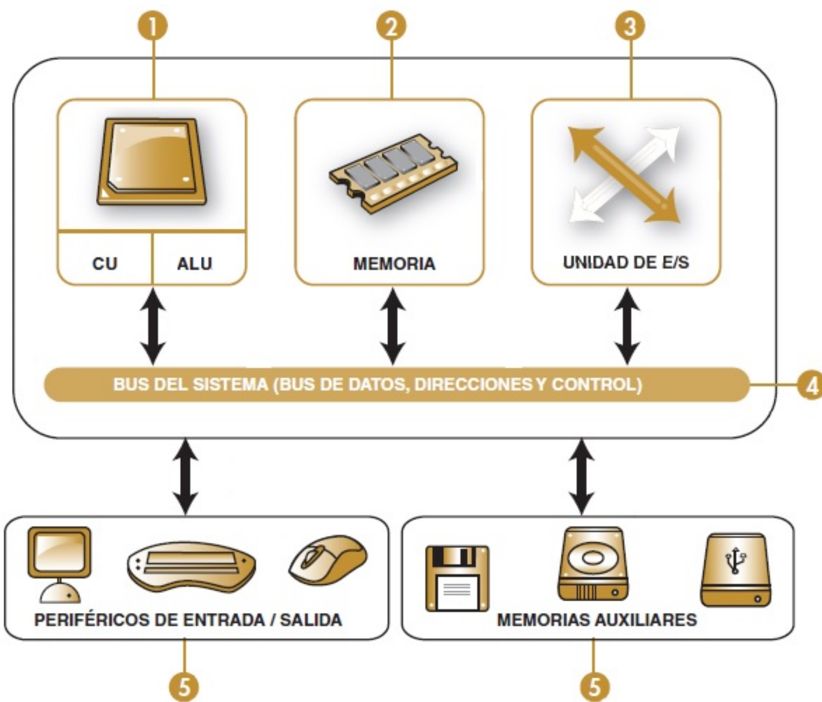


Figura 4.2: Arquitectura de un ordenador

La arquitectura típica de un ordenador (Von Neumann 1945) identifica los siguientes componentes hardware:

1. Unidad Central de Proceso (CPU) formada por:
   - Unidad Aritmético Lógica (ALU)
   - Unidad de Control (CU)
2. Memoria cental o principal
3. Unidad de Entrada/Salida (E/S)
4. Buses
5. Dispositivos periféricos

La Unidad Central de Proceso

La **unidad central de proceso** o **CPU**, también denominada **procesador**, es el elemento encargado del control y la ejecución de las operaciones que se efectúan dentro del ordenador con el fin de realizar el tratamiento automático de la información. Es la parte pensante del ordenador, se encarga de todo: controla los periféricos, la memoria, la información que se va a procesar, etc.

La CPU está formada por la **Unidad Aritmético Lógica (ALU)**, que se encarga de realizar operaciones básicas sobre los datos (tales como sumar y restar); **la Unidad de Control (CU)**, que coordina todas las actividades y elementos del ordenador; y una serie de **registros**, que son pequeños espacios de almacenamiento donde se guarda de manera temporal la información que maneja la CPU.

La comunicación entre la CPU y la memoria principal se establece mediante una colección de *cables* llamados **buses**, a través de los cuales se transmiten los bits que se leen o escriben. La CPU

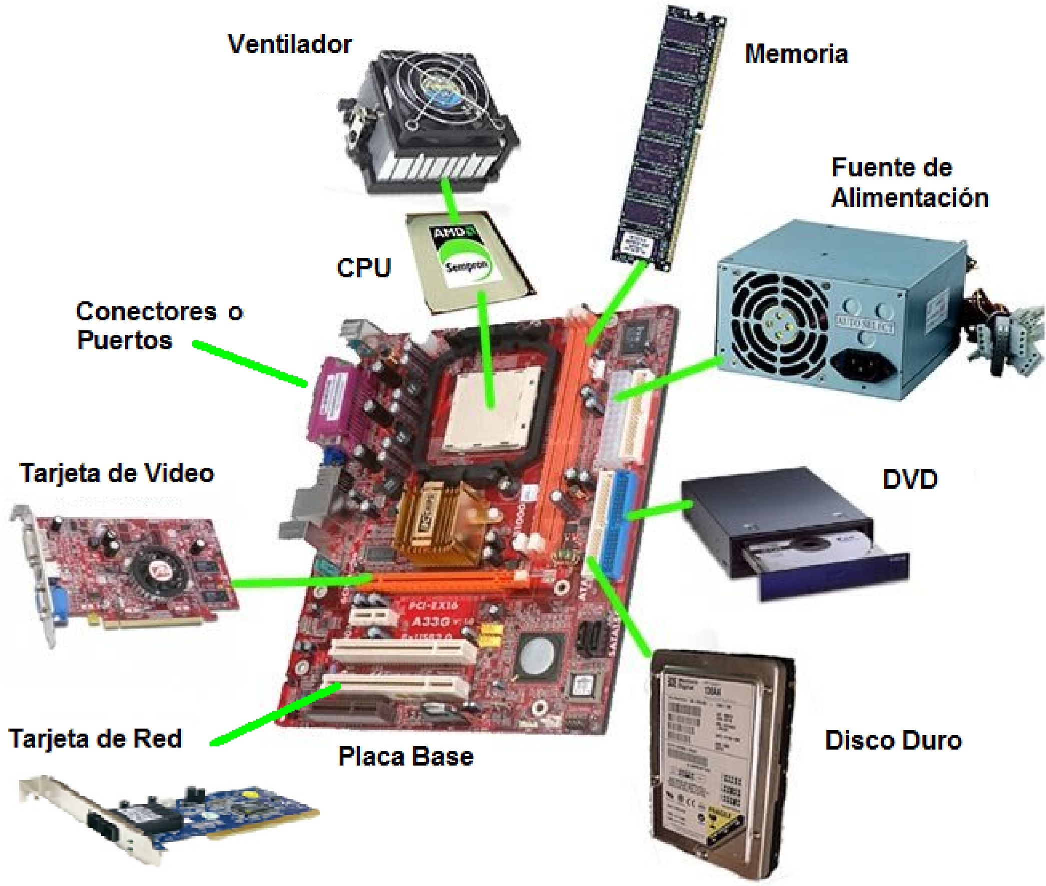


Figura 4.3: Ejemplo de componentes de un ordenador personal

extrae o lee datos de la memoria principal proporcionando la **dirección** en la memoria donde está la información que se necesita. De manera similar, la CPU escribe datos en la memoria proporcionando la dirección de destino en la memoria donde se quiere guardar la información.

La Memoria

En el tema 1 hemos visto cómo los datos de diferentes tipos (enteros, reales, booleanos, texto, etc.) se pueden codificar mediante secuencias de bits; y en el tema 2 hemos comentado cómo los programas se traducen finalmente a un conjunto de instrucciones en forma de secuencias de bits (código máquina o binario). Téngase en cuenta que para que un programa pueda ser ejecutado debe haber sido previamente cargado en la memoria principal del sistema. La memoria del ordenador no es más que un almacén de bits donde se guardan todos los programas y los datos con los que se va a trabajar y que van a ser procesados.

La memoria principal está dividida en **posiciones** de memoria de un determinado número de bits (**palabra de memoria**). El tamaño de la palabra de memoria (**n**) ha ido variando: 1 bit, 8 bits, 16 bits, 32 bits, 64 bits, etc. Siempre que se quiera escribir o leer un dato o instrucción en una posición de memoria hay que especificar su **dirección**. Las direcciones de memoria toman valores entre 0 y $2^n - 1$.

La memoria principal descrita anteriormente es del tipo **RAM** (**R**andom **A**ccess **M**emory), admite operaciones de lectura y escritura y permite el acceso a las posiciones de memoria en cualquier orden. Se trata de memorias volátiles en el sentido de que necesitan la alimentación eléctrica para

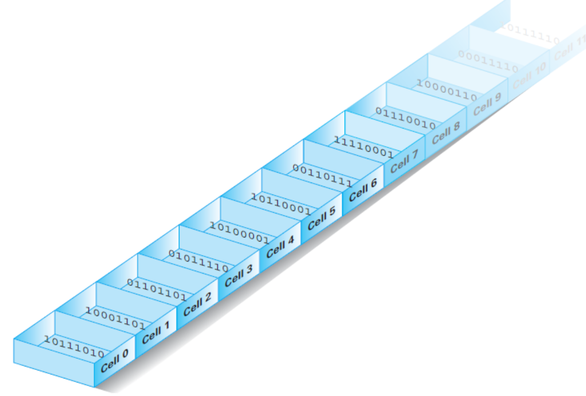


Figura 4.4: Posiciones de memoria con palabras de 8 bits

conservar la información.

Existe otro tipo de memoria llamada **ROM** (**R**ead **O**nly **M**emory) que sólo permite operaciones de lectura y que contiene programas especiales para cargar e iniciar el arranque del ordenador. En este tipo de memoria la información es almacenada de forma permanente. El software que integra la ROM forma el **BIOS** del ordenador (**B**asic **I**mput **O**utput **S**ystem) o sistema básico de entrada/salida.

Hay que tener en cuenta que la memoria central es muy rápida pero no tiene gran capacidad de almacenamiento. Por eso los ordenadores utilizan **memorias de almacenamiento externo** o **almacenamiento secundario** o **almacenamiento masivo**, como los discos magnéticos, CDs, DVDs, cintas magnéticas, pen drives, etc. Estos dispositivos son sensiblemente más lentos que la memoria central, del orden de un millón de veces más lentos, con tiempos del acceso a los datos del orden de *ms* (milisegundos) frente a *ns* (nanosegundos) que permite la memoria principal.

Para saber más: memoria caché y memoria virtual

Es instructivo comparar los tipos de memoria dentro de un ordenador en relación a su funcionalidad. Los registros se utilizan para mantener los datos que la CPU va a utilizar de manera inmediata, la memoria principal se utiliza para almacenar los datos que se necesitarán en el futuro cercano, y la memoria de almacenamiento masivo se utiliza para guardar datos que no es probable que sean necesarios en el futuro inmediato.

Muchas máquinas están diseñadas con un nivel de memoria adicional, llamada **memoria caché**. La memoria caché es una pequeña cantidad (quizás varios cientos de KB) de memoria de alta velocidad situada dentro de la propia CPU. En este área de memoria especial, la máquina intenta guardar una copia de la parte de la memoria principal que es de interés actual. En este contexto, las transferencias de datos que normalmente se hacen entre los registros y la memoria principal se realizan entre registros y memoria caché. Los cambios realizados en la memoria caché se transfieren colectivamente a la memoria principal en un momento más oportuno. El resultado es una CPU que puede ejecutar las instrucciones más rápidamente, ya que no se retrasa por la comunicación con la memoria principal. Por otra parte, la **memoria virtual** se desarrolla con el propósito de poder hacer uso de más memoria de la que físicamente se dispone. La idea consiste en usar la memoria secundaria, generalmente un disco duro, como si fuera memoria principal.

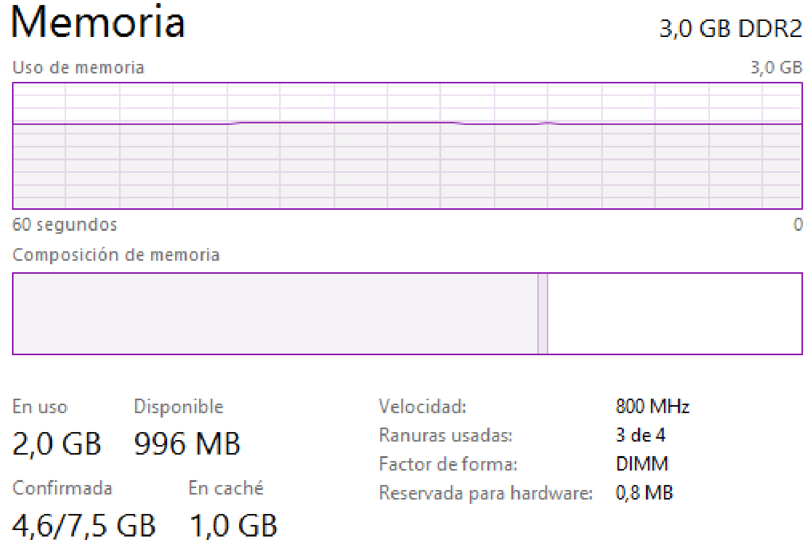


Figura 4.5: Información sobre la memoria en un ordenador con Windows 8

La capacidad de la memoria

En los primeros ordenadores la memoria empezó a medirse en bits, luego en nibbles (que son 4 bits) luego en bytes (que son 8 bits), luego en kilobytes (que son 1024 bytes). Todos los valores potencias de dos ya que lo que se guarda es información binaria. La proximidad del valor 1024 a 1000 fue lo que hizo que la comunidad informática adoptara el prefilo *kilo* para hacer referencia a esta cantidad (1KB = 1 kilobyte = 1024 bytes = 2^{10} bytes).

De esta forma, cuando hablamos de capacidad de memoria el megabyte (MB) son 1048576 bytes (1024 kilobytes) y un gigabyte (GB) son 1073741824 bytes, o 1048576 kilobytes, o 1024 megabytes.

Sin embargo, esta forma de utilizar los prefijos choca con el significado que tienen en el Sistema Internacional de Unidades, en el que hacen referencia a potencias de 10. Así por ejemplo, cuando medimos distancias, 1 kilómetro son 1000 (10^3) metros; y cuando medimos frecuencias, 1 megahercio son 1000000 (10^6) hercios.

Como norma general, los prefijos kilo-, mega-, etc. se refieren a potencias de dos cuando se usan en el contexto de la memoria del ordenador pero se refieren a potencias de 10 en otros contextos.

Unidades de entrada/salida y buses

La **Unidad de entrada/salida** sirve para comunicar el procesador y el resto de componentes internos del ordenador con los periféricos de entrada/salida y las memorias de almacenamiento externo.

El intercambio efectivo de información entre los distintos componentes del ordenador se hace a través de los **buses**. El **bus** es un conjunto de líneas hardware (*pistas de un circuito impreso* o *cables*) utilizadas para la transmisión de datos entre los componentes de un sistema informático.

En un sistema informático se distinguen tres tipos de buses (ver figura 4.2):

1. **Bus de datos**: transmite información entre la CPU y la memoria o los periféricos.

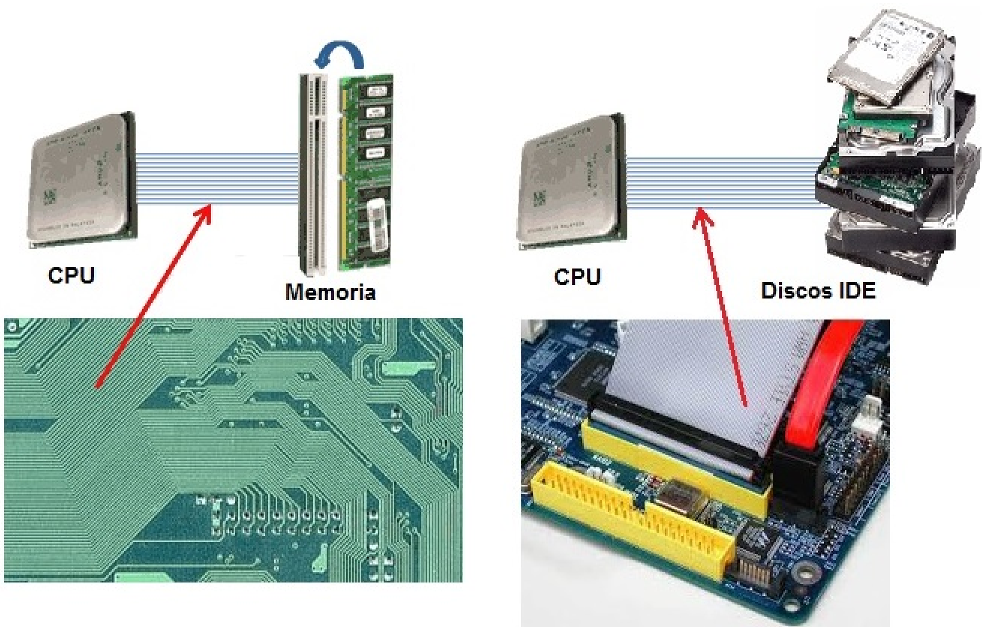


Figura 4.6: Buses en forma de pistas de un circuito impreso (izquierda) y de cables (derecha).

2. **Bus de direcciones**: identifica el dispositivo al que va destinada la información que se transmite por el bus de datos.

3. **Bus de control**: organiza y redirige hacia el bus pertinente la información que se tiene que transmitir.

Ejecución de un programa

La tarea básica que realiza un computador es la ejecución de los programas. Cada programa consta de un conjunto de instrucciones almacenadas en memoria. Es la CPU la que se encarga de procesar cada una de las instrucciones especificadas en un programa.

El punto de vista más sencillo consiste en considerar que el procesamiento de cada instrucción, denominado **ciclo de instrucción**, consta de dos pasos:

1. **Ciclo de lectura**: cada instrucción es leída (una a la vez), desde la memoria, por la unidad de control de la CPU (CU)

2. **Ciclo de ejecución**: cada instrucción es ejecutada por la unidad aritmético lógica de la CPU (ALU)

La repetición de la lectura y ejecución (pasos 1 y 2 respectivamente), conforman la **ejecución de un programa**. El ciclo de instrucción se representa en la figura 4.7, basándose en esta descripción simplificada de dos pasos que se acaba de explicar.

Las CPUs realizan sus operaciones al ritmo que marca el reloj del sistema. El reloj de un ordenador es un circuito, llamado oscilador, que genera impulsos que se utilizan para coordinar las actividades de la máquina. Cuanto más rápido vaya el reloj más rápido el ordenador ejecutará sus instrucciones. Sin embargo, hay que tener en cuenta que diferentes diseños de CPU pueden realizar diferentes cantidades de trabajo en un ciclo de reloj.

Las velocidades de reloj se miden en hercios (abreviado como Hz) donde un Hz es igual a un ciclo (o pulso) por segundo. Las velocidades típicas de reloj en ordenadores de sobremesa están en el rango de unos pocos cientos de MHz (modelos antiguos) a varios GHz. MHz es la abreviatura de megahercios (10^6 Hz) y GHz es la abreviatura de gigahercios, que es 1000 MHz o 10^9 Hz.

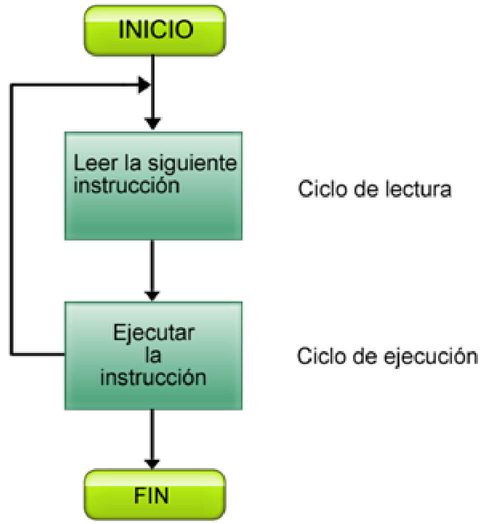


Figura 4.7: Ejecución de programas. Ciclo de instrucción

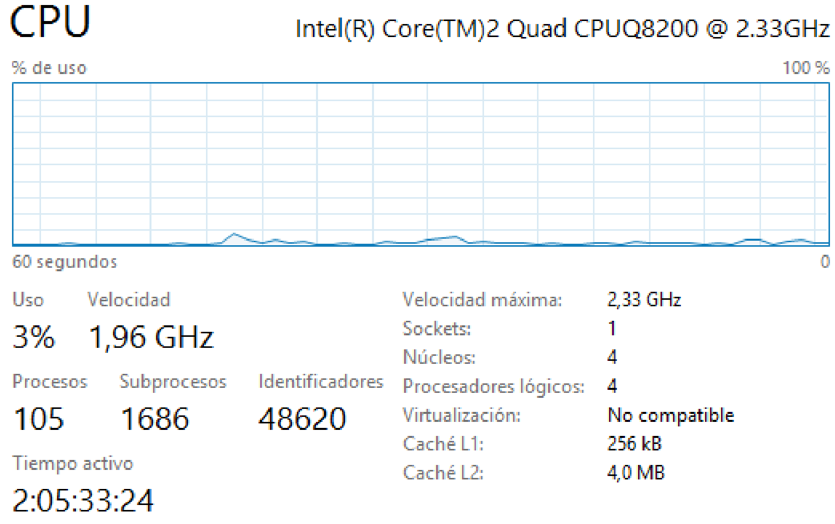


Figura 4.8: Información del procesador en un ordenador con Windows 8

Para saber más: comparación de ordenadores

Cuando vamos a comprar un ordenador personal muchas veces se utilizan las velocidades de reloj para comparar máquinas. Desafortunadamente, ya hemos dicho que el diseño de la CPU influye en la cantidad de trabajo realizada en un ciclo de reloj, y por lo tanto la velocidad de reloj sólo es relevante en la comparación de las máquinas con CPUs de tipos similares. Si queremos comparar una máquina basada en un procesador Intel con uno basado en AMD, por ejemplo, sería más significativo realizar una evaluación comparativa (*benchmarking*), que es el proceso de comparar el rendimiento de diferentes máquinas al ejecutar el mismo programa, conocido como *benchmark*. Existen *benchmarks* para CPUs, discos, tarjetas gráficas y sistemas en su conjunto.

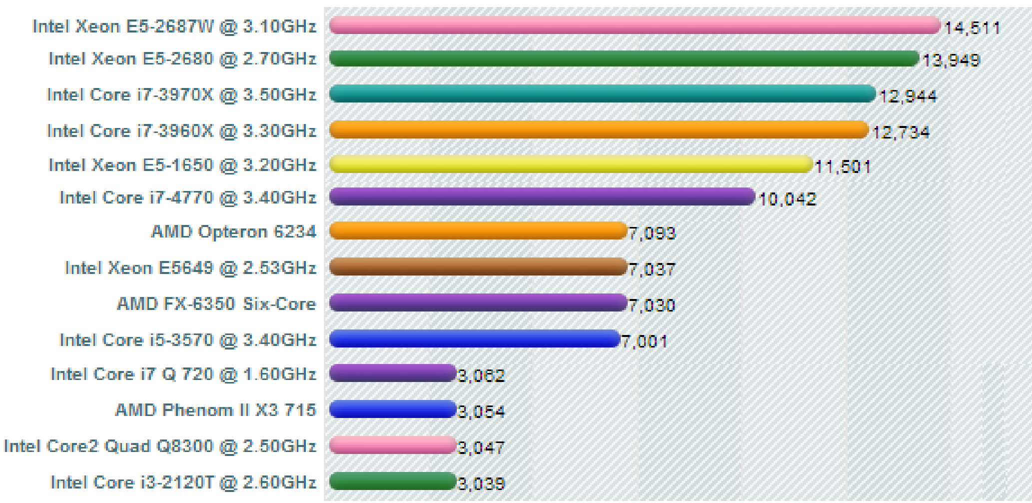


Figura 4.9: Tabla comparativa del rendimiento de CPUs

4.1.3. Dispositivos periféricos

Los periféricos son dispositivos hardware mediante los cuales el usuario puede interactuar con el ordenador.

Podemos distinguir entre periféricos de entrada, de salida y de entrada/salida, según la dirección de los datos que circulan entre el dispositivo y la memoria del ordenador. Si el dispositivo envía información a la memoria del ordenador, es un periférico de entrada (teclado, ratón, escáner, micrófono, ...). Si la información va de la memoria al periférico, entonces es de salida (pantalla, impresora, altavoz, ...). Si se envía o recibe indistintamente información desde la memoria, el periférico es de entrada/salida (tarjeta de red, fax, disco duro, pantalla táctil, ...).

Asimismo podemos distinguir entre periféricos locales y periféricos remotos, según su conexión al ordenador. Un periférico local, como el ratón, se encuentra cerca de la CPU conectado mediante cables que hacen las veces de prolongador de los buses del computador. Para un periférico remoto, como una impresora láser de un centro de cálculo, la conexión se realiza a través de una red de comunicaciones. Dentro de los periféricos locales también se puede diferenciar entre periféricos internos (como los discos duros o las unidades de DVD) que van dentro de la *caja* del ordenador y externos (teclados, ratones, impresoras, etc).

Existen dos clases de dispositivos periféricos que, aunque formen parte de algunos de los tipos anteriores, poseen unas características específicas que hacen que se estudien por separado: los dispositivos de almacenamiento (disco duro, unidad de DVD, lápiz USB, ...) y los dispositivos de comunicación (tarjeta de red, modem, router, switch, ...).

Los controladores de dispositivo

La unidad de entrada/salida sirve para comunicar el procesador y el resto de componentes internos del ordenador con los dispositivos periféricos a través de los buses. Sin embargo, los periféricos se conectan directamente a los buses sino que deben hacerlo a través de los llamados **controladores de dispositivo**. Estos controladores son **componentes del hardware** del ordenador que tienen unos conectores externos, llamados **puertos**, a los que se conectan los periféricos.

Los controladores de dispositivo pueden ser especializados, como los que controlan la pantalla o las unidades de disco, o de propósito general, como el controlador USB (**U**niversal **S**erial **B**us) o el Firewire. Estos últimos permiten que un mismo controlador maneje una gran variedad de

Figura 4.10: Ejemplos de periféricos de un ordenador personal

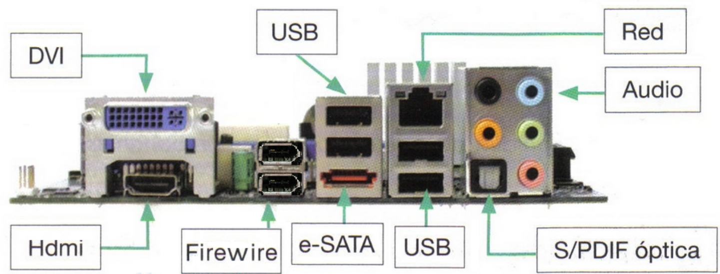


Figura 4.11: Puertos habituales de un ordenador personal

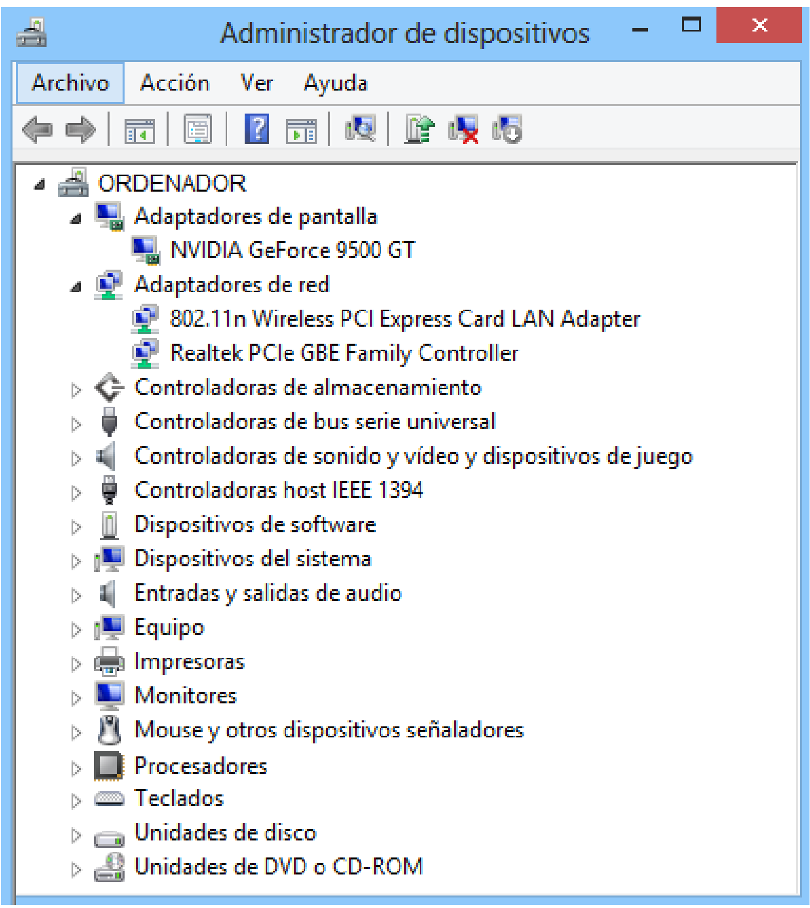


Figura 4.12: Ventana de administración de dispositivos en Windows 8

dispositivos. Por ejemplo, el controlador USB puede usarse para conectar al ordenador ratones, impresoras, discos, cámaras digitales, smartphones, etc.

No confundir este concepto con el de ***driver***, que en español también se traduce por controlador de dispositivo. Aunque en este caso *driver* es un **programa** que el sistema operativo utiliza para gestionar un determinado periférico (evidentemente a través del controlador de dispositivo hardware correspondiente). Así por ejemplo, si cambiamos la impresora USB de nuestro ordenador es casi seguro que tengamos que instalar un nuevo *driver* para poder utilizarla.

Para saber más: resolución de monitores

Los resolución de los monitores se mide en **píxeles** o puntos de color que constituyen la pantalla. A mayor número de píxeles mayor resolución. La imagen que se ve en cada momento en un monitor se crea fijando el color de los millones de píxeles de la pantalla (1 megapíxel, Mpx, equivale a 1 millón de píxeles). Es habitual dar la resolución de la forma *nº_píxeles_horizontal x nº_píxeles_vertical*. Así, una resolución de, por ejemplo 1024*x*768, indica que la pantalla tiene 1024 píxeles en horizontal y 768 en vertical. Para formar el color de cada píxel se utiliza una combinación de tres componentes: R(red), G (green), B (Blue). La combinación de las tres componentes, R, G y B, cada una con su magnitud, fija el color y el brillo de cada pixel. Por ejemplo: amarillo = 1/2 rojo + 1/2 verde, magenta = 1/2 rojo + 1/2 azul, cian = 1/2 verde + 1/2 azul.

RESOLUCIONES		
HDTV	High Definition Television	1920 x 1080
UXGA	Ultra eXtended Graphics Array	1600 x 1200
WUXGA	Widescreen Ultra eXtended Graphics Array	1920 x 1200
SXGA	Super eXtended Graphics Array	1280 x 1024
XGA	eXtended Graphics Array	1024 x 768
WXGA	Wide eXtended Graphics Array	1440 x 900
WQUXGA	Wide Quad Ultra eXtended Graphics Array	3840 x 2400

Figura 4.13: Ejemplos de resoluciones de pantalla de ordenador

4.2. Dispositivos de almacenamiento

A diferencia de la memoria principal del sistema (RAM) que pierde todos sus contenidos si se desconecta la alimentación, los periféricos de almacenamiento conservan la información digital almacenada incluso cuando se apaga el computador.

Los más empleados son: discos duros, unidades ópticas (CD/DVD/Blu-ray) y memorias Flash. Los discos duros y unidades ópticas internas suelen utilizar conectores SATA (antes IDE), mientras que los dispositivos externos, incluyendo las memorias Flash, suelen emplear USB (ver figura 4.14).

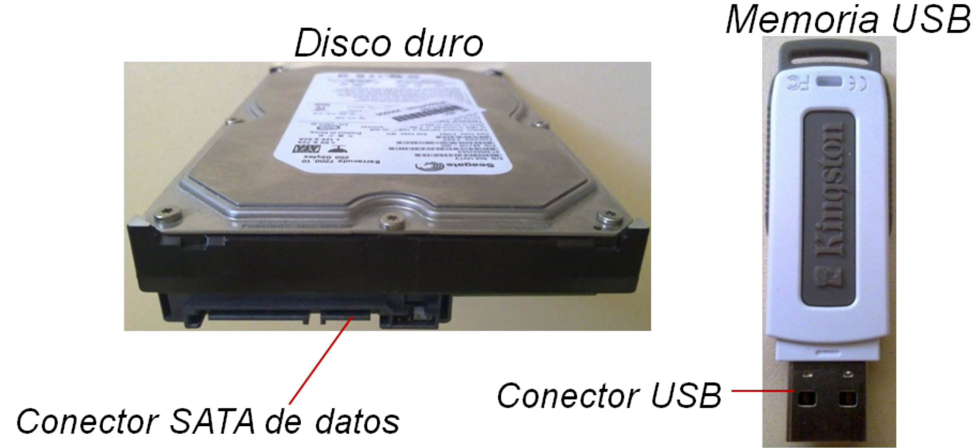


Figura 4.14: Periféricos de almacenamiento

Discos duros

Los discos duros (HDDs, **H**ard **D**isk **D**rives) almacenan grandes cantidades de información, del orden de 1 Tbyte. La velocidad de lectura y escritura es del orden de 100 Mbytes/segundo y el acceso a la información lleva del orden de 10 ó 15 ms (milisegundos).

Su funcionamiento se basa en el magnetismo. En la superficie de unos platos rígidos se deposita material magnético. Para cada plato se dispone de una cabeza de lectura y escritura en el extremo de unos brazos (ver figura 4.15). Estas cabezas pueden producir variaciones en el campo magnético, que queda registrado en el material magnético que cubre el disco, o pueden detectar las variaciones del campo magnético que se generan al hacer desplazarse (girar) el disco bajo la cabeza lectora. Los platos giran a alta velocidad, del orden de 5000, 7200, 10000 rpm (revoluciones por minuto).

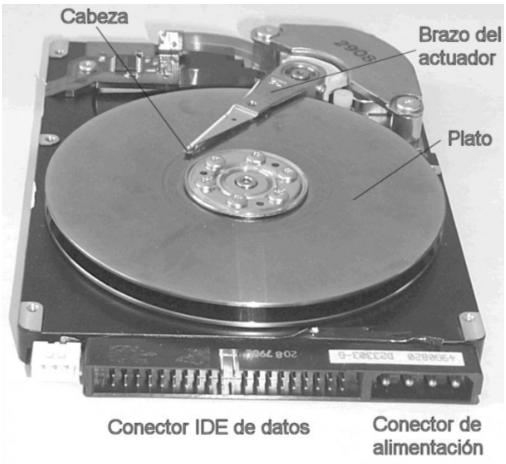


Figura 4.15: Interior de un disco duro

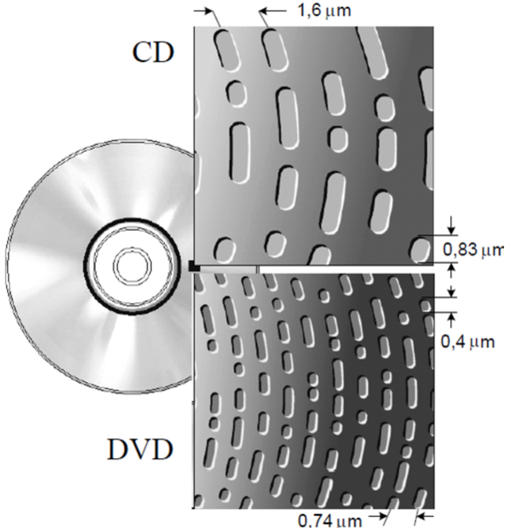


Figura 4.16: CD-ROM al microscopio

Discos ópticos

Hay tres generaciones: CD, DVD y Blu-ray. Las capacidades típicas son: 700 Mbytes, 8,4 Gbytes y 50 Gbytes. La velocidad de lectura y escritura de los discos Blu-ray (los más rápidos) es de unos 15 Mbytes/segundo.

La figura 4.16 muestra el aspecto idealizado de los pits y lands de un CD-ROM y un DVD-ROM. En el caso de un BD-ROM la diferencia es que las distancias aún son más ajustadas. Las imágenes hechas con microscopio son similares, pero menos educativas, ya que son mucho más ennegrecidas y difusas, y obviamente no muestran curvatura.

Su funcionamiento se basa en las propiedes ópticas de la superficie de estos discos, es decir, en la forma en que reflejan o dejan de reflejar la luz que incide sobre ellos. La información se almacena en forma de espiral. Los ceros y unos se codifican como cambios de reflexión de un haz láser. Dependiendo del tipo de disco, ROM, grabable o regrabable, los cambios de reflexión se consiguen con surcos (pits), o cambios químicos en el material.

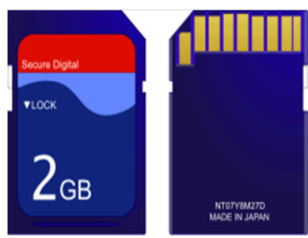


Figura 4.17: Memoria FLASH en forma de tarjeta

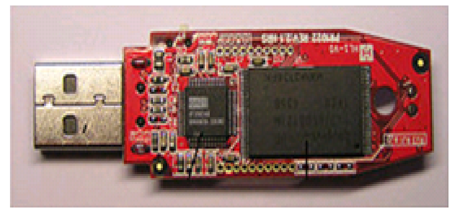


Figura 4.18: Memoria FLASH incrustada en un lápiz USB

Memorias Flash

La memoria **flash** es un tipo de memoria electrónica no volátil, es decir, que mantiene la información almacenada incluso cuando desaparece la alimentación. Al estar basada en circuitos electrónicos y no tener partes móviles como los discos y unidades opticas tradicionales, los tiempos de lectura y escritura que se pueden conseguir son inferiores que en éstos.

Actualmente tienen capacidades típicas comprendidas entre 4 Gbytes y 64 Gbytes y crecen rápidamente. Suelen presentarse en dos formatos: tarjetas de memoria (fig. 4.17) y memorias USB (pendrives, fig. 4.18).

Otra de las aplicaciones de la tecnología flash se encuentra en las tarjetas SD (Secure Digital) disponibles en tamaños muy pequeños (mini y micro), memoria SDHC (de alta capacidad) y memorias SDXC (Extended Capacity) que pueden exceder de un TB (terabyte) de capacidad. Dado su tamaño físico compacto, estas tarjetas se adaptan perfectamente a las ranuras de los pequeños dispositivos electrónicos. Por lo tanto, son ideales para cámaras digitales, teléfonos inteligentes, reproductores de música, sistemas de navegación de automóviles, etc.

Unidades de estado sólido (SSD)

Una **unidad de estado sólido** (SSD, **S**olid **S**tate **D**rive) es un dispositivo de almacenamiento secundario basado en chips de memoria flash. Sin embargo, a diferencia de las memorias flash convencionales se diseñaron como sustitutos de los discos magnéticos tradicionales. Al ser 100 % electrónico un SSD no tiene partes mecánicas en movimiento por lo que pueden acceder a los datos bastante más rápido que los HDDs. Un SDD consigue velocidades de lectura y escritura del orden de 500 Mbytes/segundo (5 veces más rápido que los discos magnéticos).

Figura 4.19: Kit de sustitución de un disco duro tradicional por una unidad SSD

Figura 4.20: Interior de una unidad SSD

4.3. Dispositivos de red

Las redes permiten la interconexión de varios computadores entre sí, la utilizaciónn conjunta de distintos dispositivos externos tales como discos, impresoras, etc., y el uso compartido de programas y ficheros de datos. Cada computador conectado a la red contempla los distintos dispositivos disponibles como si fueran propios.

Para llevar a cabo físicamente la comunicación, el computador necesita una interfaz de red. Las típicas son Ethernet y WiFi.

La interfaz Ethernet requiere un cable y un conector como los mostrados en la figura 4.21. La interfaz WiFi requiere una antena que muchas veces no es visible, pues se encuentra en el interior del dispositivo. En los dispositivos que llevan antena externa, puede tener el aspecto de la figura 4.22. En ambos casos hace falta un controlador, tarjeta o circuitería, que lo soporte.

Actualmente, la mayor parte de los computadores están conectados a través de una red que se extiende por todo el planeta, a la cual se le denomina Internet. Para comunicar de forma eficiente millones de computadores hay dos dispositivos periféricos de red fundamentales:

- *Switches* (conmutadores). Comunican de forma eficiente equipos cercanos, típicamente en la misma sala o edificio. La figura 4.24 muestra un típico *switch* para uso doméstico.
- *Routers* (enrutadores). Permiten comunicar computadores distantes, moviendo la información por las rutas adecuadas, dentro de la red.

Existen también dispositivos híbridos, que pueden realizar ambas funciones. Por ejemplo, un router doméstico que permite conectar la red local de una casa con internet, a menudo tiene también funciones de *switch* para permitir que se conecten a él varios ordenadores del ámbito doméstico.

Aunque se explica con más detalle en el capítulo de Sistema Operativos, comentar que cada computador conectado a Internet tiene una dirección, denominada dirección IP (IP viene de **I**nternet **P**rotocol), que sirve para identificarlo. Por ejemplo, la dirección 156.35.94.131. Puesto que es más

Figura 4.21: Conector Ethernet

Figura 4.22: Antena WiFi externa

Figura 4.23: *Switch* para uso doméstico

Figura 4.24: Controladores/adaptadores de red Ethernet y Wifi

fácil recordar nombres que secuencias de números, las direcciones IP tienen nombres asociados. Por ejemplo, www.uniovi.es. Para convertir nombres en direcciones IP se emplean computadores que actúan como **servidores de nombres**. Por ejemplo, uno de los servidores de nombres de la Universidad de Oviedo es el equipo con dirección IP 156.35.14.2.

4.4. Tipos de software

El computador es una máquina que permite ejecutar varias aplicaciones (programas) simultáneamente. Por ejemplo, un procesador de textos, cliente web, etc. La ejecución de las aplicaciones directamente sobre el hardware del computador no es adecuada, pues aparecen varios problemas:

- ¿Cómo puede ejecutar el computador varios programas a la vez sin que interfieran unos con otros?
- ¿Cómo evita el programador de una aplicación el tener que distribuir miles de variantes de su aplicación, debido a las variaciones existentes en el hardware del computador?
- ¿Cómo puede tratar un programador con los detalles de cualquier dispositivo del computador? Por ejemplo, para escribir un archivo en un disco duro SATA, debería programar su interfaz, lo que requeriría el manejo de miles de páginas de manuales.

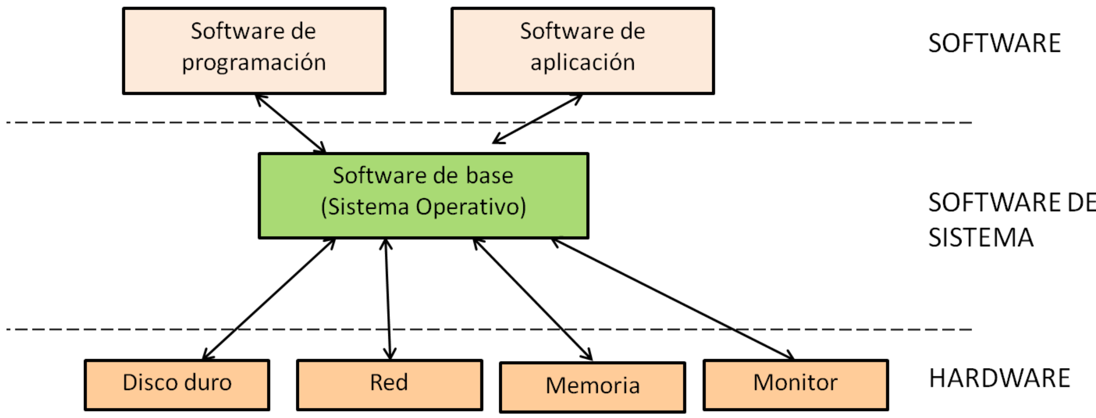


Figura 4.25: El sistema operativo en relación a las aplicaciones y al hardware

La solución a los problemas anteriores es el empleo de un programa especial denominado sistema operativo. El sistema operativo gestiona todo el hardware del computador.

La figura 4.25 muestra la relación entre el Sistema Operativo, las diferentes aplicaciones, y el hardware de la máquina. Como se puede ver, el Operativo es quien tiene acceso directo al hardware, mientras que las aplicaciones no. Esto implica que, si una aplicación necesita por ejemplo dibujar (o escribir) algo en la pantalla, en lugar de incluir las instrucciones específicas para acceder al hardware del interfaz de vídeo y escribir allí la información, lo que contiene son llamadas a funciones del Sistema Operativo, a las que les pasa qué quiere mostrar y dónde, y es el Operativo el que se ocupa del "hablar" con el interfaz apropiado.

Esto libera al programador de aplicaciones de tener que conocer los detalles de funcionamiento del hardware, y hace posible que una misma aplicación pueda ejecutarse sobre diferentes configuraciones de hardware. A la vez, el operativo puede orquestar los accesos al hardware solicitados por diferentes aplicaciones, de modo que no se interfieran unos con otros (por ejemplo, en el caso de la pantalla, destinando una "ventana" diferente para cada aplicación de modo que cada una sólo pueda solicitar escribir en su propia ventana).

Como vemos, existen diferentes **tipos de software**:

- **Software de aplicación**: es el que el usuario final utiliza para realizar su trabajo o su entretenimiento. Por ejemplo, en esta categoría tendríamos:
    - Aplicaciones Ofimáticas (procesadores de texto, hojas de cálculo, aplicaciones de presentación de diapositivas, etc.)
    - Sistemas Gestores de Bases de Datos (dedicaremos un capítulo posterior a estudiar qué función desempeñan este tipo de aplicaciones).
    - Programas para comunicación en red (navegadores web, gestores de correo, mensajería instantánea o chat, descarga de ficheros, etc.)
    - Aplicaciones de Control y Automatización industrial. Son programas diseñados para controlar un proceso industrial específico, tomando medidas de diferentes variables del proceso y mostrándolas en pantalla de una forma comprensible para el operador, posiblemente activando alarmas cuando alguna se sale de los rangos deseables, o incluso tomando acciones de control para corregirlas.
    - Software Educativo. Permiten utilizar el ordenador como herramienta de aprendizaje.
    - Software médico.

- Software empresarial
- Software de cálculo numérico
- Software de diseño asistido (CAD)
- Software de control numérico (CAM), que permite materializar los diseños realizados por CAD.
- Videojuegos, que permiten el uso del computador como plataforma de ocio.

- **Software de Sistema**: el que permite hacer funcionar el sistema y que otras aplicaciones puedan ejecutarse (y crearse) sobre él. Comprende:
  - El Sistema Operativo, cuyas funciones hemos expuesto someramente en párrafos anteriores, y al cual dedicaremos un capítulo completo más adelante.
  - Los Controladores de Dispositivo. Se trata de software que se integra con el Sistema Operativo, permitiendo a éste interactuar con diferentes componentes del hardware. En muchas ocasiones este software lo escribe el propio fabricante de cada componente hardware.
  - Utilidades. Realizan diversas funciones para resolver problemas específicos (por ejemplo, localización y reparación de fallos en el disco duro, eliminación de software no deseado, etc.) Muchas de estas herramientas se incluyen al instalar un Sistema Operativo, pero otras pueden ser desarrolladas por terceros.
  - Herramientas de desarrollo. Permiten la creación del propio sistema operativo, de los controladores, de las aplicaciones, de los programas utilitarios, etc. Entre estas herramientas se cuentan los compiladores, ensambladores, intérpretes de diferentes lenguajes, depuradores, etc.

4.5. Tipos de sistemas informáticos y sus ámbitos de aplicación

En muchas ocasiones identificamos el computador con un PC, ya sea de sobremesa o portátil. Sin embargo, hay otros muchos tipos de computadores. Prácticamente todos los dispositivos electrónicos actuales incorporan algún tipo de computador con características específicas. Por ejemplo: teléfonos móviles (reducido consumo y tamaño), sistemas de control industriales (robustez), electrodomésticos (pequeños y de bajo coste).

Cuando el computador está integrado en un dispositivo que sirve a un propósito concreto (por ejemplo, una consola de videojuegos, un simulador de vuelo para entrenamiento de pilotos, el sistema computerizado de un automóvil, un navegador GPS, etc.), se dice que es un computador de propósito específico. Si por el contrario el computador está abierto a cualquier posible uso, mediante la carga de los programas adecuados, se habla de un computador de propósito general.

Hoy día, los computadores de propósito específico no son diferentes desde el punto de vista de la arquitectura de los de propósito general. Se componen de CPU, memoria, dispositivos de entrada/salida. Lo que les hace específicos es que generalmente vienen con unos programas precargados que no se pueden sustituir por otros (o no fácilmente) y que sus interfaces de entrada/salida están específicamente diseñados para un uso concreto. Es habitual que su memoria y capacidad de cómputo estén limitadas y sean las estrictamente necesarias para poder ejecutar los programas específicos que les dan su funcionalidad.

Los computadores de propósito general son los que habitualmente conocemos. Sus capacidades de programación y potencia suelen ser mucho mayores que las de los computadores para aplicaciones específicas. Los ordenadores de propósito general podemos dividirlos a grandes rasgos en dos tipos (ver figuras 4.26 y 4.27):

Figura 4.26: Computador personal

Figura 4.27: Servidor

1. Ordenadores personales. Los ordenadores personales se emplean en el ámbito doméstico, o como medio de conexión a los servidores en el ámbito profesional. Se puede indicar que los ordenadores personales habitualmente se encuentran en dos formatos: sobremesa y portátil.

2. Servidores. Los servidores tienen muchas más capacidades: disco, memoria, CPU, etc. En el caso de los servidores, lo habitual es usar sistemas en rack que permiten una fácil sustitución y ampliación de las características del sistema, así como tolerancia a fallos. La parte mecánica y de refrigeración es importante en el caso de los servidores, puesto que está previsto que estén funcionando permanentemente, 24 horas al día durante todos los días del año. Los servidores gestionan la información de empresas y organizaciones: bancos, hospitales, universidades, etc.

Parte V

Introducción a los sistemas operativos

CAPÍTULO 5

Introducción a los Sistemas Operativos

5.1. Concepto y funciones que desempeña un sistema operativo

Los conceptos que se estudiarán en este capítulo se pueden extraer de la lectura del libro [1], y en menor medida, de [2].

Antes de nada, es conveniente definir una serie de conceptos adicionales que serán de utilidad en el resto del capítulo.

software de sistema es el conjunto de programas que realizan tareas comunes al computador, incluyendo el software de control y las utilidades.

software de control programas que gestionan el correcto funcionamiento del computador. Incluye el sistema operativo, el lenguaje de control y el software de diagnóstico y mantenimiento.

sistema operativo es el software que controla la ejecución de los programas de aplicación. Actúa como interfaz entre las aplicaciones del usuario y el hardware de la máquina.

interprete de lenguaje de control es un programa o conjuntos de programas que permiten al usuario introducir órdenes para su proceso por el sistema. Por ejemplo, desde una consola de comandos (en Windows, el programa cmd) se puede introducir el comando *dir* para visualizar el contenido de un directorio o carpeta del sistema de archivos.

software de diagnóstico y mantenimiento son los programas que utilizan las personas que se responsabilizan del buen funcionamiento tanto del hardware como del software. Estas personas se llaman **administradores**. Ejemplos de estas tareas: crear un usuario, limitar la cuota de disco de un usuario, actualizar programas, instalar y mantener el antivirus, etc.

utilidades programas que permiten realizar tareas de gestión y administración de un computador. Por ejemplo, las utilidades de análisis de discos, de configuración de dispositivos de red, etc.

software de aplicaciones es el conjunto de programas que realizan tareas concretas objetivo último por el que un usuario utiliza un computador. Ejemplos: programa de edición gráfica, navegador web, etc.

proceso es un programa en ejecución en un computador en un instante dado. Los sistemas operativos actuales permiten multitarea, esto es, el tiempo de cómputo de la CPU se reparte entre los procesos lanzados en una máquina.

sistema computarizado es el conjunto de computador (es decir, hardware) y software (incluyendo el sistema operativo y todas las aplicaciones necesarias) que está instalado en aquél.

En adelante, los términos máquina, computador y ordenador se utilizarán indistintamente.

5.1.1. Estructura de un sistema computarizado

En temas anteriores se ha estudiado cómo es la arquitectura de un computador. Sin embargo, y como se ha visto, un computador contiene tanto un hardware como un software. ¿Cómo se integra esto?

En la figura 5.1 se intenta representar el conjunto hardware y software de un computador. El harware representa todo elemento electrónico que, interconectado, es capaz de interactuar conjuntamente.

Así, si de un computador de usuario se trata, en el hardware se tendrá la placa madre conteniendo las CPU's, la memoria RAM, la memoria ROM, los controladores de los dispositivos de entrada/salida, etc. Entre los dispositivos de entrada/salida se tienen los discos duros, las unidades USB de memoria, etc.

La ROM de la placa madre es una parte básica del ordenador. En esta ROM reside el **programa de inicialización** del ordenador y la librería de funciones para acceso a la entrada y salida de la máquina. En algunos ordenadores, esta ROM es lo que se conoce como **BIOS** (Basic Input/Output System) y es suministrada con la placa madre. El programa de inicialización es el que realiza los primeros pasos para poner en funcionamiento el Sistema Operativo, y es propio de cada placa. *En realidad, este programa de inicialización se conoce como programa de arranque, pero lo denominaremos de inicialización para evitar confundirlo con el programa de arranque del sistema operativo.*

Y, ¿qué hace el hardware? Pues ejecutar el sistema operativo y tantos procesos como se le haya indicado. El sistema operativo es el conjunto de programas que actúa de interfaz entre el hardware y el exterior (incluyendo en el exterior al usuario que lo explota): toda acción que el computador debe ejecutar se le debe indicar al sistema operativo.

El sistema operativo se compone, entre otros, de un **núcleo**; éste es un programa que se ejecuta en memoria de la máquina responsable de atender las acciones y tareas del sistema computarizado.

¿Cómo se comunica el núcleo con los dispositivos de entrada/salida? Pues utilizando la BIOS o los **drivers** específicos de una tarjeta electrónica concreta.

Un driver es un conjunto de funciones para configurar el dispositivo y para intercambiar datos entre éste y el sistema computarizado.

Los sistemas operativos actuales permiten la carga y activación o desactivación de módulos o drivers para control de dispositivos. Es por ello que cuando se instala un nuevo dispositivo en la máquina (e.g., un nuevo lápiz de memoria USB), el sistema operativo busca el driver adecuado para interactuar con él, cargándolo y activándolo. Esta tarea de descubrir automáticamente hardware es tarea del sistema operativo.

Por encima de todo esto tenemos las utilidades de sistema operativo y de diagnóstico y mantenimiento, las cuáles pueden interactuar con el usuario, así como las aplicaciones de usuario, es decir, los programas que un usuario ha lanzado y que se están ejecutando en un computador en un momento dado. Ejemplos de aplicaciones de usuario: un navegador web, el Word©, etc.

5.1.2. Arranque de un sistema computarizado

Con este término se refiere a lo que ocurre en un computador desde que se pulsa el interruptor de encendido (o tras un *reset* o un reiniciado por fallo de alimentación, etc.) hasta que el sistema operativo toma el control de lo que ocurre en la máquina.

El proceso que ocurre se representa de forma esquemática en la figura 5.2. Tras alimentar con energía eléctrica el sistema, la máquina accede al programa que viene instalado en ella: este programa y el conjunto de rutinas y servicios que aporta este programa se denomina **firmware**. El firmare está almacenado en memoria no volátil (*ROM*, *EPROM* o *memoria Flash*, dependiendo del tipo de hardware).

El *firmware* es el responsable de: 1) realizar el test del hardware, de manera que se garantiza que los fallos son notificados apropiadamente, 2) accede al dispositivo de arranque, donde carga un nuevo programa: el **gestor de arranque** o (**bootloader**).

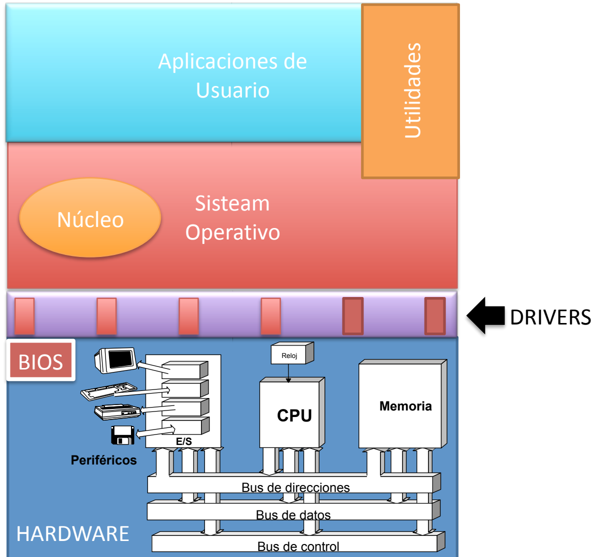


Figura 5.1: Estructura de un sistema computarizado. Por un lado, se dispone de hardware sobre el que se ejecuta el software. El sistema operativo es el software que permite explotar un hardware.

Figura 5.2: Proceso de arranque de un computador.

El *bootloader* es el responsable de localizar los posibles sistemas operativos disponibles, así como de acceder y cargar el núcleo del sistema operativo seleccionado. Una vez dispone de éste, el gestor de arranque finaliza y le pasa el control al núcleo para que el sistema operativo que el usuario ha seleccionado se inicie. Cuando se dispone de un solo núcleo de sistema operativo, entonces no se da opción de seleccionar, se arranca automáticamente con éste.

El *firmware* ha sido típicamente conocido por la **BIOS** del sistema. *BIOS* es el acrónimo de *Basic Input/Output System*. Originalmente fue el firmware de arranque de los ordenadores *IBM PC* (en arquitectura de 8 bits), el cuál fue posteriormente evolucionando para llegar a las mas actuales de 32 bits.

Actualmente, el *firmware* en los computadores sigue el estándar **UEFI**. Este término significa *Unified Extensible Firmware Interface*, y se pronuncia como las letras inglesas U-E-F-I. Fue desarrollado por *Intel* para favorecer el desarrollo de sistemas operativos en 64 y, además, dar garantías de servicio a las capas del sistema operativo.

Respecto a la *BIOS*, *UEFI* extiende la información útil tanto de la plataforma como acerca de los diferentes medios disponibles y de los posibles sistemas operativos. Igualmente, dispone de servicios de arranque y explotación sobre el hardware para su uso por parte del sistema operativo.

Es por ello que *UEFI* presenta diversas ventajas frente a *BIOS*:

- Capacidad de arranque desde discos de gran tamaño (mayores de 2 TB) usando *GPT*.
- Arquitectura y controladora de CPU independientes.
- Diseño modula debido a la posibilidad de drivers independientes de la CPU.
- Flexibilidad en el arranque de sistema operativo, disponiendo de servicios gráficos o de acceso al hardware previamente a la carga del sistema operativo.
- Acceso a los servicios de arranque y ejecución. El primero se refiere al acceso a gestión gráfica o de texto de los dispositivos de hardware, mientras que el segundo se refiere a la gestión de la fecha, la hora o el **NVRAM** (memoria de acceso aleatorio no volátil, o Non-volatile random access memory).

5.1.3. Funciones de un sistema operativo

Desde el punto de vista del hardware, el sistema operativo se encarga de la administración de los recursos de la máquina:

- Administración de procesos y de la CPU.
- Administración de memoria principal.
- Administración del sistema de ficheros.
- Administración de dispositivos periféricos.

Esto es, el hardware sólo se encarga de existir, el S.O. es el que lo gestiona y lo hace funcionar tal y como se espera que lo haga.

Desde el punto de vista del usuario, el sistema operativo ofrece un interfaz entre el usuario y el ordenador:

- Intérprete de comandos y/o Interfaz gráfica.
- Ejecución de aplicaciones.
- Acceso a dispositivos periféricos.
- Acceso controlado a ficheros.
- Utilidades relativas a la seguridad del sistema.

En las siguientes secciones se estudiarán cada una de las funciones del S.O.

5.2. Funciones que el sistema operativo presta a los programas

5.2.1. Administración de procesos

En los SSOO multitarea tenemos más de un proceso en ejecución de manera simultánea. De hecho, en una máquina es habitual que se ejecuten muchos más procesos que el número de CPU's disponibles. Y recordar que muchos computadores actuales tienen varias CPU's (o núcleos) integradas.

Por lo tanto, el tiempo de cómputo de una CPU es un recurso escaso por el que compiten los procesos en ejecución en una máquina. El responsable administrar el tiempo asignado a cada proceso es el S.O., en lo que se conoce como **planificación de la CPU**.

En los sistemas operativos modernos el S.O. asigna la CPU a cada proceso durante un muy breve lapso de tiempo (*quantum*). Sin embargo, realiza esto de forma tan rápida que para un usuario humano la sensación es que todos los procesos se ejecutan de manera simultánea.

Por otra parte, es el S.O. el responsable de ofrecer a los procesos (aplicaciones) un **entorno seguro de ejecución**, donde se garantiza que:

1. Un proceso no podrá interferir en la ejecución de otros procesos.
2. Un proceso no podrá acceder de forma incorrecta o descontrolada al hardware. Por ejemplo, en un S.O. multiusuario, con dos usuarios no adminitradores actualmente trabajando en la máquina:
   - un proceso de usuario no puede borrar los datos de otro usuario
   - un proceso de usuario no puede desinstalar programas.
   - un proceso de usuario no puede reconfigurar un dispositivo hardware.

Si se produce alguna de las situaciones anteriores el sistema operativo genera una excepción (error) con dos posibles acciones:

- Se da al proceso la oportunidad de corregir el error. Si no lo hace se termina el proceso.
- En casos graves el S.O. toma el control y finaliza al proceso, normalmente el S.O. registra este tipo de evento para control por parte del administrador.

5.2.2. Administración de memoria

¿Cuánta memoria RAM tiene instalada una máquina (memoria principal)? ¿Cuánto ocupa un programa? ¿Cuántos programas se ejecutan simultáneamente? ¿Qué pasa si la memoria principal de la máquina es menor que lo que ocupan los procesos en ejecución?

Todos los procesos deben residir total o parcialmente en memoria principal. Es decir, que no es necesario que todo el programa (que pueden ser muchos, muchos MB) esté cargado en memoria. Realmente, un programa se carga de bloque en bloque, cada bloque es un número de bytes importante, que contiene muchas instrucciones en lenguaje máquina, y que permite ejecutar durante un tiempo el programa. Cuando el bloque se ejecuta al completo, el bloque actual se desecha y se carga el siguiente bloque de programa.

Dado que la memoria principal de un computador es limitada, se han desarrollado técnicas para la gestión de memoria. Así, es ha desarrollado la *memoria virtual*, que permite extender la memoria principal con espacio en almacenamiento secundario por medio de recursos hardware (unidad de gestión de memoria) y software (rutinas y servicios del sistema operativo).

Además, se han implementado otras técnicas de gestión de memoria, como es la *paginación de memoria*. De forma simplificada, el proceso es el siguiente. Un programa se divide en páginas; el sistema operativo cargará las páginas que necesite para optimizar el rendimiento de la aplicación. Las páginas cargadas van a la memoria virtual: es decir, las que se ejecutarán inmediatamente irán a la memoria principal y el resto a memoria secundaria utilizada como memoria virtual. ¿Qué permite esta paginación? Por una parte, se pueden mantener más procesos en memoria. Por otra parte, mejroa la eficiencia de explotación de la máquina.

Aún con todo ello, en función de las características de la máquina y del número de procesos a ejecutar, es posible llegar a colapsar el sistema ante falta de memoria principal. Típicamente se puede observar esta situación ejecutando varios programas actuales que requieran mucha capacidad de computación (matlab, programas gráficos, etc.) en computadores antiguos (digamos, por poner un ejemplo, de más de 5 años).

5.2.3. Administración del sistema de ficheros

Existe una parte del S.O. que se encarga de administrar el sistema de ficheros en memoria secundaria. Los sistemas de archivos o ficheros es la forma de estructurar la información en una unidad de almacenamiento o memoria secundaria. Cada sistema operativo utiliza su propio sistema de archivos.

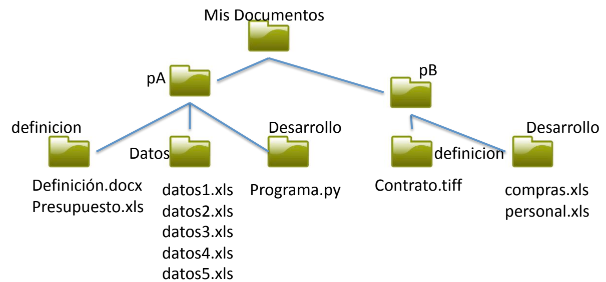


Figura 5.3: Ejemplo de jerarquía de directorios en un sistema de archivos.

Así, un sistema de archivos incluye la definición de cómo se organiza un disco, qué datos se guardan de cada uno de los directorios y archivos, cómo se almacena la información contenida en un archivo, etc.

Un sistema de archivos admite organizar los datos en una jerarquía de directorios y/o ficheros propios. Un archivo es el conjunto de datos relacionados con contenido relevante, p.e., un archivo o fichero de una imagen o una carta escrita con un editor de textos. Un directorio es totalmente equivalente a carpeta, un contenedor de elementos del sistema de achivos. Luego un directorio contiene tanto carpetas como archivos.

Todo sistema de archivos tiene un punto de montaje o de enlace que se conoce como raiz. En los sistemas unix se tiene como punto raiz un determinado bloque de disco que el sistema operativo denomina "/". En los sistemas windows, se dispone de un sistema de archivos por cada uno de los discos instalados en el sistema, asignándoles una letra, p.e. "C:\".

A partir de estos puntos de montaje se puede crear una jerarquía de directorios y subdirectorios, de manera que la información almacenada esté bien organizada.

Para ilustrar esto, pongamos un ejemplo. Sea un usuario que tiene 2 proyectos que llevar a cabo, los proyectos A y B. Para el proyecto A tiene información de definición del problema (un archivo de texto), datos extraídos de unas pruebas (5 archivos de excel), y un programa que ha desarrollado para su gestión. Por otro lado, del proyecto B tiene una imagen en formato tiff con el contrato escaneado, mas unas hojas de excel de gestión de compra de material, y un listado de personal que está trabajando en el proyecto.

Luego, una buena organización permite buscar información de cada proyecto de forma rápida. Lo más sencillo es crear un directorio para cada proyecto, con nombres que estén altamente relacionados con sus contenidos. De igual manera, para el proyecto A se debería crear un directorio para la definición del problema, otro para los datos, y otro para el programa realizado. Similarmente se procedería con el proyecto B, lo que generaría una jerarquía de directorios como la que se muestra en la figura 5.3.

Se conoce como **ruta** o **path** de un archivo el listado de directorios desde el raíz hasta donde se encuentra dicho archivo, todos ellos separados por un carácter específico del sistema operativo. Por ejemplo, si se tratase de Windows, la ruta del archivo "`Programa.py`" es "`MisDocumentos\Desarrollo\`".

Las rutas pueden ser relativas al directorio actual o absolutas desde el directorio raíz. En el caso de la figura 5.3, si suponemos que la carpeta "Mis Documentos" está en el disco "`C:\`", entonces la ruta absoluta es del archivo "`Programa.py`" es "`C:\MisDocumentos\Desarrollo\`". La ruta relativa del archivo "`Programa.py`", asumiendo que estoy visualizando el directorio "`MisDocumentos`", es "`.\Desarrollo\`".

El **nombre completo** de un archivo es la unión de la ruta absoluta y su nombre: "`C:\MisDocumentos\Desarrollo\Programa.py`"

Los sistemas operativos incluyen herramientas o utilidades para la gestión de los sistemas de

archivos, a modo de ejemplo:

- crear, copiar, mover y borrar archivos
- comprimir y descomprimir archivos
- desfragmentar las unidades de disco
- comprobación de estado
- formateado de unidad de disco (se crea un sistema de ficheros inicialmente vacío, donde solo existe el directorio raíz).

Desafortunadamente, como cada sistema operativo tiene su propio sistema de archivos, no se pueden intercambiar unidades de disco entre sistemas operativos diferentes sin más, es necesario utilizar herramientas específicas para ello.

5.2.4. Administración de dispositivos

El sistema operativo facilita el acceso a los dispositivos periféricos por parte de las aplicaciones, de manera que no es necesario conocer los aspectos técnicos del dispositivo y realizar con los dispositivos operaciones de lectura/escritura.

Cada dispositivo periférico necesita un software especial que funciona como interfaz con el S.O., conocido como driver o controlador de dispositivo.

El controlador de dispositivo es específico del hardware y del S.O. y debe instalarse en el ordenador para que el S.O. pueda utilizar el dispositivo. El controlador de dispositivo no forma parte del S.O. y debe proporcionarlo el fabricante del dispositivo.

5.3. Funciones que el sistema operativo presta a los usuarios

5.3.1. Interfaz usuario-ordenador

Los SSOO incorporan una aplicación de interfaz entre el usuario y el ordenador. El usuario realiza operaciones a través de esta aplicación:

- ejecución de aplicaciones de usuario
- manipulación de carpetas y ficheros
- acceso a dispositivos periféricos.

Los interfaces con los usuarios no forman parte de los SSOO, propiamente dicho, si bien todo S.O. incluye un tipo de interfaz con el usuario. A modo de ejemplo, para Linux se han desarrollado varios interfaces de escritorio, pudiendo el usuario elegir cuál utiliza en su máquina.

Los interfaces con el usuario pueden ser una **interfaz de texto**, mediante un intérprete de comandos de sistema operativo, o bien puede tratarse de **una interfaz gráfica** (también conocida como **GUI -interfaz gráfica de usuario**). En todos los sistemas operativos de usuario actuales se dispone de ambas; en Windows se dispone de *cmd* (*símbolo de sistema* o *ventana del DOS*), en UNIX se dispone de varias interfaces (*sh*, *csh*, *ksh*, *bash*, . . .).

Las terminales de comandos fueron los primeros interfaces incorporados en los SSOO (por ejemplo UNIX y MS-DOS). Esto es así debido a que el coste computacional es muy bajo, y en esos tiempos no se disponía de mucha potencia computacional. Actualmente están presentes en todos los SSOO modernos.

En estos interfaces, todas las órdenes deben realizarse mediante comandos de texto. Estos comandos son órdenes que el sistema operativo entiende perfectamente: nombres de programas, comandos de sistema, archivos ejecutables, etc.

En cuanto a los comandos y archivos ejecutables, la interfaz de comandos dispone de un lenguaje de programación para llevar a cabo tareas de forma secuencial y sin atención directa por parte del usuario (desatendida). Por ejemplo, el sistema operativo tiene un comando para listar el contenido de directorios. Si se desease buscar dónde se encuentran unos ficheros concretos en una jerarquía de archivos, sería posible generar un código de forma que deje como salida los archivos y su ruta.

A favor de los intérpretes de comandos se pude indicar que a) permite gestionar el sistema operativo con más profundidad dado que se tiene control absoluto sobre las opciones, b) permite realizar operaciones masivas y de forma desatendida, c) incorporan un lenguaje de programación para diseñar comandos más complejos, y d) requieren un menor consumo de recursos. Por contra, es necesario un mayor conocimiento técnico del S.O. y resultan poco intuitivos.

En cuanto a los interfaces gráficos se puede decir que son una evolución del intérprete de comandos, facilitando el acceso y la interacción del usuario mediante objetos gráficos como ventanas, botones, iconos enlazados, imágenes, etc. Requieren pocos conocimientos técnicos al ser, por regla general, muy intuitivos. Por contra, en muchos casos limitan la operatividad y las posibilidades de configuración al estar pensados para las labores más usuales y rutinarias en un computador. Claramente, al manejar recursos gráficos requieren mayor consumo de recursos de la máquina. Finalmente, son poco aconsejables para operaciones masivas y permiten directamente realizar operaciones desatendidas.

5.3.2. Acceso a las Redes de Computadores

Entenderemos por redes de computadores todos los elementos hardware y software que permiten que los computadores intercambien datos de cualquier índole. Las redes de computadores se pueden clasificar de muchas maneras; en este tema no se dará una clasificación exhaustiva sino que se mostrará una clasificación basada en el alcance de la red. Seguidamente se explicará el modelo de red TCP/IP de forma muy abreviada. Ffinalmente, la configuración del mismo para el sistema operativo Windows 8.

Atendiendo al alcance, es decir, la distancia máxima de conexión, podemos clasificar las redes como:

Personal Area Network (PAN's) tienen como objetivo comunicar computadores con periféricos por lo que se trata de redes de muy limitado alcance y con cierta limitación en el ancho de banda. Ejemplos de estas redes: BlueTooth, ANT+.

Local Area Network (LAN) conecta computadores dentro de un área de trabajo. En cuanto a alcance, tradicionalmente se asumen distancias de cientos de metros; aunque con la tecnología actual las distancias realmente no están tan acotadas. Ejemplos de estas redes: Ethernet, Zigbee, la familia IEEE 802, etc.

Wide Area Network (WAN) enlaza tanto LAN's como otras WAN's para interconectar redes mediante infraestructura privada o pública. En este caso, podemos citar redes como ATM, X.25, etc.

Existen muchos posibles actores en una red pero sólo se detallarán los más relevantes para poder entender de forma sencilla el funcionamiento de las mismas . Los principales actores que intervienen en una red son (ver figura 5.4):

Computador: elemento a interconectar, incluyendo tanto computadores personales, servidores, etc.

Punto de Acceso o Access Point (AP): se refiere al elemento que dota de red local inalámbrica a un área determinada.

Figura 5.4: Esquema de una red inalámbrica doméstica con acceso a Internet.

Puerta de enlace o Gateway: es el dispositivo que permite el paso de una red a otra. Si es a nivel de hardware hablamos de puentes (bridge), si es a niveles de red superior hablamos de routers, etc.

Enrutador o Router: permite la inteconexión entre redes.

La red WiFi doméstica

Para intentar que se entienda todo esto, usaremos un ejemplo que hoy en día es muy común: el acceso a Internet en el hogar mediante un router WiFi (llamémosle rWF). Los rWF's actuales suelen disponer de unos puertos para conectar computadores mediante el estándar IEEE 802.3 (Ethernet); para ello se utiliza el cable con de 8 hilos con conector RJ45. Además, funciona como punto de acceso a la red local inalámbrica (WiFi) según el estándar IEEE 802.11. Esto significa que ese dispositivo rWF actúa de puente entre los puertos RJ45 y el acceso inalámbrico, pero todos ellos formarán una red local: todos ellos se podrán ver e interactuar entre sí. Finalmente, para que todos los ordenadores puedan acceder a Internet, los rWF incluyen: i) la función de enrutado, que permite el intercambio de datos entre computadores pertenecientes a diferentes redes, y ii) un puerto WAN específico para conectarse al proveedor -Telecable, MoviStar, Orange, ...-.

Actualmente, el modelo de red por excelencia es el modelo TCP/IP (es el que utiliza Internet). Este modelo está basado en capas: la capa 1 de acceso al medio físico y enlace, la capa 2 de red, la capa 3 de transporte y la capa 4 de aplicación. La capa 1 es la que define cómo se conectan ordenadores a una red local, con posibles diferentes tecnologías como puede ser IEEE 802.3 e IEEE 802.11.

De forma simple, en el modelo TCP/IP cada ordenador tiene un identificador único o dirección de red -conocida como *dirección IP de la máquina*-. Este identificador es un entero de 32 bits agrupado en 4 grupos de 8 bits. Como $2^8 = 256$, con cada grupo de 8 bits podemos tener 256 valores diferentes, de 0 a 255. Por todo ello, las direcciones IP se indican con esos 4 valores, por ejemplo, 156.35.33.105 de www.uniovi.es.

Todos los ordenadores de una red local comparten los bits más significativos de la dirección de red de 32 bits. Para conocer esta parte común se utiliza la *máscara de red*, otros 32 bits expresados

de forma similar a la dirección IP, que permiten saber la dirección de la red local: si se hace al *and lógico* entre la dirección IP y la máscara, el resultado es la parte común. Consecuentemente, en la máscara de red se ponen a uno todos los bits que son comunes a la dirección de la red, el resto a 0. Un ejemplo de las operaciones realizadas se visualiza en la tabla 5.1. *Tanto la dirección de red como la máscara de red es un dato que el administrador de red nos indicará si tenemos que especificar dicha información a nuestro ordenador para conectarnos en red.*

	IP en decimal	IP en binario
Dirección de red	156.35.33.105	10011100 00100011 00100001 01101001
Máscara de red	255.255.255.0	11111111 11111111 11111111 00000000
Resultado del AND	156.35.33.0	10011100 00100011 00100001 00000000

Tabla 5.1: Uso de la máscara para determinar la dirección de la red.

Para conectar ordenadores entre diferentes redes se utiliza la capa 2 del modelo TCP/IP, mientras que la capa 3 de transporte permite disponer de puntos de acceso de red a las aplicaciones o programas. Cada punto de acceso a la red que proporcional la cada de transporte está formado por una dirección de red mas un entero de 16 bits denominado *puerto*. Un puerto que es probable todos conozcan es el 80, puerto estándar para el servicio HTTP.

Finalmente, está la capa de aplicación que especifica cómo deben comunicarse los programas. Por ejemplo, el protocolo HTTP especifica como intercambian información un servidor web y un cliente web. Un servicio imprescindible es el servicio de nombres en Internet. Este servicio permite asignar nombres -según una notación jerárquica- a máquinas o servicios, con lo que no es necesario conocer la dirección de red sino un nombre, por ejemplo www.uniovi.es. Las máquinas siguen teniendo la dirección de red, pero se les pone como apodo un nombre más fácil de recordar por el ser humano. Este servicio de nombres es el conocido como *Servicios de nombres de dominio, DNS*.

Resumiendo, la información que un ordenador debe tener para poder conectarse a internet es:

1. Dirección de red IP,
2. Máscara de Red,
3. Dirección de red del gateway o puerta de enlace,
4. Dirección o direcciones de red del/de los servidores DNS.

Estos datos los debe aportar el administrador de red. Es posible que se ajusten automáticamente, lo que hoy por hoy es lo más común, mediante el protocolo conocido como *DHCP, Dynamic Host Configuration Protocol*. El administrador de la red puede indicar que los datos de dirección de red, máscara y gateway se adquieren dinámicamente -especificando los servidores de nombres- o que todos los datos se adquieren mediante DHCP, dinámicamente.

En las sucesivas imágenes se verá cómo se configura la red en Windows 8. Hay que acceder al *Panel de Control*, y de ahí, al *Centro de Redes y Recursos Compartidos* de Windows.

5.3.3. Aplicaciones relativas a la seguridad del sistema

No forman parte del sistema operativo propiamente dicho, sino que se trata de utilidades desarrolladas por otros que se integran en el sistema operativo. Es habitual que en SSOO modernos se incorporen este tipo de utilidades de serie.

Algunas de estas herramientas son:

Antivirus es una aplicación diseñada para combatir y evitar de forma activa la infección del ordenador por un virus informático.

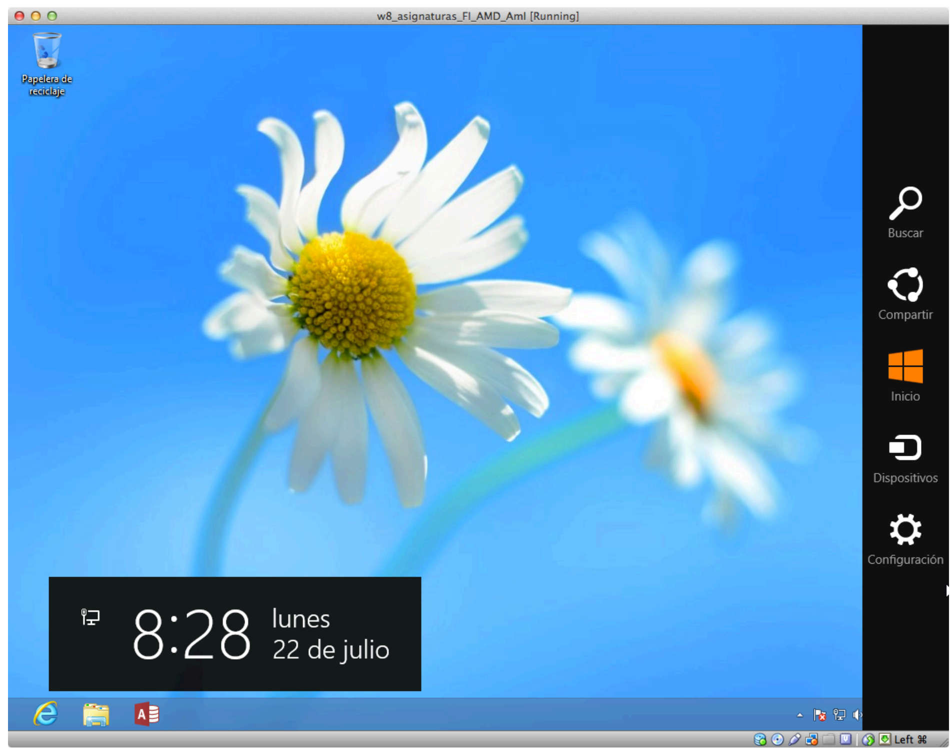


Figura 5.5: Paso 1.- Acceso a Configuración desde el escritorio.

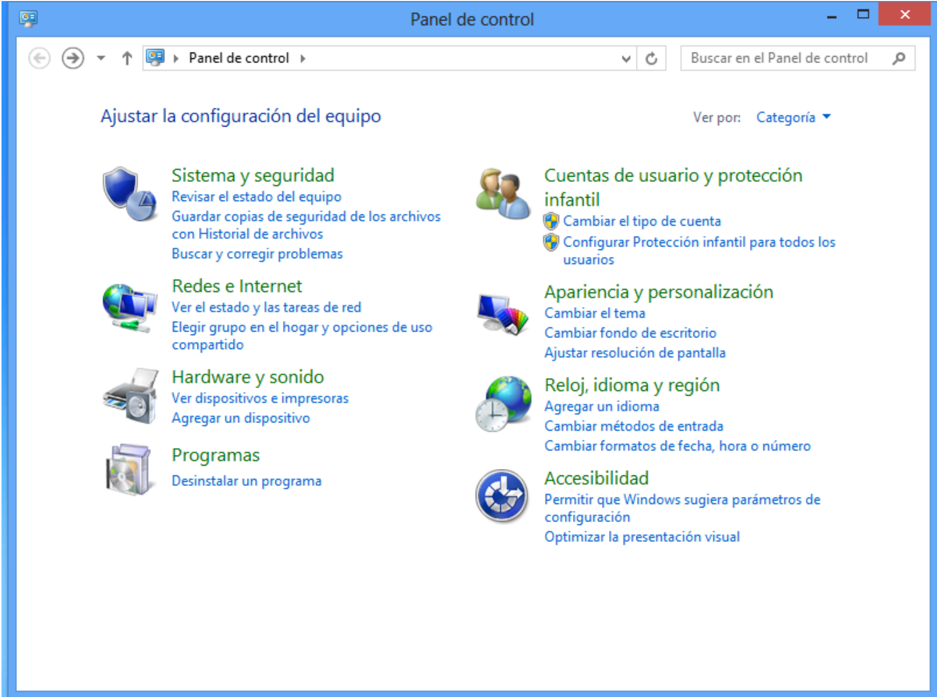


Figura 5.6: Paso 2.- Acceso al Panel de Control.

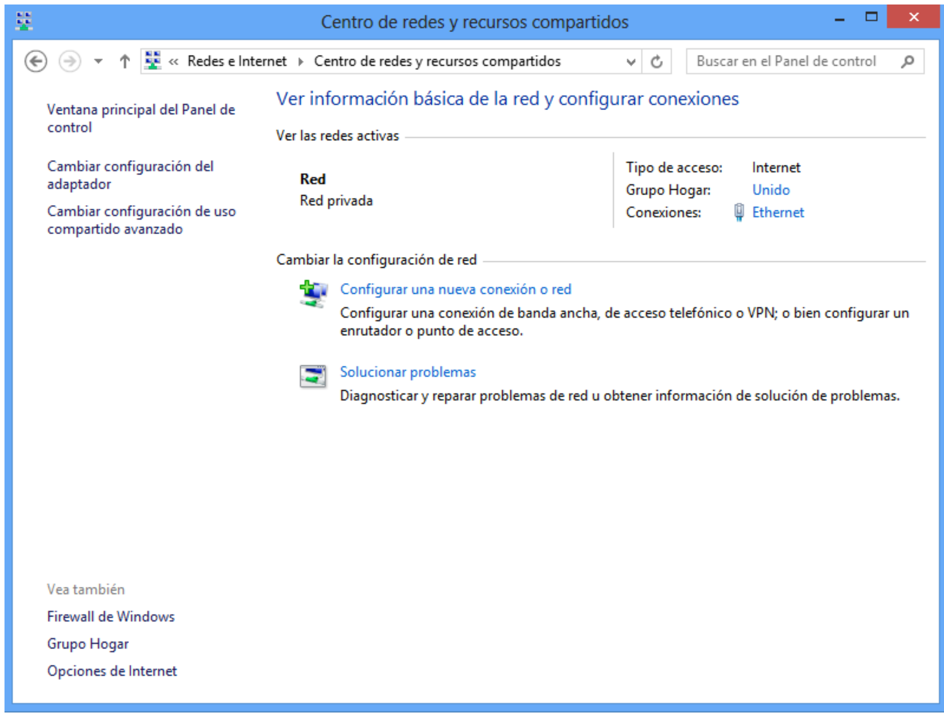


Figura 5.7: Paso 3.- Acceso al Centro de Redes y Recursos Compartidos.

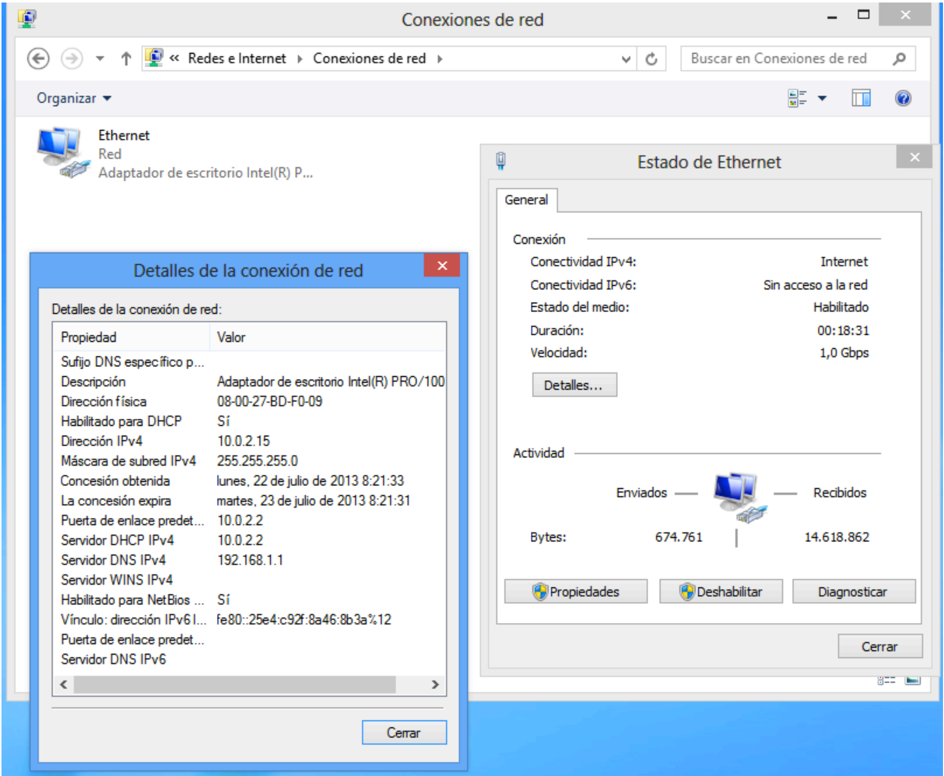


Figura 5.8: Paso 4.- Acceder a las propiedades de la conexión. También se muestra el cuadro de diálogo de detalles con la información de la conexión de red de la interfaz de red -tarjeta de red-.

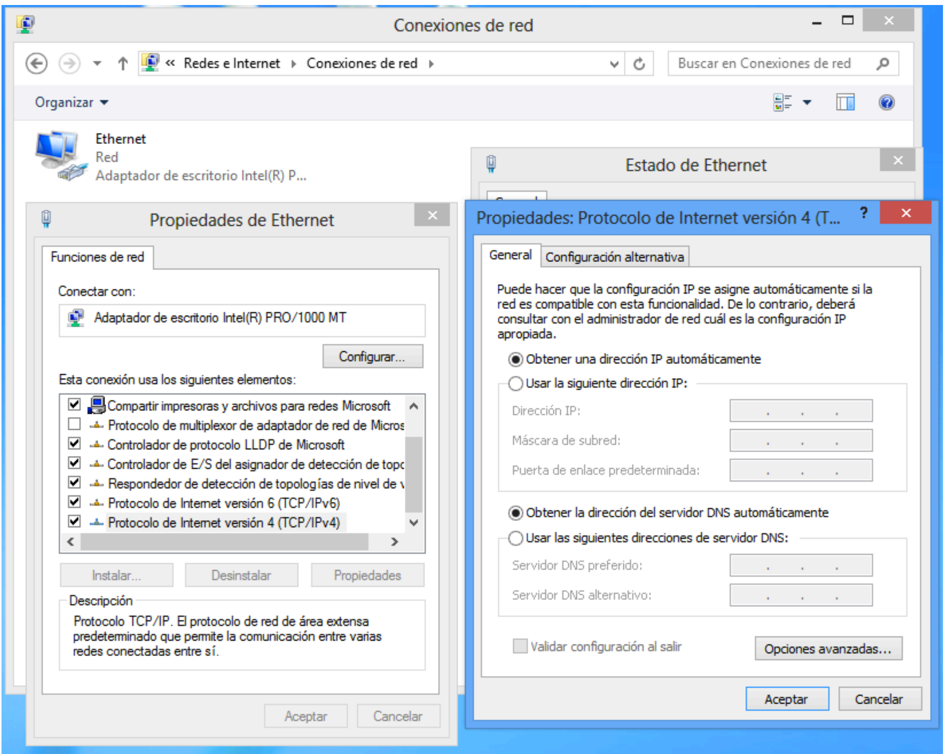


Figura 5.9: Paso 5.- Acceso a las propiedades de TCP/IP y configuración de los diferentes datos dados por el administrador de red.

Un virus informático es un software malintencionado que altera el funcionamiento normal del ordenador. Aunque existe la creencia que existe algún S.O. al que no atacan los virus, en la realidad todos los SSOO son susceptibles de tener virus: ¡solo tiene que haber personas que realicen los programas adecuadamente!

Un virus se camufla en ficheros ejecutables e incluso en el código de arranque del sistema operativo, y suele modificar otros ejecutables para que incluyan una copia del propio virus.

Típicas formas de transmisión de virus es mediante el intercambio de ficheros infectados en el disco o lápiz USB, correo electrónico, acceso a páginas web, etc.

Un virus se activa por primera vez al ejecutar o al acceder a un fichero infectado. Al activarse el virus queda residente y camuflado en memoria principal. Paulatinamente, toma el control de los servicios básicos del S.O. y se propaga infectando otros ficheros. Posteriormente suele instalarse en el código de arranque del ordenador para tomar el control al encender el equipo.

Un software antivirus dispone de una base de datos con todos los virus conocidos. Permanece residente en memoria y chequea todos los ficheros abiertos por el S.O. buscando trazas de algún virus conocido. Es por ello que es de importancia vital actualizar la base de datos de virus diariamente, básicamente aprovechando que la mayor parte de los antivirus se actualizan automáticamente a través de internet.

Anti-spyware es una aplicación diseñada para combatir de forma activa la infección del ordenador por un programa espía de manera similar a un antivirus.

Un spyware o spy es un software malintencionado que recopila información de las actividades del usuario. Su objetivo principal es recopilar información de los hábitos y gustos del usuario para enviarlo a empresas publicitarias.

Existen programas espía no malintencionados, como puede ser la información de usuario recopilada y enviada por software legal instalado en la máquina (p.e., el paquete informático Office).

La información que suele recopilarse incluye datos de mensajes y contactos de correo electrónico, dirección IP, páginas web visitadas, software instalado, descargas realizadas, etc.

Al igual que los antivirus, los anti-spyware dispone de una base de datos con programas espía conocidos que conviene mantener actualizada. Su funcionamiento se basa en bloquear el envío de datos confidenciales a través de internet (Ej: datos personales), bloquear paneles emergentes de publicidad no autorizados (anti-adware), etc.

Cortafuegos es una aplicación que controla el acceso de otros computadores a la máquina actual, así como la capacidad de las aplicaciones instaladas en ésta para comunicarse con el exterior a través de la red (habitualmente Internet).

Las aplicaciones de red pueden iniciar la comunicación en dos sentidos:

1. *Conexiones salientes*.- Nuestra aplicación solicita a otro ordenador de la red que le envíe cierta información (Ej: Navegador web).
2. *Conexiones entrantes*.- Otro ordenador de la red solicita a una aplicación de nuestro ordenador que le envíe cierta información (Ej: escritorio remoto, aplicación P2P, juegos).

Salvo las procedentes de programas spy-ware, las conexiones salientes no representan un problema de seguridad que no esté controlado por el usuario (p.e., evitar acceder a sitios web de seguridad comprometida, no abrir correos de procedencia desconocida o sospechosa, etc.).

Sin embargo, las conexiones entrantes son potencialmente peligrosas por varios motivos. En primer lugar, pueden acceder a nuestro ordenador para coger información. Por otra parte, pueden controlar remotamente el ordenador (escritorio remoto).

El cortafuegos o **firewall** controla de forma activa las conexiones de red realizadas en nuestro ordenador. Al igual que otras aplicaciones de seguridad permanece residente en memoria principal.

Su misión fundamental es la de *notificar al usuario* intentos de conexión de aplicaciones de red desconocidas. Una vez que el usuario decide si permite o no la conexión se memoriza la respuesta creando una regla. Algunos cortafuegos solo notifican conexiones de entrada y no de salida (Ej.: Firewall de windows).

5.3.4. Herramientas para el mantenimiento del Sistema Operativo

El uso habitual de un ordenador sin realizar ninguna tarea de mantenimiento periódico conduce en la mayoría de ocasiones a una *degradación del sistema operativo* y como consecuencia de ello a un menor rendimiento del mismo.

Existen varias causas que pueden originar esta degradación tales como la instalación y desinstalación de programas, la navegación en Internet e incluso en trabajo diario con archivos. Estas actividades que realizamos en nuestros equipos generan entre otros, archivos temporales, cookies, *logs* de acceso, fragmentaciones de disco, etc.

Para el mantenimiento de nuestro ordenador se pueden utilizar programas específicos de terceros o aplicaciones del propio sistema operativo. En este capítulo mostraremos algunas de las herramientas que incorporan los sistemas operativos Microsoft Windows y Apple OS X. Las versiones utilizadas son Windows 8 y OS X Mavericks. Si el alumno tiene alguna versión anterior, la forma de acceder a las herramientas de los sistemas operativos, las opciones disponibles en cada una de ellas y las imágenes mostradas podrían variar ligeramente.

En cualquier caso, se pueden dar unas recomendaciones generales para que nuestro ordenador se mantenga en condiciones adecuadas de utilización:

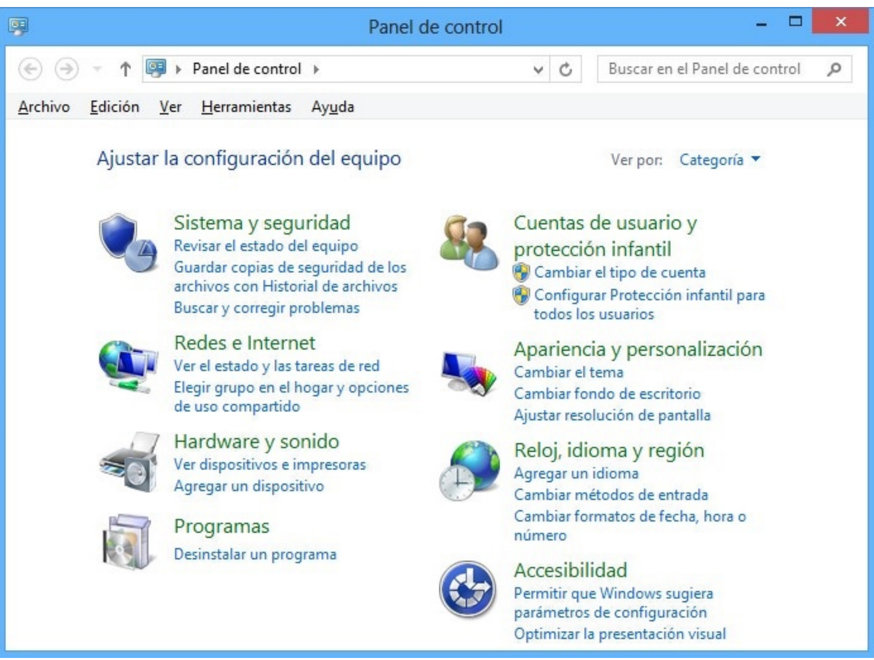


Figura 5.10: Panel de Control de Microsoft Windows 8

- Mantener el Software actualizado
    - Sistema Operativo
    - Programas de usuario
    - *Drivers*
    - Antivirus
- Mantener optimizados los discos
    - Liberando espacio en disco
    - Desfragmentando los discos
- Realizar copias de seguridad periódicas

Actualización de software

Todos los programas pueden contener fallos o problemas de seguridad que los fabricantes van detectando y corrigiendo a lo largo del tiempo. Por tanto, es muy importante disponer de la última versión de los mismos. En la actualidad, tanto los sistemas operativos como otras aplicaciones incorporan mecanismos para su actualización automática. En el caso de Microsoft Windows esta característica, y otras que comentaremos más adelante, se puede encontrar a través del *Panel de Control* (figura 5.10), categoría *Sistema y seguridad* y se llama *Windows Update* (figura 5.11). Se puede configurar para que se realicen automáticamente o que el proceso sea controlado por el usuario.

También OS X permite realizar las actualizaciones de manera automática o manual. Para acceder a la configuración de esta característica tendremos que acceder a la opción de *Preferencias* del *Menú General de OS X* (figura 5.12) y luego la opción *App Store* (figura 5.13).

Para la actualización de los drivers (controladores de dispositivo) en Windows se puede acceder a la utilidad de varias maneras: a través de *Panel de Control* en la categoría *Sistema*, o a través de la *Administración de Equipos*, tal como se muestra en la figura 5.14. En OS X este proceso se realiza automáticamente.

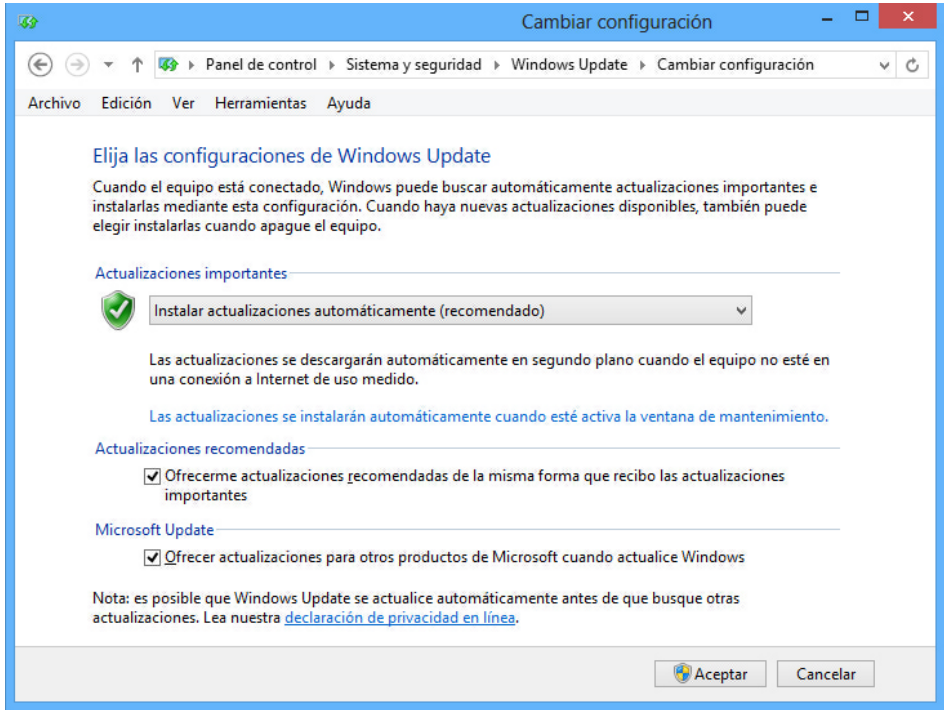


Figura 5.11: Actualización de Software en Windows 8

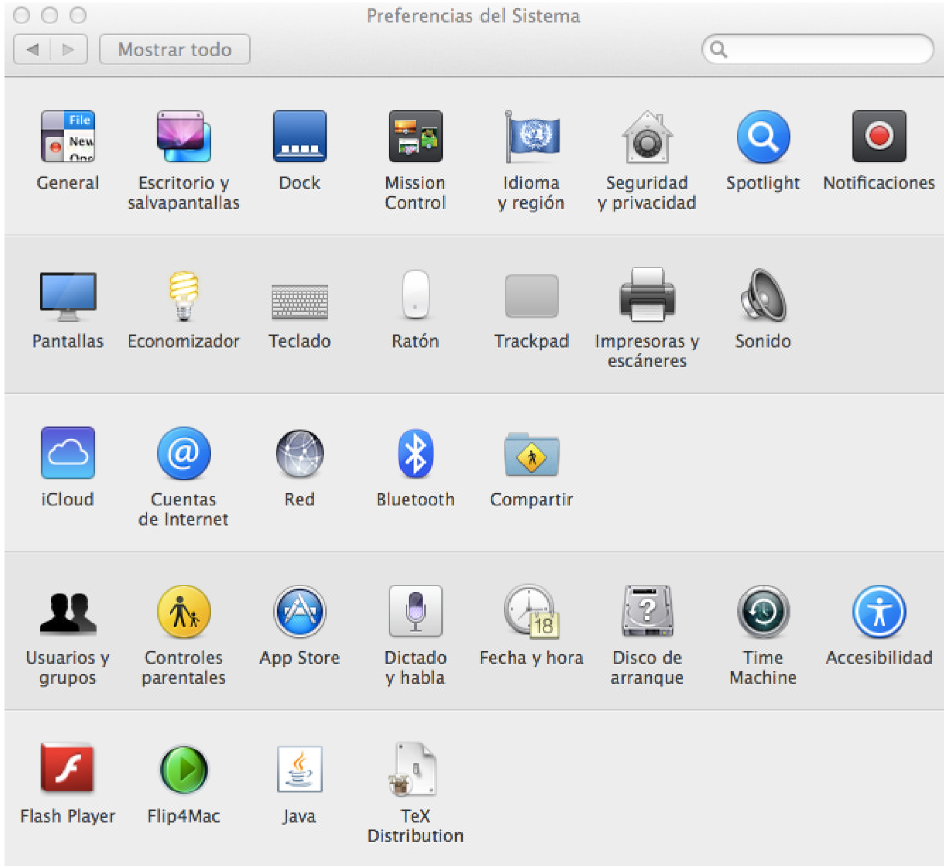


Figura 5.12: Preferencias del Sistema en Mac OS X

Figura 5.13: Configuración de las actualizaciones de software en Mac OS X

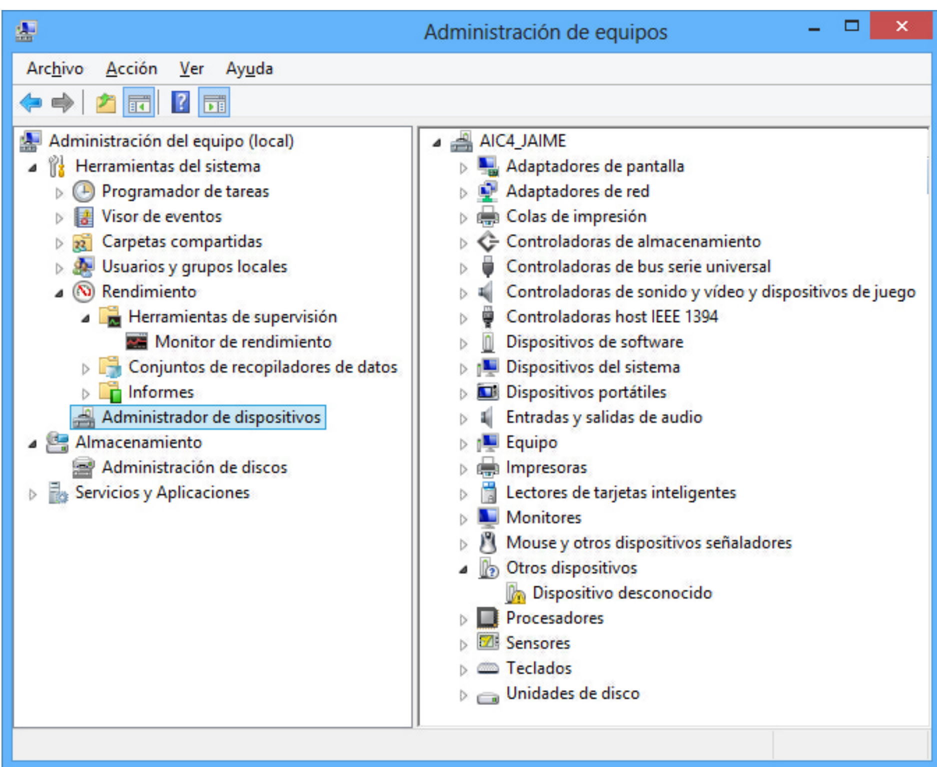


Figura 5.14: Acceso a la administración de dispositivos en Windows 8

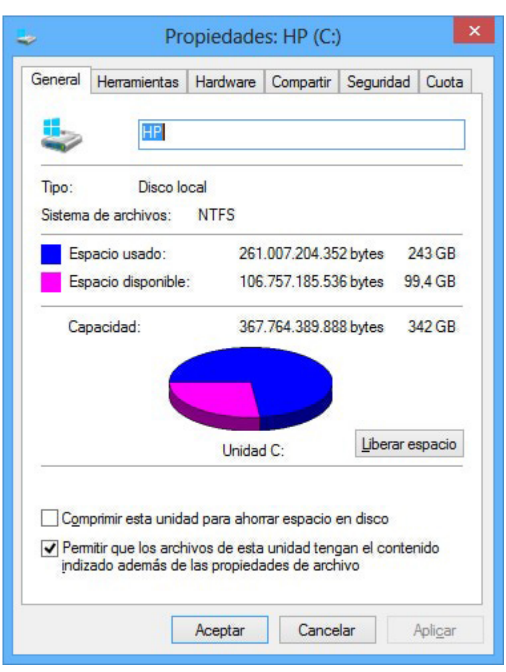


Figura 5.15: Ventana con las propiedades de un disco en Windows 8

Administración de discos

En cualquier Sistema Operativo para eliminar archivos temporales de Internet, cookies y otro tipo de material debidos a la utilización de un navegador, se pueden utilizar las opciones que incorporan los propios navegadores. En caso de Windows también se puede invocar la herramienta llamada *Liberar Espacio* asociada a cada una de las unidades de disco de las que dispongamos. Basta con hacer click en el botón derecho del ratón para acceder a las propiedades del disco (figura 5.15) o bien a través del *Panel de Control* y la categoría *Herramientas Administrativas*.

La *fragmentación* de un disco ocurre con el paso del tiempo a medida que se guardan, cambian o eliminan archivos. Consiste en la dispersión (fragmentación) de los archivos en posiciones no contiguas del disco y provoca que el rendimiento del equipo disminuya. No afecta por igual a todos los sistemas de archivos. Se da muy comúnmente en el sistema operativo Windows aunque también afecta a otros sistemas (pero con menor incidencia).

La *desfragmentación* del disco es un proceso que consiste en volver a organizar los datos fragmentados en la unidad de almacenamiento para que ésta funcione con mayor eficacia.

El *desfragmentador* de Windows es una herramienta que puede ejecutarse manualmente o según una programación establecida. La forma más fácil de acceder a esta característica es a través del *Panel de Control* y la categoría Herramientas Administrativas. Escogiendo *Desfragmentar y optimizar unidades* se accede a la venta que aparece en la figura 5.16:

En OS X el problema de la fragmentación de discos es mucho menor que en Windows y por eso el sistema operativo no contiene un desfragmentador. De todas formas, existen aplicaciones de terceros para realizar esta tarea si se necesita. Lo que sí incorpora OS X es una utilidad llamada *Utilidad de Discos* (figura 5.17) que nos permite hacer otras tareas interesantes como son la verificación de los discos y la restauración de los permisos, algo que de vez en cuando da problemas en Mac.

Copias de seguridad

Para asegurarse de no perder sus archivos debe realizar copias de seguridad regulares de los mismos. Puede configurar copias de seguridad automáticas o ejecutar copias de seguridad manuales en cualquier momento. OS X dispone de una herramienta muy completa y totalmente transparente

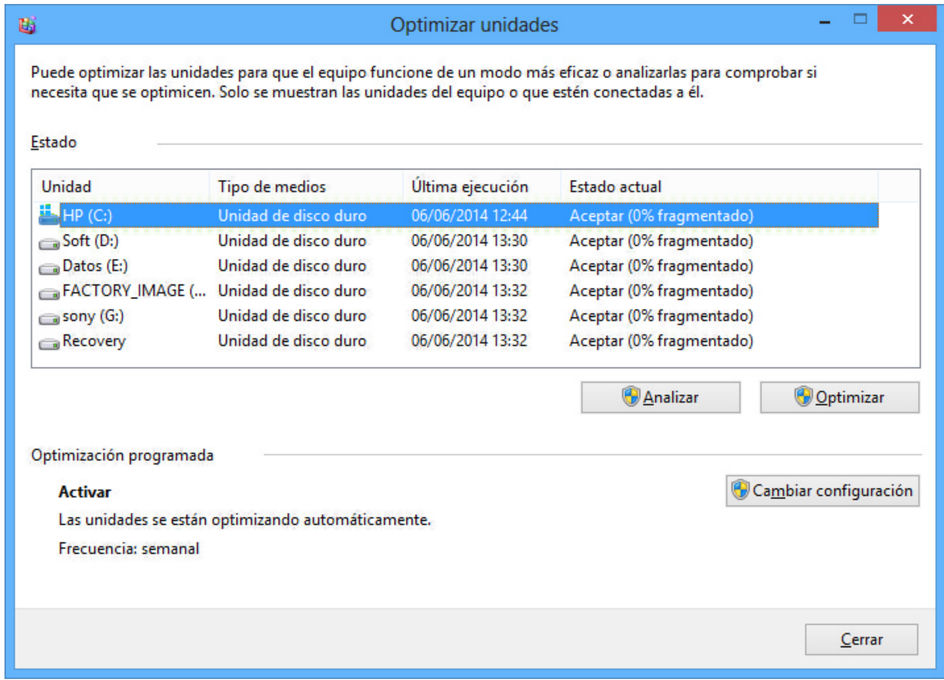


Figura 5.16: Herramienta para optimizar discos en Windows 8

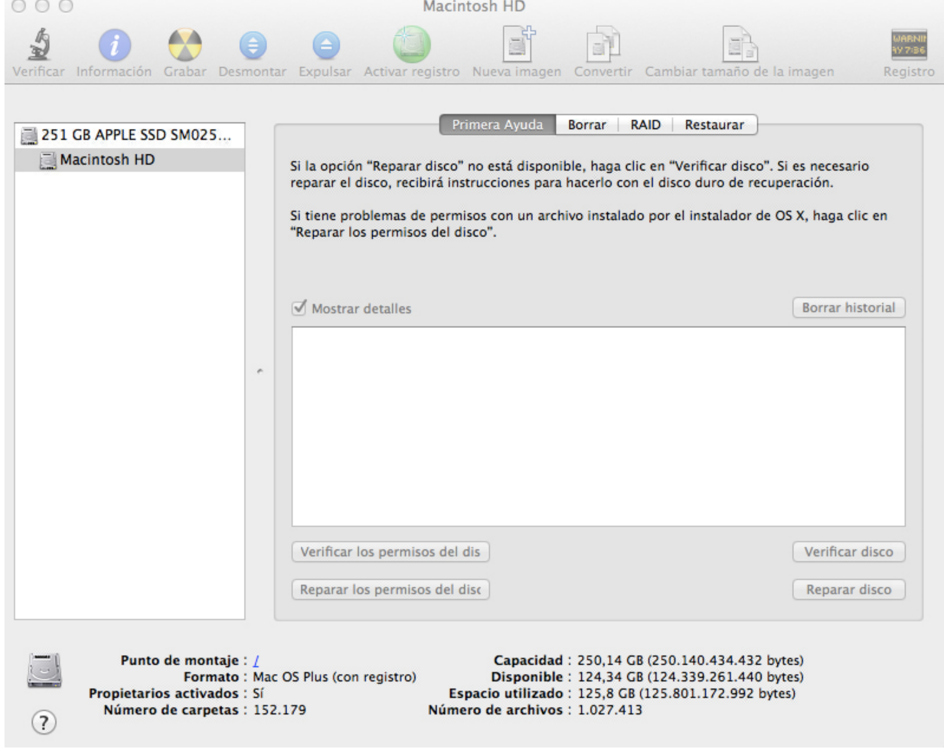


Figura 5.17: Utilidad de discos en Mac OS X

Figura 5.18: Configuración de Time Machine en Mac OS X

al usuario para la realización de copias de seguridad llamada *Time Machine*. Para acceder a la configuración de esta herramienta tendremos que ir a la opción de *Preferencias* del Menú General de OS X (figura 5.12) y luego la opción *Time Machine* (figura 5.18).

La herramienta para realizar copias de seguridad en Windows 8 se llama *Historial de Archivos* y se puede acceder mediante el *Panel de Control* y la categoría *Sistema y Seguridad*. Desde la ventana correspondiente (figura 5.19) se puede configurar los archivos de los que queremos hacer copias, la programación temporal, etc. También se puede realizar el proceso de restauración de archivos en caso de que se produzca alguna incidencia con los mismos.

Como habíamos comentado anteriormente, la versión del sistema operativo utilizada puede hacer que las opciones tengan el nombre o la presentación un poco diferente. A modo de ejemplo se muestra en la figura 5.20 la herramienta para copias de seguridad de Windows 7 a la que accedemos a través del *Panel de Control*, categoría *Sistema y Seguridad* y que se llama *Copias de Seguridad y Restauración*.

Administración de tareas y rendimiento del equipo

Hay ocasiones en las que una cierta aplicación que se está ejecutando empieza a ir muy lenta o simplemente deja de responder. En estos casos podemos detener su ejecución mediante una facilidad que incorporan los sistemas operativos que nos permite gestionar las tareas: en Windows recibe el nombre de *Administrador de Tareas* y en OSX se llama *Monitor de Actividad*. Se puede invocar al *Administrador de Tareas* de Windows mediante la combinación de teclas *CTRL+ALT+SUPR* o haciendo *click* con el botón derecho del ratón sobre la barra de herramientas. Aparece la ventana que se muestra en la figura 5.21. En ella se detallan las tareas que se están ejecutando junto con información tal como el consumo de memoria, de CPU, de disco, etc. Mediante el botón de la parte inferior derecha de la venta podemos finalizar las tareas que nos estén dando problemas.

En OS X accedemos al *Monitor de Actividad* a través del panel de herramientas. Aparecen todos los procesos que se están ejecutando junto con una serie de información asociada. Con el botón llamado *Salir del proceso* podemos finalizar los procesos que queramos.

Además de para detener tareas o procesos, tanto el Administrador de Tareas de Windows como el Monitor de Actividad de OS X permiten realizar otras operaciones de supervisión del ordenador. No tenéis más que fijaos en las pestañas de cada ventana: Rendimiento, Historial de aplicaciones, Usuarios, etc. (en el caso de Windows) y Memoria del Sistema, Actividad de disco, Red, etc (en OS

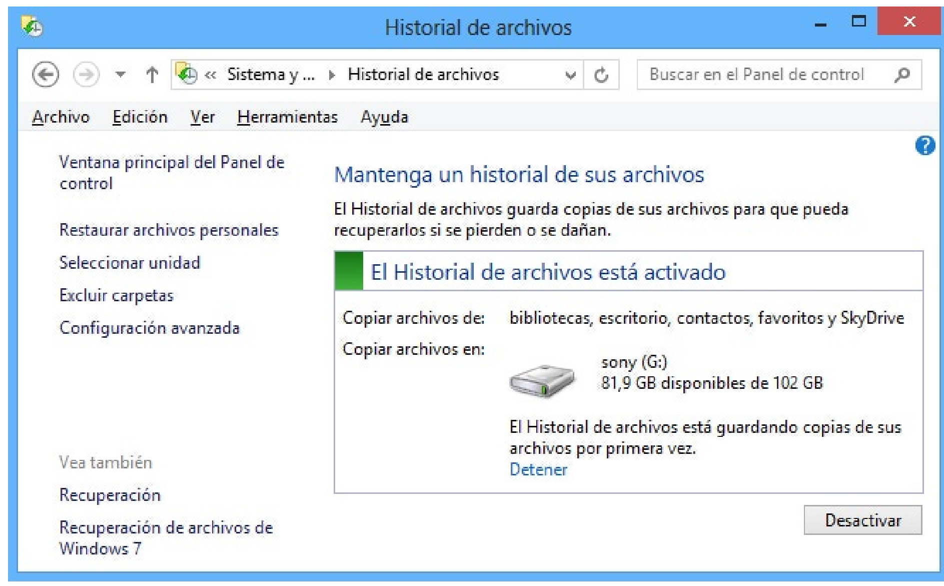


Figura 5.19: Configuración de copias de seguridad en Windows 8

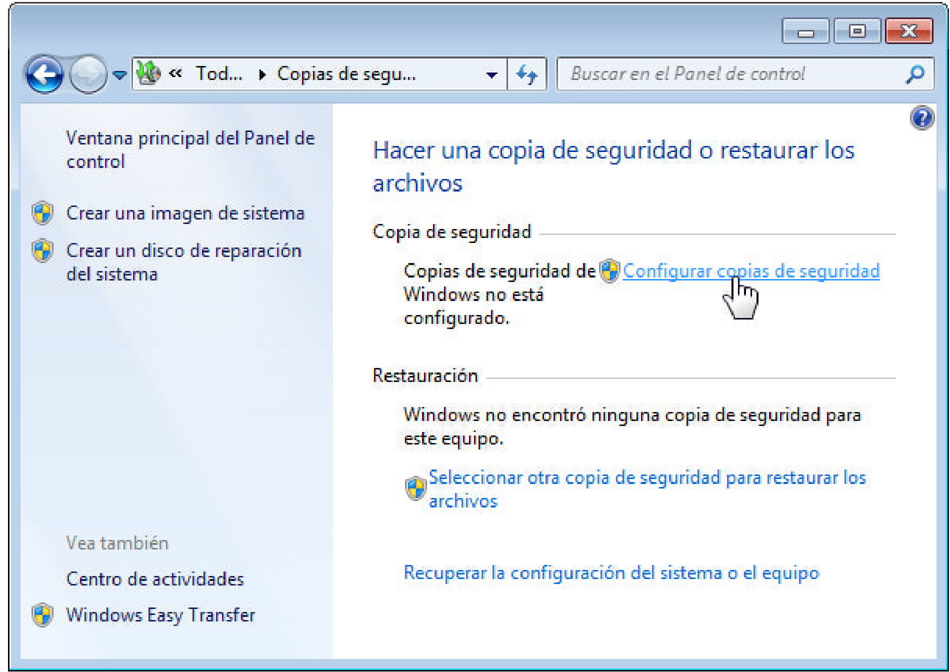


Figura 5.20: Configuración de copias de seguridad en Windows 7

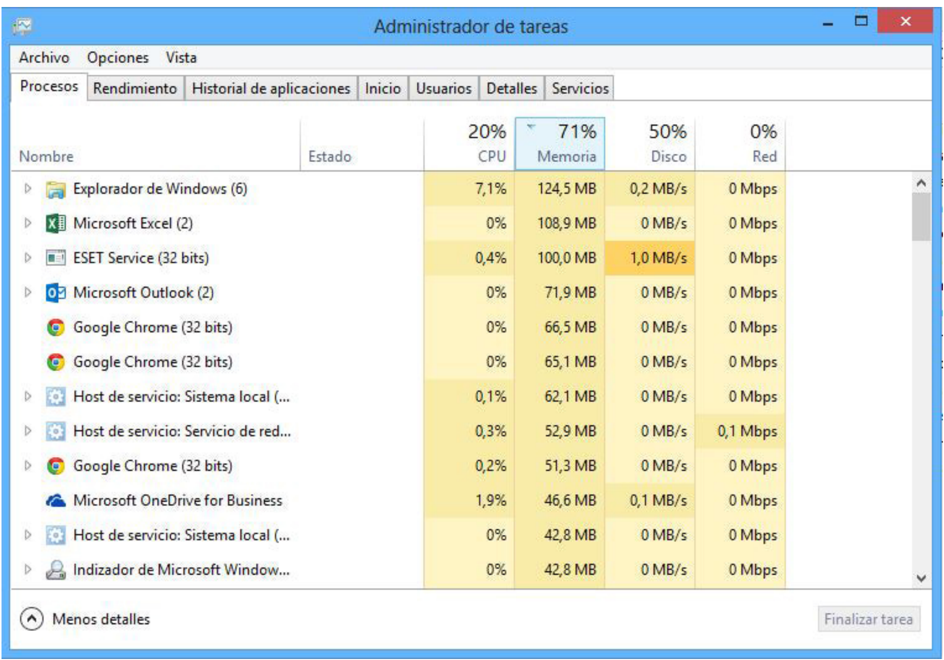


Figura 5.21: Administración de tareas en Windows 8

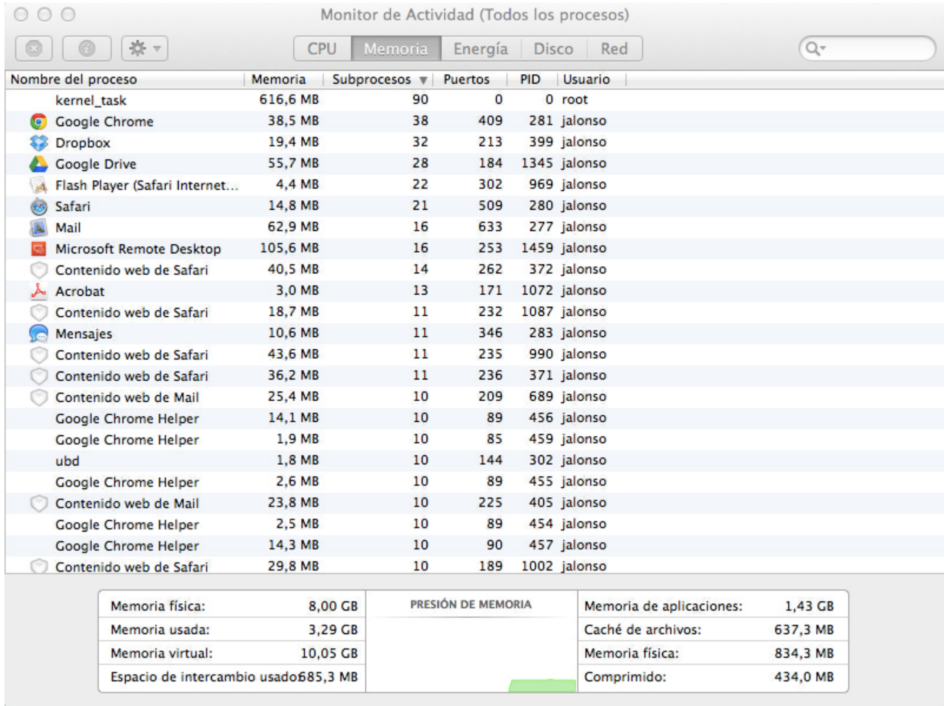


Figura 5.22: Administración de tareas en Mac OS X

X).

5.4. Sistemas operativos utilizados en entornos profesionales de ingeniería

5.4.1. Sistemas operativos en tiempo real

Los sistemas operativos en tiempo real dan soporte a aplicaciones en tiempo real, y por lo tanto deben responder correctamente dentro de un intervalo de tiempo determinado.

En este tipo de sistemas operativos se da prioridad a las aplicaciones (procesos) frente al usuario, dada la importancia de las tareas controladas. Por lo tanto, en estos SSOO el interfaz de usuario es poco importante.

Algunos campos de aplicación son el tráfico (tanto aéreo, terrestre o naútico), sistemas de gestión y control de trenes, o los actuales sistemas de fabricación integrada.

Algunos ejemplos de sistema operativo en tiempo real son:

VxWorks S.O. basado en UNIX, creado por Wind River Systems.

LynxOS S.O. basado en UNIX, creado por LynuxWorks.

eCos basado en Linux Red Hat.

Ubuntu Studio basado en Linux Ubuntu.

5.4.2. Sistemas operativos empotrados

Se conoce como sistema empotrado los sistemas informáticos integrados en un sistema de ingeniería más general. El ejemplo clásico es el de los automóviles actuales, donde un computador de a bordo de un automóvil gestiona todos los subsistemas: frenado, suspensiones, motor, confort, etc. El sistema de ingeniería más general es el automóvil, el sistema empotrado es el computador de a bordo.

Los sistemas empotrados no son computadores de propósito general como los usuales en todas las oficinas (p.e., un PC), sino que están diseñados específicamente para un determinado propósito. Es usual que los sistemas empotrados se encarguen de realizar funciones de control, procesamiento y/o monitorización. A los sistemas empotrados con restricciones de tiempo real se les conoce como **sistemas empotrados de tiempo real**.

Ejemplos de sistemas empotrados:

- Electrónica de consumo: videos, lavadoras, frigoríficos, ...
- Automóviles: Control de velocidad, climatización, ABS, ...
- Telecomunicaciones: Radios, teléfonos móviles, ...

A los sistemas operativos utilizados en sistemas empotrados se les denomina **sistemas operativos empotrados**, los cuáles deben adaptarse para las restricciones de tamaño, memoria principal disponible, energía a consumir, etc. Los sistemas operativos empotrados pueden ser:

- Software muy pequeño desarrollado específicamente para algún sistema embebido en particular.
- Versión reducida de algún sistema operativo de propósito general (Ejemplos: Windows C.E., «Linux embebido»).

Algunos ejemplos de sistema operativo empotrado son:

- Symbian O.S.

- Windows C.E.
- Palm O.S.
- Linux embebido.
- iOS (S.O. del iTouch, iPad, iPhone)

Parte VI

Referencias

Referencias

[1] A. Silberschatz, P. Galvin, and G. Gagne, *Fundamentos de Sistemas Operativos, 7ma Edición.* McGraw Hill, 2006.

[2] A. Prieto and B. Prieto, *Conceptos de Informática.* McGraw Hill, 2005.