Capítulo 8

Otros Estilos de Programación

Índice General

8.1. Introducción **211**
8.1.1. Breve Perspectiva Histórica de los Lenguajes de Programación 212
8.1.2. Metodología y Tecnología de la Programación 215
8.2. Programación Tradicional y Programación Moderna. **216**
8.3. Programación Orientada a Objetos **220**
8.3.1. ¿Qué es? 220
8.3.2. Primer Paso: Programación Procedural. 220
8.3.3. Segundo Paso: Programación Modular. 222
8.3.4. Tercer Paso: Programación Orientada a Objetos. 226
8.3.5. Eventos y Excepciones. 229
8.4. Python: Tipos Básicos y Estructuras de Control. **231**
8.4.1. Introducción. ¿Qué es Python y por qué Python? 231
8.4.2. Tratamiento de Variables y Tipos Básicos en Python. 232
8.4.3. Estructuras de Control de Flujo de Datos. 236
8.4.4. Acciones no Primitivas en Python. 241
8.5. Python: Gestión de la Información. **247**
8.5.1. Tipos Compuestos: Secuencias. 247
8.5.2. Ficheros en Python. 256
8.6. Python: Actualizaciones. **257**
8.7. Agradecimientos. **258**
8.8. Glosario. **258**
8.9. Bibliografía. **259**

“Los maestros te enseñan las puertas, pero eres tú quien debe cruzarlas.”
Proverbio chino.

8.1. Introducción

Todo lo que se ha visto hasta este momento se puede identificar con los contenidos de muchos otros cursos de programación: es un temario clásico, en el sentido de que incluye los principales

aspectos básicos de inicio en el manejo de un lenguaje. También es clásico en el sentido de que refleja los mismos contenidos que podría haber tenido hace quince o veinte años (y que es básico que un ingeniero conozca). Pero la informática ha cambiado mucho en este tiempo. Especialmente en lo referido a la interactuación entre el usuario y los programas: lo más habitual hoy en día no es trabajar frente a una consola y teclear datos. Es mucho más normal estar utilizando un entorno gráfico en el que las acciones típicas son abrir y cerrar ventanas de diálogo, arrastrar dichas ventanas por un "escritorio", utilizar el ratón más que el teclado...

En este tema no se puede explicar completamente cómo se realizan estas aplicaciones gráficas y cómo incorporarlas al trabajo propio. De hecho, para dominar estos aspectos sería necesario comenzar una nueva asignatura. El objetivo primordial de ésta, sigue siendo conocer los conceptos fundamentales de la programación y estos son por sí solos unos contenidos lo bastante amplios como para que no haga falta aumentarlos.

En este tema se pretende sólo dar una pequeña idea de cuál ha sido la evolución que ha permitido llegar hasta este tipo de programación. Por supuesto, en esta evolución ha jugado un papel muy importante el desarrollo del *hardware*: para desarrollar este tipo de aplicaciones son necesarias unas prestaciones (velocidad de la CPU, cantidad de memoria y capacidad de direccionamiento, etc.) que no se conocían hace veinte años. Pero también ha sido muy importante la evolución del *software* y el desarrollo de metodologías y formas de programar. Y es a estos aspectos a los que se pretende prestar mayor atención: a la evolución en la forma de diseñar el *software* y a la propia evolución de los lenguajes de programación, incorporando herramientas que facilitan el diseño de aplicaciones de este tipo.

Primero se realiza una visión resumida de la evolución de los lenguajes de programación, una reseña histórica de cuándo surgieron los más conocidos o los más significativos desde el punto de vista del desarrollo de la metodología y tecnología de la programación.

A continuación, se realiza una pequeña discusión de cuáles son las diferencias más significativas entre el estilo de programación "tradicional" y el "moderno"; al analizar esas diferencias, se llega a la conclusión de que, en gran parte, el estilo de programación "moderno" se está desarrollando de la mano de las herramientas que soporta el paradigma de Programación Orientada a Objetos, POO. La aparición de este paradigma fue un hito muy destacable en la evolución del desarrollo del *sotfware*; surgió de la necesidad de disponer de herramientas flexibles, genéricas y que permitieran un desarrollo más racional de grandes aplicaciones informáticas. Pero, de rebote, su potencia y flexibilidad ha dotado a los lenguajes que se basan en él de herramientas que han permitido esa evolución en las formas de trabajo. Por su importancia teórica y por su relación con estas técnicas, en este tema se presentan también las nociones básicas de este paradigma.

Para finalizar el tema, se realiza una introducción al lenguaje de programación Python. Este lenguaje, bastante fácil de aprender, es un lenguaje orientado a objetos y con una gran colección de bibliotecas que permiten desarrollar de forma no muy complicada aplicaciones gráficas, *scripts* de usuario, aplicaciones de tipo `cgi`, etc. Puede ser muy útil a quien esté interesado en el desarrollo de este tipo de aplicaciones.

8.1.1. Breve Perspectiva Histórica de los Lenguajes de Programación

La máquina de Babbage sólo podía ser programada cambiando los engranajes que ejecutaban los cálculos. Esta forma de trabajar era la misma en el ENIAC en 1942: los engranajes podían ser ruedas dentadas o podían estar gobernados por señales eléctricas, pero la programación se realizaba configurando interruptores y rediseñando completamente las conexiones físicas del sistema para cada nuevo "programa".

En 1945, John Von Newmann introdujo dos importantes conceptos, que afectaron directamente a la forma en la que se concebía la programación. El primero era la *"técnica del programa almacenado"*, en la que se apostaba por que el *hardware* del computador fuera tan simple como fuera posible, de forma que se pudieran utilizar instrucciones complejas para controlarlo, en lugar de reconfigurarlo. Así la reprogramación sería más rápida, ya que para obtener un nuevo programa, bastaría con reescribir las instrucciones, en lugar de reprogramar los circuitos. Es decir, fue Von Neumann el primero en proponer la existencia de programas almacenados en la memoria del computador.

El segundo concepto fue primordial en el desarrollo posterior de los lenguajes de programación. Von Neumann lo denominó *"transferencia condicional de control"*. Esa idea dió lugar a la noción de estructura de control de flujo de datos (inicialmente condicionales simples, `IF` <expresion> `THEN`, y bucles `FOR`) y a la noción de subrutina.

En 1949, apareció el lenguaje Short Code. Fue el primer lenguaje de programación de computadores electrónicos y requería que el propio programador tradujese las instrucciones a secuencias de 1's y 0's. En 1951, Grace Hooper escribió el primer compilador, `A-0`. En 1957 apareció el lenguaje FORTRAN, FORmula TRANslation, diseñado por un equipo de IBM. A pesar de que hoy en día parezca primitivo (en el sentido de que sólo disponía de las instrucciones `IF`, `FOR` y `GOTO`), supuso un gran paso en la programación; tanto más cuando sus tipos básicos incluían el lógico (`boole`), el entero, el real y los números de doble precisión.

FORTRAN es un lenguaje orientado al cálculo científico, pero no está indicado para el manejo de un gran volumen de datos de entrada y salida. Para las aplicaciones más orientadas a la gestión de empresas, se desarrolló el COBOL (COmmon Business Oriented Language): sólo podía manejar números y cadenas, pero permitía su agrupación en registros y tablas, lo que facilitaba la gestión de los datos. Su orientación a la administración de empresas[1] dotó a COBOL de su estilo peculiar: dividido en secciones, como un informe, y con instrucciones construidas con una sintaxis muy similar a la del idioma inglés.

En 1958, John McCarthy creó el LISP, LISt Processing, para la investigación en Inteligencia Artificial. Es un lenguaje *funcional* [2] y diseñado para un campo específico de investigación, por lo que su sintaxis puede resultar chocante. Su diferencia más notable consiste en disponer de un único tipo de datos, la lista, denotada como una colección de elementos agrupados entre paréntesis. Los propios programas en LISP son un conjunto de listas, lo que confiere al lenguaje de la habilidad única de modificarse a sí mismo. Su alta especialización y su naturaleza abstracta hacen que este lenguaje esté todavía en uso.

El lenguaje Algol, ALGOrithmic Language, fue creado para aplicaciones científicas en 1958. Su principal aportación fue la de ser el primer lenguaje cuya sintaxis se expresaba mediante una gramática formal, conocida como Backus-Naur Form o BNF; también implementó nuevos conceptos como las llamadas recursivas de las funciones. Se puede considerar el padre de lenguajes como Pascal, C y C++. De hecho, su sucesor, el Algol 68, demasiado farragoso, no pudo competir con Pascal, mucho más compacto.

Pascal nació en 1968 y fue desarrollado por Niklaus Wirth. Su desarrollo fue motivado especialmente por la necesidad de disponer de un buen lenguaje orientado a la docencia: sus diseñadores no confiaban en su éxito comercial y dedicaron su esfuerzo, principalmente, a la creación de buenas herramientas de desarrollo para facilitar su uso en la educación, sobre todo, herramientas de edición y depuración.

[1] El éxito de este lenguaje fue muy grande; todavía en la actualidad existen grandes empresas y bancos cuyos programas están escritos en COBOL (y, por supuesto, en funcionamiento).

[2] Recuérdese lo ya comentado en el primer tema sobre paradigmas de programación: además del imperativo, existen el funcional y el lógico, cuyas principales áreas de aplicación caen dentro del campo de la Inteligencia Artificial.

Pascal fue diseñado combinando las mejores características de COBOL, FORTRAN y Algol, y eliminando muchos de sus defectos, lo que hizo que ganara rápidamente usuarios. Combinaba un buen proceso de entrada/salida con amplias posibilidades de cálculo y estructuras de datos básicas. Además, mejoró el uso del tipo *puntero*, que posibilitaba el desarrollo de variables dinámicas, mediante las instrucciones `NEW` y `DISPOSE`. Sin embargo, no soportaba ni los vectores dinámicos ni los grupos de variables, lo que supuso que fuera perdiendo popularidad. Su sucesor, Modula-2, surgió cuando el lenguaje `C` ya había ganado muchos adeptos.

`C` fue desarrollado en 1972 por Dennis Ritchie, en los laboratorios Bell. Incluía todas las características de Pascal; su uso intensivo de los punteros lo hacía más rápido y poderoso a expensas de su legibilidad. Además eliminaba muchos de los errores descubiertos en Pascal, por lo que muchos usuarios de éste, lo aceptaron rápidamente. Hay otro hecho indiscutible, la relación entre el desarrollo de `C` y el del sistema operativo Unix. La decisión de realizar el sistema operativo Unix en el lenguaje `C`, hizo que al lenguaje se incorporase una mejor interfaz para acceder a las llamadas básicas del sistema operativo, lo que hizo de `C` el lenguaje habitual para quien quisiera programar sistemas operativos como Unix, Windows, MacOS o Linux.

Entre finales de la década de los 70 y principios de los 80 se desarrolló la Programación Orientada a Objetos, POO. Bjarne Stroustroup desarrolló extensiones de `C` con esta nueva orientación obteniendo el llamado "`C` con clases" que, finalmente, derivó al lenguaje C++ en 1983. La aparición de este lenguaje impulsó el uso y el desarrollo de este paradigma, que surgió conceptualmente asociado al lenguaje *Smalltalk*[3].

A principios de los 90, un proyecto orientado hacia la televisión interactiva, desarrollado por Sun Microsystems, desembocó en el desarrollo del lenguaje Java, altamente portable debido a su concepción inicial. Su incorporación al navegador Netscape le dió gran popularidad y le hizo convertirse en el "lenguaje del futuro". Pero, la aparición de los lenguajes de guión (también llamados de "script"), ha puesto en evidencia algunos problemas de optimización de Java en este tipo de aplicaciones.

Perl, Practical Extraction and Reporting Language, se ha hecho especialmente popular con la aparición de Internet, ya que a sus características como lenguaje de guión, hay que añadir su aplicación en interfaces web. Sus funciones de manejo de texto lo hacen ideal para estas tareas. Fue desarrollado por Larry Wall en 1987, como una extensión de las herramientas Unix `sed` y `awk`.

Esta descripción de la historia de los lenguajes no está completa, ni mucho menos. Hay más de 2500 lenguajes de programación. Una referencia muy interesante es la página web sobre la historia de los lenguajes de Éric Lévénez [4]. En ella se encuentra un árbol genealógico que no sólo muestra la explosión de lenguajes de programación existentes, sino que también ilustra las relaciones entre ellos y su cronología.

Las ramas principales parten, en 1958/1959 de FORTRAN, COBOL, Algol y Lisp; al cabo de 10 años, 1968/1969, se mantienen estos cuatro lenguajes, y han aparecido PL/I, Basic, Snobol, Forth, Logo y, sobre todo, Simula y Smalltalk. Prolog, Pascal y `C` aparecen en los dos años siguientes y Algol desaparece.

La situación al cabo de 10 años más, 1978/1979, muestra la aparición de Rex, `awk`, Scheme, Modula 2 y Ada. La década de los 80 aparece dominada por los lenguajes OO y variantes de lenguajes adaptándose a este paradigma; aparece C++ y nuevos lenguajes funcionales como ML.

Al cabo de otros diez años, en 1988/1989, comienzan a irrumpir en escena los lenguajes de

[3] Y, de hecho, muchas de las innovaciones en la interactuación entre usuario y programa, fueron introducidas por el equipo de desarrollo de este lenguaje, pertenecientes a *Xerox Parc*: ahí se desarrollaron originalmente las ideas que posibilitaron el desarrollo de interfaces amigables, como las que siempre han sido características en los sistemas Atari, Amiga y Apple (y que, posteriormente, imitó Microsoft con Windows).

[4] `http://www.levenez.com/lang/`

guión, principalmente Perl, Tcl/Tk y Python, y también aparecen Java y Haskell. A partir del año 2000 los lenguajes son Python, C# (C++ orientado a Internet), Java, Ruby, Perl y O'caml.

Si se estudia de dónde parten sus ramas, se ve que Python nace a partir de Modula 3 y `ANSI C`. C#, parte de C++ y Java; Java, a su vez, hereda características de C++, Smalltalk y Scheme. Ruby surge desde Python, Eiffel, C++ y Perl y, a su vez, Perl evoluciona desde `awk` y `csh`. O'caml es un derivado de ML. Efectivamente, esta es una época de lenguajes multiparadigma, tal y como se aborda en la siguiente subsección.

8.1.2. Metodología y Tecnología de la Programación

Tras esta descripción histórica de los principales lenguajes de programación, se esconde una evolución en conceptos metodológicos. Hasta principios de la década de los 60 no se disponía de compiladores y los principales esfuerzos se dirigieron a su desarrollo. Una vez que los compiladores permitieron el desarrollo de programas de una cierta magnitud y de forma cómoda para el programador, se produjo la llamada "crisis del software" que provocó que la programación comenzara a constituirse como una verdadera disciplina y empezara a tomar cuerpo la metodología y tecnología de la programación. El primer paso fue la definición de la *Programación Estructurada*, que ha perdurado hasta nuestros días a partir de los trabajos de Dijkstra, Hoare y Wirth, entre otros. Esta época coincide con el hito de los años 1968/1969 en la anterior descripción histórica de los lenguajes, y de esa época datan también los primeros trabajos sobre métodos de diseño, verificación y prueba de algoritmos.

A principios de la década de los 70, se conocían ya los resultados más importantes relativos a estructuras de datos. En este ámbito hay que destacar la obra de Donald Knuth, "The Art of Computer Programming", y su contribución a la modernización del tratamiento de las estructuras de datos. Así, la programación estructurada dio paso, de forma natural y a partir de los tipos estructurados de Pascal, al concepto de Tipo Abstracto de Datos. La abstracción funcional y la abstracción de datos condujeron a la Programación Modular. Es el segundo hito marcado en la descripción anterior, los años 78/79 y la aparición, entre otros de Modula-2 y Ada.

La evolución continuó y en los 80 aparecieron tres nuevos paradigmas en programación. La abstracción funcional y el renacido auge de la Inteligencia Artificial, derivó en los paradigmas declarativos, el *Funcional* y el *Lógico*. Tal y como ya se comentó, en esa década aparecieron muchos lenguajes funcionales (Haskell, ML, Scheme) y también fue la década de mayor popularidad del Prolog. A su vez, la abstracción de datos derivó en el paradigma de *Programación Orientada a Objetos*. Corresponde a la época de la aparición de C++ y la adaptación de otros lenguajes al paradigma OO.

La última década no está marcada por la aparición de nuevos paradigmas, ni por desarrollos conceptuales tan innovadores como los de épocas anteriores, sino que está dominada por el fenómeno que ha supuesto la aparición de Internet (gran cantidad de lenguajes de guión) y por la aparición de entornos gráficos de trabajo (con lenguajes que permiten desarrollar la Programación Orientada a Eventos, POE, con su base teórica en la POO). Como resultado, se produce algo muy curioso, la aparición de lenguajes multiparadigma: casi todos estos lenguajes no aportan nada nuevo en este sentido, más bien parecen un "refrito" de conceptos ya desarrollados anteriormente, adaptados a los nuevos tiempos y a la nuevas necesidades de la sociedad de la información. Aunque también es posible que aún nos falte la necesaria perspectiva histórica para poder contemplar esta evolución y advertir cuáles son las herramientas en desarrollo, en el ámbito de la programación, que demostrarán ser fundamentales y tener un impacto duradero.

8.2. Programación Tradicional y Programación Moderna.

Tal y como se ha visto el lenguaje `C` apareció y comenzó a captar gran número de usuarios a mediados de los 70. La forma de trabajar frente a un ordenador ha variado mucho desde entonces, tal y como ya se comentó en la introducción: de sistemas operativos en modo comando se ha pasado a sistemas amigables con interfaz gráfica (ventanas, ratón...) y con la aparición y el uso común de internet han nacido muchos conceptos técnicos inexistentes hace 20 años: páginas web, *e-mail*, aplicaciones `cgi`, `asp`, `jsp`, *applets*... Y el uso (tanto en teoría como en prácticas) que se ha dado al lenguaje en la asignatura puede recordar más al entorno de trabajo de hace 20 años que al moderno.

En este sección se intentará marcar cuáles son las principales diferencias entre un tipo de programación al que se denominará *"tradicional"* frente al tipo que se va a denominar *"moderna"*. Quizás la diferencia más evidente entre ambos tipos de programación sea la referida a cómo se comunican los programas con el usuario; esta será la primera a analizar, pero hay más y no tan evidentes.

Es posible que se dé una visión algo simple, pero pretende recalcar sobre todo, las diferencias (o las dificultades) que un programador "tradicional" puede encontrarse para adaptarse a la programación "moderna" [5].

Para situarse correctamente en el tema, es necesario recordar que las diferencias entre la programación tradicional y la moderna no se sitúan sólo en el ámbito del desarrollo de los lenguajes, sino que tienen también su origen inmediato en la evolución de los Sistemas Operativos, evolución que ha sido posible a su vez por la evolución del hardware (máquinas capaces de direccionar más memoria, con mayor velocidad de proceso...) y por la evolución de las herramientas de programación (lenguajes y bibliotecas de desarrollo).

Las características y diferencias de la programación tradicional frente a la moderna pueden resumirse en el siguiente cuadro:

Programación tradicional	Programación moderna
Representación en modo texto	Representación en modo gráfico
Trabajo en monoprograma	Trabajo en multiprogramación
Trabajo en monotarea	Trabajo en multitarea
Ejecución controlada por el programa	Ejecución controlada por el Sistema
Programación procedural	Programación orientada a objetos

Representación en modo texto/Representación en modo gráfico

Representación en modo texto Se trabaja en un entorno de texto (no gráfico) y el programa en ejecución controla la información representada en la totalidad de la pantalla (no hay "ventanas"); el control de esta se realiza en término de filas y columnas (generalmente 24×80) y un surtido muy limitado de 128 caracteres (*ASCII char set*). Los únicos atributos que pueden tener los caracteres suelen ser el color de tinta y el de papel (trazo y fondo) y con pocas posibilidades, como subrayado y parpadeo.

Es decir, lo que se conoce como un entorno de *consola*.

Representación en modo gráfico El programa en ejecución controla la información representada en su "ventana", de la cual el usuario puede controlar tanto su ubicación como su tamaño. La información se representa tomando como unidad el *pixel* y se dispone de un amplísimo

[5] Y, tal y como se ha desarrollado la asignatura, vosotros sois programadores "tradicionales" ¿A qué tendréis que enfrentaros para pasar a un entorno "moderno"?.

surtido de herramientas y parámetros de representación: un amplio juego de caracteres (*fonts*) con todos sus atributos, así como iconos e imágenes preconstruidas de todo tipo.

La posibilidad de multiprograma (reseñada a continuación) junto con las nuevas posibilidades gráficas, hacen que la pantalla pueda contener múltiples ventanas, representativas de otros tantos procesos en ejecución, en lo que se conoce como "escritorio". Además, la pantalla se convierte en otro dispositivo de entrada: las ventanas pueden arrastrarse, cortarse y redibujarse y se puede copiar información de una ventana a otra, incluso entre aplicaciones diferentes.

Desde el punto de vista de la interfaz que percibe el usuario, existen dos tipos de aplicaciones, SDI y MDI. SDI significa *"Single Document Interface"*. Son aplicaciones que se desarrollan en una sola ventana. Por su parte, en MDI, *"Multiple Document Interface"*, la interfaz de la aplicación es una ventana maestra (*frame window*), que puede contener múltiples ventanas hijas, abiertas simultáneamente. Estas ventanas dependen de la maestra y se cierran cuando se cierra ésta (termina el proceso principal).

Evidentemente, la programación de este tipo de entorno se puede hacer muy compleja; por ello, es habitual encontrar entornos de desarrollo que ofrecen multitud de soluciones preconstruidas (*clases*) y bibliotecas que facilitan esta labor. Así, cuando un programador quiere insertar, por ejemplo, un botón en un formulario no tiene que preocuparse de "dibujarlo": basta con indicar su tamaño, el texto o dibujo de su etiqueta, su posición en una ventana y las acciones que hay que desarrollar cuando en dicho objeto se produzcan determinados *"eventos"* (que pase por encima el cursor, que sea el botón por defecto de un `return`, que se haga clic sobre él, etc.).

Trabajo en monoprograma/Trabajo en multiprograma.

Trabajo en monoprograma Si el SO no admite multiprogramación, sólo se puede ejecutar una aplicación cada vez. Es decir, un programa se ejecuta sin tener que compartir recursos con ningún otro (y, por ejemplo, puede contener órdenes de impresión dirigidas directamente al "puerto" de impresora).

Por ello, se puede trabajar utilizando rutinas y llamadas de "bajo nivel" sin peligro alguno de interferir con nada. La totalidad de los recursos, el procesador, la memoria, los puertos E/S, etc. están a disposición exclusiva del programador.

Trabajo en multiprograma El SO admite multiprogramación y se pueden ejecutar varias aplicaciones a la vez. De hecho, incluso en un sencillo ordenador personal, el propio SO puede mantener en ejecución simultánea varias aplicaciones para sí mismo, además de los programas de aplicación del usuario.

El usuario puede estar ejecutando diversos programas al mismo tiempo, siendo muy fácil saltar de uno a otro. Por ejemplo, puede estar escribiendo un documento con un procesador de texto y simultáneamente, estar consultando ciertos datos que necesita en el navegador de Internet o en una hoja de cálculo. Incluso puede estar ejecutando al mismo tiempo diversas opciones de un mismo programa.

En este entorno, el control es del SO: tiene un control continuo sobre el procesador y los diversos programas en ejecución. Por ejemplo, el SO puede abortar, suspender o asignar tiempos de ejecución a cualquiera de ellos.

En estas condiciones, cuando se ejecuta un programa, se deben compartir los recursos con todas las demás aplicaciones en ejecución. Y, por ejemplo, ya no es posible dirigir directamente al puerto de impresora una orden de impresión; en lugar de eso, se envía una petición al gestor de tareas. En general, no es posible utilizar rutinas y operaciones de E/S de ”bajo nivel", ya que deben ser gestionadas por el SO, sino que se realizan peticiones de determinados servicios.

Esto suele suponer un mayor esfuerzo para el programador que trabaja sobre uno de estos entornos, ya que además del conocimiento de un lenguaje de programación adecuado, se le exige el conocimiento de la interfaz, lo que suele denominar API, *Application Programmer Interface*, del sistema. El resultado de que sea el propio SO el que controla las E/S del programa, supone un continuo diálogo entre programa y sistema, un continuo intercambio de *mensajes*, recibiendo determinada información (entradas), y solicitando determinados servicios (salidas).

Trabajo monotarea/Trabajo multitarea.

Trabajo en monotarea El programa solo tiene una vía (también llamado hilo o hebra, del inglés *thread*) de ejecución, es decir, sólo hace una cosa cada vez.

Trabajo en multitarea El programa puede tener más de una vía de ejecución secuencial; es multihebra (*"multithread"*), puede hacer varias cosas al mismo tiempo y el programador debe controlar dos o más vías de ejecución paralela que pueden estar, o no, sincronizadas entre sí.

Realmente, sólo los equipos multiprocesador son capaces de realizar una auténtica multitarea; los dotados de un solo procesador son capaces de realizar una simulación de tiempo compartido, siempre que el SO y el lenguaje utilizado estén habilitados para ello.

Ejecución controlada por el programa/Ejecución controlada por el sistema.

Ejecución controlada por el programa Desde el diseño inicial del programa, el programador decide exactamente todo lo que va a ocurrir en el programa: cuándo se va a "leer" de teclado, cuándo se va a atender al puerto serie para gestionar una llegada de datos.

El programa (si el lenguaje es `C`) se inicia en la función `main` y termina cuando esta termina. La función `main` puede llamar a otras funciones (que pueden llamar a su vez a otras funciones), pero siempre es el programa el que decide cuál es la acción a ejecutar en cada caso y qué funciones se invocan y cuándo.

En el fondo, esta característica es consecuencia de que el programa controla más o menos directamente sus propios procesos de entrada/salida.

Ejecución controlada por el sistema También podría hablarse de "control de ejecución orientado a eventos": en esto radica la gran diferencia entre la forma de controlar la ejecución de un programa entre un entorno "tradicional" y un entorno "moderno".

En contra de lo que ocurre en la programación "tradicional", en la que es el propio programa el que decide qué se hace y cuándo, en la programación orientada a eventos, POE, la ejecución está controlada por el SO. El programa se parece a un bucle que espera continuamente la recepción de mensajes del SO y responde en consecuencia, ejecutando determinados procesos. Los procesos pueden incluir peticiones de nueva información, o solicitud de determinados servicios.

Los mensajes del sistema tienen su origen en eventos de naturaleza muy diversa: una interrupción del teclado, un clic o un doble clic del ratón en cualquier área de la ventana de la aplicación, o simplemente pasar sobre una zona determinada (por ejemplo, en un navegador, al pasar el cursor sobre un enlace se produce un evento cuya respuesta es el cambio de la forma del cursor). Incluso la terminación del programa ocurre cuando se recibe una petición del sistema en este sentido (porque se ha hecho clic en el botón de cerrar la aplicación o por cualquiera de los otros procedimientos típicos en las aplicaciones gráficas).

Programación Procedural/Programación Orientada a Objetos.

Programación procedural Los lenguajes empleados adoptan una fuerte compartimentación entre los tipos de datos disponibles y las operaciones posibles con ellos, que son fijas e incluidas en el propio lenguaje. Se puede superar con la programación modular, que permite la definición de nuevos tipos definiendo nuevas estructuras y sus operaciones de manejo, pero ello no garantiza la encapsulación y el uso correcto del tipo[6].

Programación Orientada a Objetos Los lenguajes empleados en la programación moderna utilizan los recursos de esta técnica de programación. ¿Por qué?

Hay que aclarar, en primer lugar, que la POO puede utilizarse en la programación tradicional. Surgió como necesidad de una mejor estructuración de la información en función de los objetos a manejar para resolver un problema. Facilita el diseño de aplicaciones complejas, puesto que se centra en la definición de los objetos que la forman, en las operaciones relacionadas con ese objeto y en sus relaciones. Pero sobre todo, dota al programador de una serie de mecanismos que le permiten defender la integridad de un objeto. Sus principales características son el *encapsulamiento de los datos y sus operaciones*, la *herencia*, la *sobrecarga* y el *polimorfismo*.

Virtualmente no existe límite a la complejidad de los tipos nuevos de datos que pueda crear el programador, ni de sus operaciones. Aparte de las ventajas genéricas antes enunciadas, la POO está especialmente indicada para la programación en los nuevos entornos, primero porque facilita el acceso (a través de las clases) al manejo de las bibliotecas gráficas existentes, de manejo de objetos gráficos y de manejo de eventos. Y, segundo, porque la comunicación entre objetos, tal y como se verá, se establece por medios de *mensajes*. Se ha presentado un entorno de trabajo en el que el control está en manos del sistema operativo, que envía mensajes a las aplicaciones y recibe mensajes de las aplicaciones, en el que la ejecución se ve controlada por medio de eventos y en el que se conciben las aplicaciones, y sus relaciones con el mundo exterior, como un mundo de objetos que dialogan y transaccionan. En este entorno sólo es necesario conocer las reglas de diálogo y transacción, y el paradigma OO proporciona ese medio de comunicación de forma natural.

Para finalizar esta sección, es preciso recalcar que los conceptos manejados son generales e independientes; es decir, que un programa “moderno” puede ser multitarea, pero en modo texto, o no orientado a objetos. Sin embargo, la mayoría de las características se dan juntas.

Esta colección de características (o de diferencias entre los dos tipos de programación) podría hacerse más amplia, pero las expuestas son quizás las más significativas. Se ha comenzado con la más obvia, la diferencia entre interactuar en modo consola o en modo gráfico y se ha finalizado con el uso del paradigma OO. Todas las diferencias comentadas hacen uso de características de este paradigma y, por ello, el objetivo de la siguiente sección es realizar una introducción a la POO.

[6]Este aspecto se ilustrará mejor en el ejemplo de la sección siguiente, al hablar de la evolución hacia la POO.

8.3. Programación Orientada a Objetos

8.3.1. ¿Qué es?

En la sección anterior se puso de relieve que la Programación Orientada a Objetos presenta unas características que la hacen de uso común en un entorno de programación moderna: al trabajar en modo gráfico, puesto que se accede a bibliotecas con operaciones que facilitan la programación gráfica; en un entorno de multiprogramación, en el que la comunicación se establece mediante mensajes entre el SO y las aplicaciones; en la gestión de eventos, que no son más que tipos especiales de mensajes que el programa espera para "reaccionar", ...

Pero, ¿qué es la POO?. A partir de la crisis del software, a finales de la década de los 60, se marcaron una serie de objetivos en el desarrollo del software:

- abstracción: cómo representar una situación real en un programa sin elementos superfluos, sólo con los elementos significativos,
- conseguir una división lógica en el proceso de diseño,
- reutilización del código,
- ocultación de la información no significativa (la que no mejora a entender el problema, sino que además puede dificultar su comprensión).

Buscando satisfacer estos objetivos, el desarrollo del software ha seguido una evolución, que ya se comentó en la introducción histórica. Uno de los últimos pasos de esa evolución, el último paradigma significativo, fue el POO. Este paradigma se basa en la identificación y manipulación de *objetos*, es decir, no se busca tanto el diseño de un algoritmo para resolver un problema, sino que el primer paso debe ser la identificación de los objetos que intervienen en la definición de dicho problema, cómo hay que manipularlos y qué relaciones mantienen entre sí.

Los objetos son conjuntos de datos y de funciones que operan sobre esos datos. Tanto datos como funciones se agrupan en una unidad (*clase*) de forma que presentan una interfaz común al usuario, que puede manipular los objetos y tener acceso tanto a los datos (*atributos*) como a las funciones (*métodos*) disponibles.

Con el fin de ilustrar este paradigma, se propone un ejemplo que permita seguir los principales avances en la evolución:

Crisis del software → Procedimientos → Módulos → Objetos.

8.3.2. Primer Paso: Programación Procedural.

Su aparición está ligada a la de la *programación estructurada*; además, de utilizar sólo un conjunto de estructuras de control de flujo con un único punto de entrada y un único punto de salida, la aparición de este paradigma mejoró el concepto ya conocido de subrutina, e introdujo el concepto de parámetro tal y como se conoce hoy en día. En consecuencia, los algoritmos comienzan a estructurarse en diversos procedimientos y funciones.

Por su parte, los datos se representan en estructuras definidas o predefinidas en el lenguajes y se pueden manipular directamente desde el programa o desde funciones y procedimientos. Un lenguaje de programación típico es el lenguaje Pascal estándar UCSD, University of California at San Diego.

Como ejemplo, se propone el siguiente: se ha diseñado en `C` un procedimiento que sobreescribe los elementos nulos de un vector con la media de los elementos anteriores.

sobreescribirCeros()

```
void sobreescribirCeros (float vect[], int tam){
  int i;
  float acum=vect[0];
  for (i=1; i<tam; i=i+1) {
    if (vect[i]==0)
      vect[i]=acum/i;
    acum=acum+vect[i];
  }
}
```

A continuación, se utiliza para escribir un programa que lea, sobreescriba ceros y muestre el resultado, sobre dos vectores distintos, uno de tamaño 10 y otro de tamaño 20. Se propone el siguiente código:

```
1  #define MAX1 10
2  #define MAX2 20
3  void sobreescribirCeros(float v[], int tam);
4
5  int main() {
6    float v1[MAX1], v2[MAX2];
7    int i;
8
9    for (i=0; i<MAX1; i=i+1) {
10     printf("Introduce el elemento %d:", i);
11     scanf(" %f", &v1[i]);
12   }
13   sobreescribirCeros(v1, MAX1);
14   for (i=0; i<MAX1; i=i+1) {
15     printf("Elemento %d es %f", i, v1[i]);
16     printf("\n");
17   }
18   for (i=0; i<MAX2; i=i+1) {
19     printf("Introduce el elemento %d:", i);
20     scanf(" %f", &v2[i]);
21   }
22   sobreescribirCeros(v2, MAX2);
23   for (i=0; i<MAX2; i=i+1) {
24     printf("Elemento %d es %f", i, v2[i]);
25     printf("\n");
26   }
27 }
28 /* A continuación, vendría la implementación de */
29 /* sobreescribirCeros() */
```

Este programa es correcto y está bien estructurado, pero se le pueden hacer críticas: es muy repetitivo y casi todo el trabajo lo realiza `main`... Podría pensarse en la definición de dos procedimientos que permitan leer y escribir un vector, respectivamente, lo que permitiría que no se repitieran secuencias de código casi idénticas (como las de la líneas 11–14 y 20–23, o las de las líneas 16–19 y 25–28).

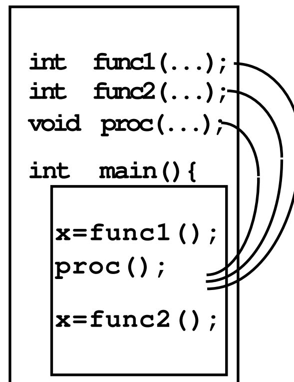


Figura 8.1: Programación procedural: las funciones y procedimientos sólo pueden utilizados desde el programa en cuyo fichero se definen.

Sin embargo, la figura 8.1 intenta poner de relieve un problema: si se definen los dos procedimientos mencionados sólo podrán utilizarse en este programa. Y, sin embargo, parecen dos procedimientos lo suficientemente genéricos como para ser “reutilizados”.

Para ello, es mejor pensar en una solución modular.

8.3.3. Segundo Paso: Programación Modular.

Si se quiere reutilizar el código, es preciso definir mecanismos que permitan implementar sólo una vez aquellas operaciones más cómunes y utilizarlas siempre que sea preciso.

La programación modular, se basa en la definición de módulos. Las funciones y procedimientos se agrupan en diferentes módulos, según el uso más o menos común que puedan tener, teniendo en cuenta, además, que en cada módulo es posible definir estructuras propias e indicar qué funciones o procedimientos se pueden utilizar de forma pública (son accesibles a cualquier usuario del módulo). Así, siguiendo con el ejemplo, parece que tiene sentido pensar en diseñar un módulo que ofrezca una manipulación básica de vectores.

Es habitual que los módulos se definan en dos ficheros. El fichero de definición (el que en `C` se identifica con la extensión `.h`) que determina la interfaz con el usuario: qué funciones y qué estructuras o tipos de datos suministra el módulo. El segundo fichero es el de implementación (el que en `C` se identifica con la extensión `.c`), que implementa las operaciones declaradas en el fichero de definición (las públicas), además de otras funciones que, bien por ser auxiliares, bien porque no interesa que sean accesibles, no se han definido en aquel fichero: son funciones privadas. Un lenguaje típico de este paradigma es el lenguaje Modula-2.

En el ejemplo, la adopción de esta forma de trabajo, supone lo siguiente: se identifica un tipo básico, el vector de enteros y una serie de operaciones básicas (leer, escribir, ...). Asimismo, se observa que dichas operaciones pueden referirse a vectores de distinto tamaño. Para evitar definir

distintos módulos para vectores de distintos tamaños, se adopta como estructura de datos básica una tupla que englobe una definición de vector y la información sobre su tamaño real.

Esto podría conducir al siguiente fichero de definición del módulo `vectorreal`:

vectorreal.h

```
#define MAX 50

typedef struct {
  float vect[MAX];
  int tam;
} VectorReal;

void crearVectorReal(VectorReal *v, int t);
void leerVectorReal(VectorReal *v);
void escribirVectorReal(VectorReal *v);
void sobreescribirCeros(VectorReal *v);
```

que, por supuesto, va acompañado de su fichero de implementación:

vectorreal.c

```
#include "vectorreal.h"

void crearVectorReal(VectorReal *v, int t){
    v->tam=t;
}
void leerVectorreal(VectorReal *v){
  int i;

  for (i=0; i<v->tam; i=i+1) {
    printf("Introduce el elemento %d:", i);
    scanf(" %f", &(v->vect[i]));
  }
}
void escribirVectorReal(VectorReal *v){
  int i;

  for (i=0; i<v->tam; i=i+1) {
    printf("Elemento %d es %f", i, v->vect[i]);
    printf("\n");
  }
}
void sobreescribirCeros(VectorReal *v){
  int i;
  float acum=v->vect[0];
  for (i=1; i<v->tam; i=i+1) {
    if (v->vect[i]==0)
      v->vect[i]=acum/i;
    acum=acum+v->vect[i];
  }
}
```

Esto módulo puede ser utilizado para escribir el programa que se proponía anteriormente:

```
#include "vectorreal.h"
#define MAX1 10
#define MAX2 20

int main() {
  VectorReal v1, v2;

  crearVectorReal(&v1, MAX1);
  leerVectorReal(&v1);
  sobreescribirCeros(&v1);
  escribirVectorReal(&v1);

  crearVectorReal(&v2, MAX2);
  leerVectorReal(&v2);
  sobreescribirCeros(&v2);
  escribirVectorReal(&v2);
}
```

Este programa ya es más elegante y, lo que es mejor, el uso de módulos proporciona una serie de ventajas:

- se ha construido un módulo que es posible reutilizar (código general), tal y como muestra la figura 8.2,
- si se detecta un error en el funcionamiento de este u otro módulo, basta con corregirlo en ese único módulo, y, una vez recompilado y enlazado, la corrección se propagará automáticamente en todos los programas (u otros módulos) que lo utilicen,
- si hay que realizar alguna modificación, no hay que rehacer los programas (u otros módulos) que lo utilicen,
- se consigue una cierta ocultación de la información, una capa de abstracción para el usuario que no esté interesado en detalles de implementación, ...

Y lo que es más importante, y que es realmente la “clave”: tal y como se ha desarrollado el ejemplo anterior, de alguna forma se unen (se “encapsulan”) en una única identidad, datos que, por su naturaleza, siempre deben ir juntos (un vector y su tamaño, en el ejemplo). Además, se especifica cuáles son las operaciones que manipulan esa entidad y se consigue algo que está ya muy cerca de la POO.

Pero, este modo de trabajo aún presenta inconvenientes: no es posible crear varias instancias de un módulo, por ejemplo. Si se desea tener las mismas operaciones para trabajar con vectores enteros, hay que definir completamente todo otra vez en otro módulo.

Y lo que es peor: el módulo no garantiza la integridad de las estructuras de datos definidas en él, tal y como pone de relieve la figura 8.3 o el siguiente programa:

```
#include "vectorreal.h"
#define MAX1 10
```

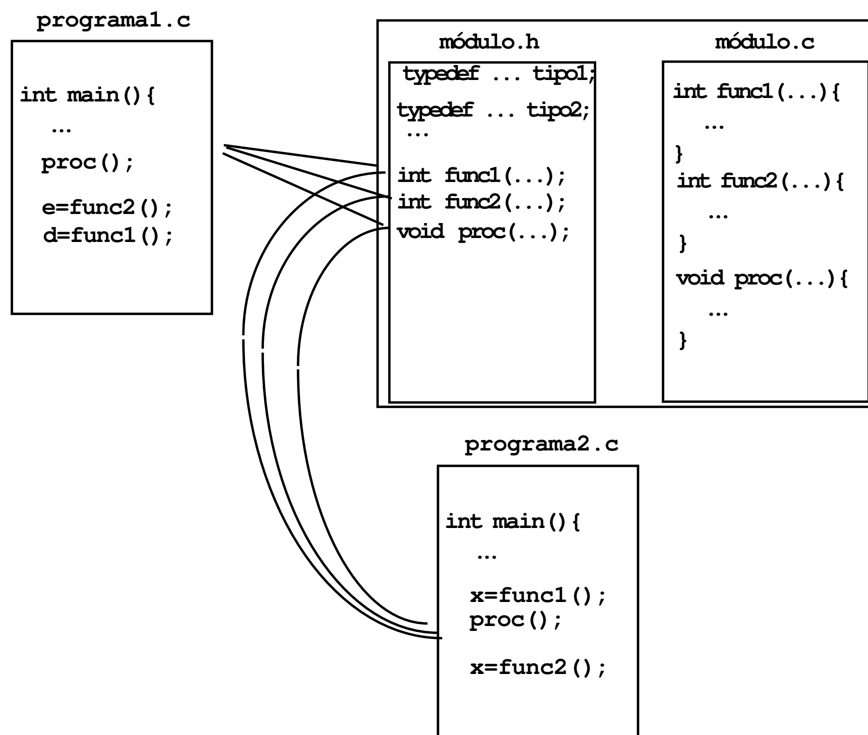


Figura 8.2: Programación modular: pueden definirse módulos cuyas funciones y procedimientos se pueden utilizar desde cualquier programa que incluya al módulo.

```
#define MAX2 20

int main() {
  VectorReal v1, v2;

  crearVectorReal(&v1, MAX1);
  leerVectorReal(&v1);
  v1.tam=30; /* o, peor: v1.tam=100; */
  sobreescribirCeros(&v1);
  escribirVectorReal(&v1);

  ...

}
```

¿Qué ocurre? Un usuario con conocimiento sobre cómo se ha definido el tipo `VectorReal` puede acceder directamente, por ejemplo, al campo tamaño, `tam`, después de que se haya dado un valor concreto a dicho campo. Puede ser simplemente una infracción leve... o puede tener consecuencias insospechadas si, tal y como se hace en el comentario, se da un tamaño mayor del máximo permitido.

Esto es debido a que la programación modular permite al programador hacerse la ilusión del "encapsulamiento" de los datos... pero no le dota de las herramientas que consiguen que ese encapsulamiento sea real, tal y como lo hace la programación orientada a objetos.

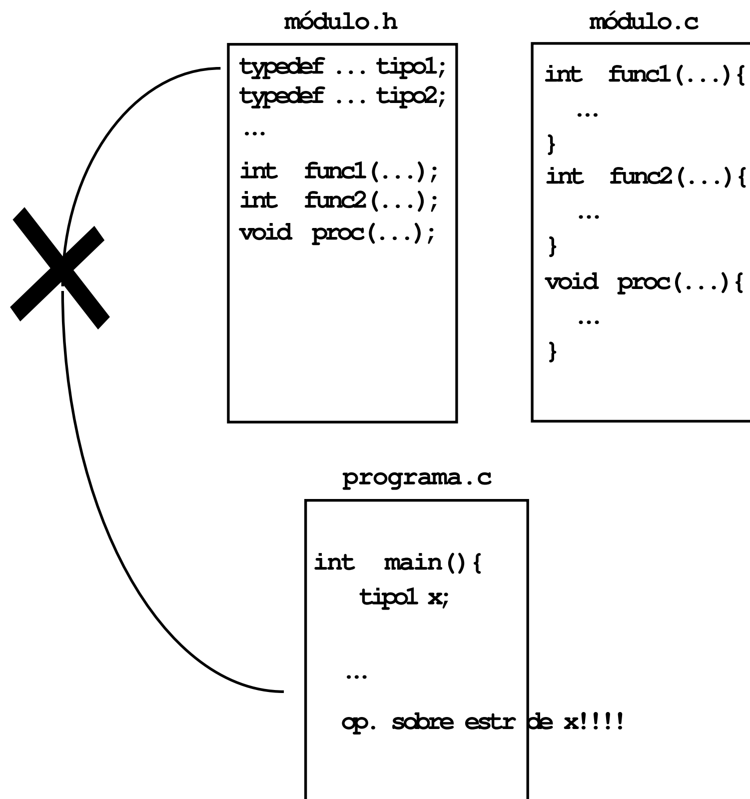


Figura 8.3: Programación modular mal utilizada: un usuario con acceso a la definición de estructuras puede acceder internamente a un tipo con consecuencias imprevisibles.

8.3.4. Tercer Paso: Programación Orientada a Objetos.

Tal y como se ha dicho, la POO se preocupa sobre todo por la identificación de objetos, la identificación de las operaciones que puede realizar y la identificación de relaciones entre diferentes objetos. Quizás no sea el más conocido, pero el lenguaje a partir del cual se definió el paradigma fue el *Smalltalk*.

Para trabajar con objetos, el primer paso es la definición de una *clase*: en ella se indica una serie de ítems de información y cómo operar con ellos. Además, se indica qué partes de esa clase son *públicas* y cuáles son *privadas*. Y el acceso a la zona privada *no es posible*.

El ejemplo anterior, podría dar lugar a la siguiente definición de la clase `VectorReal`:

vectorreal.h

```
#include <iostream>
using namespace std;

```

```
const int TAMANHOMAX=100;

class VectorReal {
private:
  float vect[TAMANHOMAX];
  int tam;
public:
  VectorReal(const int);
  void leerVectorReal(void);
  void escribirVectorReal(void);
  void sobreescribirCeros(void);
};
```

Sin pretender entrar a detalles de sintaxis (la del ejemplo corresponde a C++), sí que se quiere hacer notar el parecido que existe entre la definición de esta clase y la definición de un `struct` en `C`.

En esta definición de clase, aparecen una serie de funciones que, por supuesto, hay que definir. Entre estas funciones, las puede haber de distinto tipo: *constructoras*, *destructoras*, *de manejo*...

En concreto, en el ejemplo hay una operación constructora, `VectorReal`, y una serie de operaciones de manejo. No se permite el acceso a los datos, ya que se han declarado como privados; sí que se permite el acceso a las funciones (que se pasarán a denominar ya correctamente como *métodos*, que es la notación habitual en POO).

vectorreal.cpp

```
#include "vectorreal.h"
VectorReal::VectorReal (const int n){ // Constructor
  tam=n;
}
void VectorReal::leerVectorReal (void){
  int i;
  for (i=0; i<tam; i=i+1) {
    cout << "Introduce el elemento " << i << " : ";
    cin >> vect[i];
    cout << endl;
  }
}
void VectorReal::escribirVectorReal (void){
  int i;
  for (i=0; i<tam; i=i+1) {
    cout << "Valor del elemento " << i << ": " << vect[i] << endl;
  }
}
void VectorReal::sobreescribirCeros (void){
  int i;
  float acum=vect[0];
  for (i=1; i<tam; i=i+1) {
    if (vect[i]==0)
      vect[i]=acum/i;
    acum=acum+vect[i];
  }
}
```

Y ¿qué es un objeto?. Formalmente, se define como una *instancia de una clase*. En el ejemplo siguiente se muestra cómo realizar dicha instancia: no hay acceso a la estructura de datos, pero sí al método constructor `VectorReal`.

```
1  #include "vectorreal.h"
2
3  main () {
4    VectorReal v1(10), v2(20);
5
6    v1.leerVectorReal();
7    v1.sobreescribirCeros();
8    v1.escribirVectorReal();
9
10   v2.leerVectorReal();
11   v2.sobreescribirCeros();
12   v2.escribirVectorReal();
13 }
```

En la línea 4 del siguiente programa *se están realizando dos instancias* sobre la clase `VectorReal` mediante el uso del constructor y dando valores concretos a su parámetro `n` (lo que se podría asimilar a "declarar dos variables de ese tipo").

Y, en las líneas siguientes, se *envían mensajes* a los dos objetos instanciados: en las líneas de la 6 a la 8, se envían a `v1` para que realice los métodos `leerVectorReal`, `sobreescribirCeros` y `escribirVectorReal` respectivamente (lo que se podría asimilar a "realizar una llamada a una función"). Algo similar ocurre en las líneas de la 10 a la 12, pero esta vez sobre el objeto `v2`.

De nuevo, insistir en que no se pretende hacer hincapié en la sintaxis, pero hay que hacer notar que si la definición de la clase recuerda a la definición de un `struct`, la forma de invocar métodos, utilizando el operador punto, `'.'`, recuerda a la forma de referirse a los campos de un `struct`.

Los objetos utilizan la metáfora de la *mensajería*: un objeto pasa un mensaje a otro y el receptor responde ejecutando una de sus operaciones (*método*). Un objeto es invocado por medio de un mensaje que contiene un destinatario, una operación y una lista de argumentos:

`v1.leerVectorReal()` ⇒ el objeto `v1` es invocado para realizar el método `leerVectorReal`, que no precisa parámetros de entrada.

Ya se ha comentado en la sección anterior el uso intensivo que de este mecanismo, los mensajes, se realiza en la programación moderna. Además, el uso de la POO garantiza de verdad la *abstracción* y el *encapsulamiento* de los datos: ya no hay acceso a ninguna estructura de datos, sólo a los métodos declarados como públicos y la distinción entre código público y privado garantiza el uso correcto de las estructuras (si en el programa anterior se intentase realizar la asignación `v1.tam=30`, el compilador indicaría un error). La figura 8.4 intenta reflejar cómo se realiza la comunicación entre una clase y el programa que la utiliza.

Además, este paradigma dota al programador de un mecanismo verdaderamente robusto y versátil para la definición de nuevos datos. En cursos posteriores, se completará el estudio del paradigma. Mencionar aquí, otras dos características importantes de la POO, que pueden dar una idea de hasta donde llega la robustez y la versatilidad del paradigma:

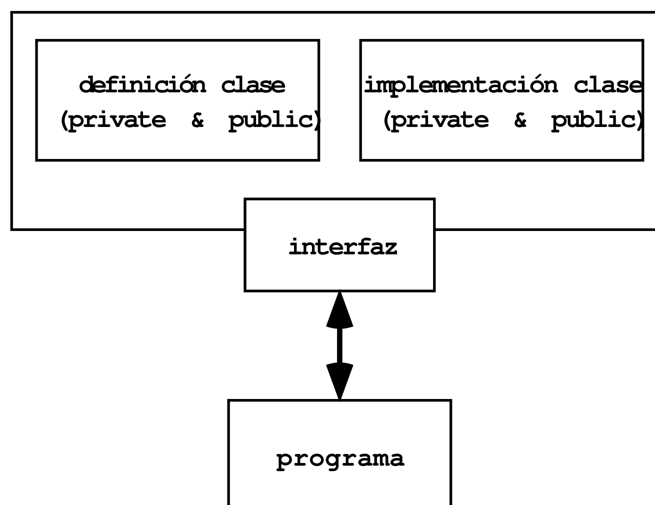


Figura 8.4: Programación Orientada a Objetos: un usuario sólo puede acceder a aquellos elementos que son públicos; se ofrece una interfaz con las posibilidades de comunicación y se garantiza la privacidad de los elementos que no son accesibles.

Polimorfismo: Esta característica está relacionada con la posibilidad de definir métodos con el mismo nombre, pero que reaccionarán de formas distintas ante un mensaje.

Se puede conseguir, entre otras formas, mediante la *sobrecarga de funciones* que permite definir métodos con el mismo nombre, pero distinto número y/o tipo de parámetros, o mediante la *sobrecarga de operadores*, de forma que un operador además de su operatividad original (por ejemplo, `a+b` para representar la suma entera), realice alguna operación especial en la clase a la que están asociados (por ejemplo, que `a+b` también sirva para representar la suma de complejos).

Herencia: Esta característica permite que, partiendo de una clase ya definida, se pueda implementar otra nueva de forma que *herede* ciertos métodos de su clase *madre* con la ventaja de que no hay que volver a implementarlos, sólo habría que añadir aquellos que fueran nuevos o sobreescribir aquellos que interese modificar.

Asociada a esta característica surgen conceptos como *Jerarquía de clases*, formada por una clase madre y todas sus clases descendientes o la *herencia múltiple*, relacionada con la posibilidad de heredar métodos de más de una clase madre.

8.3.5. Eventos y Excepciones.

Eventos y excepciones no son característicos de la POO, pero sí que es cierto que tanto un *evento* como una *excepción* generan mensajes similares al que puede generar un objeto; y, que el manejo de eventos y de excepciones es hoy en día una posibilidad que admiten casi todos los lenguajes orientados a objetos.

Eventos: Los *eventos* son sucesos captados por el Sistema Operativo. Estos sucesos pueden ser tratados íntegramente por el SO o pueden ser trasladados a los programas de usuario en ejecución en forma de mensajes. Según su origen se dividen en dos categorías:

- Eventos de usuario (o externos): Los eventos de usuario son acciones que tienen su origen en el usuario del programa. Ejemplos de eventos de este tipo son que el usuario pulse el botón del ratón, que el usuario pulsa una tecla del teclado o que el usuario pulse dos veces el botón del ratón. Es decir, están siempre relacionados con alguna acción del usuario.
- Eventos de Sistema (o internos): Este tipo de eventos son aquellos que se inician en el Sistema Operativo. Por ejemplo, que es necesario redibujar una ventana, o que se ha creado una nueva ventana o bien que se ha atendido una determinada interrupción...

Excepciones: El manejo de *excepciones* permiten gestionar errores de ejecución o errores del sistema. Un ejemplo de la forma tradicional de manejar este tipo de errores puede ser la evaluación del resultado devuelto tras la llamada a una función (como cuando se intenta abrir un fichero y se comprueba que el resultado devuelto no es el puntero nulo, por ejemplo). Hay dos estrategias bastante comunes para manejar este tipo de situaciones: incluir el código del error en el resultado de la función, o utilizar una variable global con el estatus del error.

En ambos casos, el programador debe comprobar si ha ocurrido un error, y predecir una acción apropiada que lo maneje, como ocurre en los siguientes ejemplos:

```
void ecSegGra(float a, float b, float c,
                        float *x1, float *x2, boole *haySol) {
  float d;

  d=b*b-4*a*c;
  if (d<0)
    *haySol=FALSO;
  else {
    *haySol=CIERTO;
    d=sqrt(d);
    *x1=(-b+d)/(2*a);
    *x2=(-b-d)/(2*a);
   }
}
```

```
FILE *fptro;

fptro=fopen("mifichero","r");
if (fptro!=NULL) {
  while (!feof(fptro)) {
     /* Código para procesar datos */
  }
  fclose(fptro);
}
else {
  /* Código en caso de error */
}
```

Esto produce programas de gran calidad, pero en los que una gran parte del código se dedica a controlar si todas las acciones se desarrollan normalmente o si se produce algún tipo de error. Además de que puede ser difícil prever todas las posibles situaciones de error y todos los posibles

puntos en los que puedan aparecer, la inclusión de estas porciones de código pueden contribuir a que el programa sea bastante difícil de seguir.

En los entornos de programación más recientes se ha desarrollado una forma alternativa de manejar los errores, conocida como *manejo de excepciones*: se genera una excepción tan pronto como aparece un error. El sistema fuerza un salto hacia el bloque de excepciones más cercano del código, y en él se toman las acciones apropiadas para o bien solucionar el error o bien alertar acerca del error producido. El sistema, además, proporciona un "manejador" estándar por defecto que toma todas las excepciones y que muestra los mensajes de error, deteniendo la ejecución del programa. El bloque que maneja las excepciones se codifica de manera análoga a un bloque `if/else if/else if/../else`, en el que, en cada rama, se especifican las acciones a tomar dependiendo de la excepción detectada.

Además del manejador estándar, suele haber otro tipo de bloque de excepciones asociado que permite "limpiar" el entorno después de que se haya generado un error y que se usa típicamente para cerrar archivos, vaciar búferes, etc. Este bloque se ejecuta al final de la ejecución, independientemente de las acciones que haya tenido que realizar el manejador de excepciones.

También existe la posibilidad de que el propio programador pueda generar sus propias excepciones, de forma similar al uso de condicionales para detectar errores de cálculo habitual. Por ejemplo, el programador puede generar una excepción si detecta que el denominador es cero antes de realizar una división.

8.4. Python: Tipos Básicos y Estructuras de Control.

8.4.1. Introducción. ¿Qué es Python y por qué Python?

El lenguaje Python fue diseñado por Guido Van Rossum en 1991 y, desde entonces, está en continua evolución: ha sido ampliamente aceptado por la comunidad de desarrolladores de software de código abierto y, de ahí, que casi cada seis meses se presente una nueva versión.

En relación al motivo para dar una introducción a Python, hay dos objetivos claros. Primero, que se adquiera la consciencia de los conocimientos previos: cualquiera que haya cursado la asignatura hasta este punto ha adquirido ya más conocimientos de los que pueda sospechar. Y, como demostración, está el hecho de que en un sólo tema se cubrirán casi todos los aspectos que se desarrollaron hasta el momento ¡en siete!.

El segundo objetivo es presentar un lenguaje que permite desarrollar de forma sencilla ese tipo de programación "moderno" al que se ha hecho referencia. Con la ventaja añadida de que Python resulta sencillo de aprender y de que se han desarrollado gran cantidad de bibliotecas que permiten realizar una gran cantidad de aplicaciones en este lenguaje.

Como ejemplo del entusiasmo de mucho de sus usuarios, sirva el siguiente: cuando se pregunta sobre las ventajas de Python a alguno de estos usuarios [7], el repertorio de respuestas es bastante amplio:

1. Tiene una curva de aprendizaje muy rápida, en comparación con otros lenguajes como el `C`, Java o sus compañeros de viaje Perl y PHP (en esto coinciden casi todos: es fácil de aprender).

[7] Básicamente, compañeros vuestros de cursos superiores de Ingeniería Informática.

2. Su sintaxis es limpia y estructurada, y la indentación de los programas también facilita su lectura.
3. Como sus hermanos Perl y PHP, es de fácil uso en las “nuevas tecnologias de la información”, como páginas web (CGIs), listas de correo (mailMan), PDAs, etc.
4. Gestión automática de la memoria; esto permite olvidarte de malloc/free.
5. Proporciona tipos abstractos de datos de alto nivel, como diccionarios y listas. Esta característica es “built-in”, no es necesario incluir una biblioteca específica para obtenerla.
6. Es un lenguaje orientado a objetos, como la mayoría de los lenguajes modernos. Y es fácil crear módulos.
7. Es fácil llamar o cooperar con otros programas desde Python, incluidos los creados en otros lenguajes como C, C++ o Java.
8. Es uno de los lenguajes para programar en Zope o Narval.
9. Apto para “proceso por lotes” (.BAT). Este lenguaje se utiliza en los ficheros de configuración de distribuciones como SuSe, o escritorios como Gnome, entre otras.
10. Disponible para una gran variedad de sistemas operativos y plataformas: Linux, Sparc Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, Palm OS, QNX, Playstation, VxWorks, Sharp Zaurus, entre otras... Windows y Pocket PC, también.
11. Su núcleo esta hecho en C. ¿Por qué pelearse con el C si realmente son primos hermanos y Python es más fácil?[8]

El lenguaje Python es un lenguaje interpretado; esto es, cada vez que se ejecuta un programa se va traduciendo cada una de las instrucciones que lo forman. Esta circunstancia permite trabajar en un entorno interactivo, en el que se dan órdenes al intérprete y éste las ejecuta, o bien, editando un fichero con un programa y lanzándolo posteriormente a ejecución. En este tipo de entorno no es necesario hacer una declaración de variables: hay quien ve en esta característica una ventaja (¿escribir menos líneas?), hay quien lo ve como un inconveniente (es más difícil leer e interpretar el código: al fin y al cabo la “lista de variables” siempre da un referencia de cuántas hay y de qué tipo son... y, por lo tanto, qué se puede hacer con ellas).

A lo largo de las siguientes subsecciones, se mantendrá el mismo esquema que se desarrolló en el resto de apuntes con el lenguaje C. Además, se mantienen los mismos ejemplos de forma que sea posible comparar ambos lenguajes mediante problemas ya conocidos.

8.4.2. Tratamiento de Variables y Tipos Básicos en Python.

El lenguaje Python, al igual que otros lenguajes de programación, impone restricciones sobre cómo nombrar a los objetos que se manipulan en los programas. Las reglas a seguir para construir *identificadores* son las siguientes:

1. Los identificadores pueden ser arbitrariamente largos,
2. Los identificadores pueden contener números o letras (o _), pero el primer carácter no puede ser un número,

[8] La transmitimos directamente y dejamos constancia de que... motivos haberlos, haylos ;-). Tampoco conviene olvidar que los compañeros entre los que hicimos la encuesta ya han aprobado Sistemas Operativos y Redes, entre otras asignaturas ;-)

3. Pueden utilizarse letras mayúsculas o minúsculas, teniendo en cuenta que Python es *case sensitive*, distingue entre carácteres en mayúscula y en minúscula.

De acuerdo a lo anterior, los siguientes identificadores no serían válidos: `15dias`, ya que empieza por dígitos; `mi-nombre` ó `aux$` tampoco, puesto que contienen carácteres especiales (sí que se admite, `mi_nombre` como identificador); `def` tampoco sirve como un identificador válido ya que es una *palabra reservada* del lenguaje, es decir, forma parte de su sintaxis.

De entre los tipos que se han denominado básicos, en Python hay tres. El entero, que se denomina `int`, el real, `float`, y la cadena, `string`. No tiene un tipo `boole` como tal[9], pero se identifican, cuando es necesario, el valor entero `0` con el valor `falso` y un valor distinto de 0 con el valor `cierto`.

Los valores de tipo `int` se representan con o sin signo, los de tipo `float` con o sin signo y utilizando el punto decimal o la notación científica, y los de tipo `string` entre comillas dobles, `""`, o apóstrofes, `''`, de forma indiferente.

Con la expresión `type()`, puede saberse de qué tipo es un valor. Por ejemplo, `type(17)` produce como respuesta `<type 'int'>`. El lenguaje Python es un lenguaje interpretado y no hay necesidad de declarar las variables antes de usarlas. Ello no quiere decir que no deba tener presente qué variables se están utilizando, para qué y de qué tipo deben ser. Esta instrucción puede ser útil en este sentido, especialmente cuando se trabaja con la terminal de intérprete[10].

En Python, como en `C`, la asignación es una expresión en la que primero se evalúa la parte izquierda y, posteriormente, se asigna sobre la parte derecha.

<identificador> = <expresión>

Se permite la coerción entre el tipo `int` y el tipo `float`; y, además, permite realizar conversiones entre los tipos, cuando ello es posible, utilizando las expresiones `int()`, `float()` y `str()`, que convierten a entero, real y cadena respectivamente, de la siguiente forma:

- `int("32")`, transformaría el string "32" en el entero `32`; pero, `int("Hola")` provocaría un error ya que no hay ningún valor entero representado por esa cadena.

  Cuando se hace la conversión desde `float`, `int` devuelve el entero obtenido al truncar el correspondiente real, `int(38.67) = 38`.

- `float("38.97")` transforma la cadena "38.97" en el real `38.97`; al igual que ocurre con `int` no admite otro tipo de cadena que las que se pueden asimilar a valores reales.

  `float(3)` devuelve el valor real `3.0`.

- Por último, `str(3)` devuelve la cadena "3" y `str(34.5)` devuelve "34.5".

Es importante asumir la diferencia que existe entre, por ejemplo, `3`, `3.0` y "3". Cada uno de los anteriores es un valor constante de distinto tipo.

Los operadores básicos de cada tipo coinciden, en general, con los habituales:

[9] Veáse sección 8.6.

[10] En este entorno, otra función útil es `dir()`, que devuelve la lista de identificadores definidos en un momento dado, los del usuario y los propios del sistema.

int: Cambio de signo (-), Suma (+), Resta (-), Multiplicación (*), Cociente de la división entera (/)[11], Resto de la divisón entera (%) y Exponenciación (**).

float: Cambio de signo (-), Suma (+), Resta (-), Multiplicación (*), División real (/) y Exponenciación (**).

Los operadores tienen un orden de precedencia: primero, se evalúa lo que vaya entre paréntesis. Después, las exponenciaciones. En tercer lugar van la multiplicación y la división. Por último, adiciones y substracciones. Se puede utilizar el mnenotécnico PEMDAS.

Cuando hay operadores de la misma preferencia, se realizan las operaciones evaluando de izquierda a derecha.

En relación al tipo entero, hay que comentar que existe la posibilidad de trabajar con enteros grandes (de aproximadamente unos 25000 dígitos), bien añadiendo a un entero el sufijo `L` o bien utilizando la función `long()`[12].

string: Concatenación (+) y Repetición (*). La repetición consiste en concatenar varias veces una misma cadena.

Por ejemplo, `("hip!"*2 + "hurra!")*3` daría como resultado

`"hip!hip!hurra!hip!hip!hurra!hip!hip!hurra!"`.

Por supuesto, a la hora de utilizar los operadores anteriores, hay que tener muy presente otra vez el concepto de *polisemia*, especialmente a la hora de distinguir entre los enteros y los reales. Tal y como ocurre en `C`, no es lo mismo la división entre dos valores enteros (`178/60` dará como resultado `2`) que entre dos reales o un entero y un real (`178/60.0` dará como resultado `2.97`), por ejemplo[13].

Aunque no trabaje explícitamente con el tipo `boole`, Python, como todos los lenguajes, necesita evaluar predicados. Los operadores conectores son `not`, `and` y `or` y se evalúan en este orden. Los operadores relacionales también son los habituales, y su sintaxis en Python es,

== (igual), != (distinto), $<$, $<=$ (menor o igual), $>$, $>=$ (mayor o igual)

Los operadores relacionales se evalúan antes que los conectores.

Hay más operadores, y se incorporarán sobre todo en el trabajo de laboratorio. Por ejemplo, para trabajar con el tipo `float` se dispone de todas las operaciones implementadas en la biblioteca `math`.

El tipo `string` dispone de operaciones que permiten conocer la longitud de una cadena, `len()`, consultar el valor del carácter i-ésimo de la cadena, indicando el valor de i entre corchetes, `[i]`, e incluso permite consultar subcadenas de una cadena con el operador *slice*, `[n:m]`, que devuelve los carácteres desde la posición `n` hasta la `m-1`. Para usarlo, se debe tener presente que Python comienza a numerar desde el valor `0`. Si no se indica un valor específico para `n`, se asume que es `0` y si no se indica un valor para `m`, se asume que coincide con `len()`.

Es decir, si `mi_nombre = "Moncho"`, entonces,

- `mi_nombre[0]` se refiere a la cadena `"M"`, `mi_nombre[-1]` a `"o"`,

[11] Veáse sección 8.6.
[12] Si bien en las últimas versiones de Python está conversión se realiza automáticamente en cuanto el intérprete detecta que un valor entero excede de rango.
[13] Veáse sección 8.6.

- `mi_nombre[1:4]` a la cadena `"onc"`,
- `mi_nombre[:5]` a la cadena `"Monch"` y
- `mi_nombre[2:]` a la cadena`"ncho"`.

Este operador sólo permite *consultar*. No es posible asignar un valor de un elemento de una cadena. Es decir,

```
mi_nombre[0] = "G"
```

no es una operación permitida en Python.

Entrada y salida de datos en Python.

Para realizar la entrada y salida de datos por teclado y pantalla, se dispone de las siguientes instrucciones:

Lectura de Datos Se dispone de dos mecanismos básicos,

- `raw_input()`, lee cualquier secuencia que se introduzca por teclado y asume que es un `string`. Por lo tanto, si se espera leer un valor entero o real habrá que utilizar convenientemente las operaciones que permiten cambiar de tipo.
- `input()`, es algo especial; por ahora, se asume que sólo permite leer enteros.

Ambas instrucciones admiten que se añada un mensaje, una cadena, lo que mejora la comunicación con el usuario.

Escritura de Resultados Para ello se dispone de la instrucción `print` a la que puede acompañar cualquier expresión válida en una asignación, es decir, `print 3`, `print mi_nombre` o `print minutos/60.0` serían correctos.

También admite la posibilidad de añadir mensajes para ayudar al usuario a enteder la información que recibe.

Como ejemplo del uso de estas instrucciones, y como ejemplo de esta sección, se puede completar el ejemplo del cálculo de la media aritmética de 4 enteros, que fue también el primer ejemplo en el lenguaje `C`:

media.py

```
# Lectura de datos
num1 = int(raw_input("Primer numero:"))
num2 = int(raw_input("Segundo numero:"))
num3 = int(raw_input("Tercer numero:"))
num4 = int(raw_input("Cuarto numero:"))
#Proceso
suma = num1 + num2
suma = suma + num3
suma = suma + num4
media = suma / 4.0 # Ojo, división real
# Escritura del resultado
print "La media es:", media
```

En el ejemplo se han utilizado comentarios: comienzan con el carácter # y finalizan con el salto de línea.

Cuando se ejecuta la acción,

```
num1 = int(raw_input("Primer numero:"))
```

ocurre lo siguiente: primero, se escribe la constante de tipo `string` `"Primer numero:"` en pantalla; se interrumpe la ejecución hasta que el usuario escribe un valor por teclado (preferiblemente, un valor entero o se produciría un error de ejecución). Una vez que el usuario finalice la introducción del valor, éste se toma como un `string` que es convertido a un valor entero, ya que se ha utilizado `int()`. Una vez se ha realizado la conversión, ya se puede realizar la asignación sobre el objeto `num1`. Las demás instrucciones `raw_input()` producirían un efecto similar sobre los demás datos.

Cuando se ejecuta la acción,

```
print "La media es:", media
```

se escribe la constante de tipo `string` `"La media es:"` en pantalla; a continuación, y sin cambiar de línea (ya que se ha utilizado una coma como separador), se escribiría el valor del objeto `media` en ese momento. También existe la posibilidad de utilizar caracteres de control que mejoren el aspecto del mensaje[14]; así, la instrucción anterior se podría haber escrito como:

```
print "La media es%f."% (media)
```

Es muy similar a la que se presentó con `C`: un carácter de control permite indicar la posición en la que se escribirá el valor de la variable. Para los distintos tipos, se utilizan distintos caracteres de control.

8.4.3. Estructuras de Control de Flujo de Datos.

Esquemas Condicionales en Python.

El *esquema condicional simple* o *ejecución condicional*, responde a la siguiente formulación,

if <predicado> :
    <instruccion>

La ejecución de este esquema es la ya conocida: se evalúa el predicado y si el resultado de su evaluación es `cierto`, se ejecuta la instrucción (o instrucciones) contenidas en el cuerpo del condicional (las indentadas tras el separador `':'`). Si no se cumple, el procesador la ignora y pasa a ejecutar la instrucción siguiente al condicional.

El *esquema condicional doble* o *ejecución alternativa* responde a la formulación

[14] Básicamente, coinciden con los de `C`; también es posible incluir caracteres especiales tales como `'\n'` o `'\t'`.

if <predicado> :
<instruccion_1>
else :
<instruccion_2>

En este caso, si se cumple el predicado se ejecuta la instrucción (o instrucciones) indentadas entre el delimitador `':'` y la palabra reservada `else`. Si no se cumple, las instrucciones que ejecuta el procesador son las indentadas después de los dos puntos, `:`, que hay tras la palabra reservada `else`.

Ejemplos del Uso de Esquemas Condicionales.

Tal y como se hizo en el tema 3 con `C`, el primer ejemplo que se propone es, dados dos objetos del mismo tipo, `A` y `B`, determinar cuál es el valor del máximo. Las dos versiones de la solución, son en Python:

maximo1.py
```
# Máximo de A y B
if (A>B):
   maximo = A
else:
   maximo = B
```

maximo2.py
```
# Máximo de A y B
maximo = A
if (B>maximo):
   maximo = B
```

El ejemplo en el que se debía calcular el precio de un pedido de paquetes de papel, sabiendo que el precio del paquete depende de la cantidad de paquetes comprada (2 euros, cuando la cantidad es menor de 200 paquetes; 1.8 euros, si la cantidad está entre 201 y 500 y 1.6 euros, si son más de 500), puede quedar de la siguiente forma:

precioPedido1.py
```
# CPP (entero): cantidad paquetes; total (real): precio total
if (CPP <= 200):
   total = CPP * 2.0
else:
   if (CPP <= 500):
      total = CPP * 1.8
   else:
      total = CPP * 1.6
```

Ejecución anidada en Python.

De nuevo, se propone como modificación añadir un tramo más de facturación en el problema del precio del pedido de folios:

Si el pedido es superior a los 800 paquetes, el precio del paquete es de 1.35 euros. Se añade este nuevo tramo a la solución anterior:

precioPedido2.py

```
# CPP (entero): cantidad paquetes; total (real): precio total
if (CPP <= 200):
   total = CPP * 2.0
else:
   if (CPP <= 500):
      total = CPP * 1.8
   else:
      if (CPP <= 800):
         total = CPP * 1.6
      else:
         total= CPP * 1.35
```

En Python, la indentación forma parte de la sintaxis; por lo tanto, la aparición de nuevos tramos puede contribuir a hacer difícil la tarea de leer el algoritmo. El lenguaje ofrece un condicional de ejecución anidada, más compacto que el de `C` y que, además, se puede utilizar también en aquellos casos en que se utiliza el `switch` en `C`:

precioPedido3.py

```
# CPP (entero): cantidad paquetes; total (real): precio total
if (CPP <= 200):
   total = CPP * 2.0
elif (CPP <= 500):
   total = CPP * 1.8
elif (CPP <=800):
   total = CPP * 1.6
else:
   total = CPP * 1.35
```

La palabra reservada `elif` proviene de la unión de `else` e `if`. No hay límite en cuanto al número de claúsulas `elif` a utilizar, pero es obligatorio que la última rama sea una claúsula `else`.

Cada una de las condiciones se comprueba en el orden en que aparece. Si una es cierta, se ejecuta la instrucción correspondiente y finaliza la ejecución del condicional. Si ninguna de las condiciones expuestas es cierta, se ejecuta la rama correspondiente al `else`. En este sentido, se puede comprobar que la lógica de este condicional anidado es exactamente igual que la correspondiente secuencia de condicionales anidados.

Hay que comentar, por último, una peculiaridad de Python en el uso de los predicados de control. Ya se dijo al introducir el cálculo de predicados que hay expresiones, como por ejemplo, $A > B > C$ o $0 < x \leq 100$, que son válidas en matemáticas pero que en lógica se deben considerar como una especie de abreviatura de los predicados compuestos $(A > B)$ `and` $(B > C)$ o $(0 < x)$ `and` $(x \leq 100)$, respectivamente.

El lenguaje Python sí que admite tales "abreviaturas" como predicados lógicos; pero no es muy recomendable acostumbrarse a su uso, ya que esto puede ser una fuente continua de errores sintácticos al trabajar con otros lenguajes.

Esquemas Iterativos en Python.

La forma general del bucle condicional en Python es la siguiente,

while <predicado> :
<instruccion>

Se evalúa el predicado de control *antes* de ejecutar ⟨instruccion⟩ y, si el resultado de evaluar ⟨predicado⟩ es `falso`, finaliza la ejecución del cuerpo del bucle y se pasa a ejecutar la instrucción siguiente; si es cierto, se ejecuta la instrucción (o instrucciones) contenidas en el cuerpo del bucle, es decir, las indentadas a partir del delimitador `':'`. Y, al finalizar la ejecución completa de todas las instrucciones que forman el cuerpo del bucle, se vuelve a evaluar la condición.

El número de veces que se repite la ejecución de la secuencia de acciones es desconocido de antemano (depende de la evaluación del predicado de control) y puede variar desde 0 a cualquier número de veces.

Ejemplos del Uso de Esquemas Iterativos.

El primer ejemplo de este tipo de esquemas en el tema 3, era el cálculo del cociente y el resto de la división entera mediante restas sucesivas.

En Python, este algoritmo se expresaría como:

divEntera.py

```
# Datos: dvdo (entero): dividendo
#        dvsor (entero): divisor
# Resultados: coc (entero): cociente div entera
#             resto (entero): resto div entera
resto = dvdo
coc = 0
while (resto >= dvsor):
    resto = resto - dvsor
    coc = coc + 1
```

El segundo ejemplo, permitía calcular la suma de todos los números enteros entre `1` y `n`. El algoritmo en Python es el siguiente:

sumaEnteros.py

```
# Datos: NUM (entero) límite superior del rango
# Resultados: resultado (entero) acumulador de sumas
# Variables: indice (entero) como índice
resultado = 0
indice = 1
while (indice <= NUM):
    resultado = resultado + indice
    indice = indice + 1
```

Bucles con Contador.

La versión alternativa para calcular la suma de los n primeros números enteros mediante un bucle con contador en Python, es la siguiente:

```
sumaEnteros2.py
# Datos: NUM (entero) límite superior del rango
# Resultados: resultado (entero) acumulador de sumas
# Variables: indice (entero) como índice
resultado = 0
for indice in range(1, NUM+1):
    resultado = resultado + indice
```

El esquema general del bucle con contador es el siguiente *cuando el paso es positivo*

```
for v in range(vi, vf+1, vp) :
    <instruccion>
```

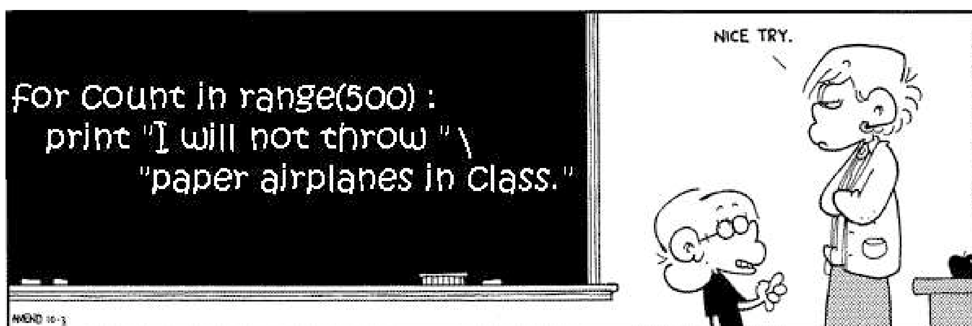
y

```
for v in range(vi, vf-1, vp) :
    <instruccion>
```

cuando el paso es negativo, por ejemplo:

```
for v in range(10, 0, -1) :
     <instruccion>
```

si se desea que el valor inicial de v sea 10 y el final 1.



El funcionamiento general es similar al descrito en el caso de C: la ejecución se repite tantas veces como se pueda añadir al valor inicial de v el valor del *paso* o incremento, vp, partiendo del valor inicial, vi, sin que se sobrepase el valor final, vf. Es decir, la primera iteración se realiza con v = vi; la segunda, con v = vi+vp y así, sucesivamente, hasta superar el valor final (y por eso se indica vf+1 en el bucle). En general, el valor de v variará según las fórmulas

- Si el paso es positivo: v = v + vp, $vi \leq v < vf+1$,

- Si el paso es negativo: `v = v + vp` (ojo, es negativo), `vi` $\geq$ `v` $>$ `vf-1`.

¿Por qué hay que trabajar con `vf+1` o `vf-1`?. Esto se debe a que al utilizar la expresión,

```
i in range(vi, vf+1, vp)
```

por ejemplo, lo que se usa es una operación propia del lenguaje; si bien el efecto es similar al de otros lenguajes en los que la variable `i`, efectivamente, va tomando los valores `vi`, `vi+vp`, etc., en Python lo que ocurre realmente es que `range` devuelve *todos* los valores comprendidos entre `vi` y `vf` en una estructura de datos[15] y el contador `i` realmente lo que hace es "recorrer" la estructura y tomar cada uno de los valores en ella contenidos.

Esta característica permite, además, que se pueda utilizar un objeto de tipo `string` para controlar un bucle `for`:

```
nombre = "Dolores Fuertes Cabeza"
for letra in nombre :
   print letra
```

El contador `letra` "recorre" la cadena almacenada en `nombre` y va tomando el valor de cada uno de sus carácteres.

8.4.4. Acciones no Primitivas en Python.

En Python, se pueden definir acciones no primitivas dentro del propio fichero del programa o en un fichero independiente. Si la definición se hace en el fichero del programa, se hará antes del propio programa. Si la definición se hace en un fichero distinto al del programa que utilizará la acción no primitiva (para definir un módulo, por ejemplo) el fichero del programa debe contener la línea,

`from` <fichero> `import` <identificador>

siendo <fichero>, el nombre del fichero `.py` que contiene a la acción <identificador>. Este fichero contiene la implementación de las acciones no primitivas, que sigue la siguiente sintaxis:

def <identificador> (lista_de_parametros_entrada):
Cuerpo de acciones INDENTADO
return [lista_de_parametros_salida]

Ambas listas de parámetros pueden tener 0, 1 o varios elementos. Para ver cómo se utiliza según el número de parámetros de salida, se estudiarán varios ejemplos. Se asume que todas las acciones no primitivas están definidas del fichero `ejemploT8.py`.

[15] Una lista, tal y como se verá en la próxima sección.

Un único parámetro de salida.

En este caso, sólo se devuelve un valor asociado al nombre de la acción no primitiva mediante `return`. Para llamar a la acción se usará el número adecuado de parámetros reales de entrada; el valor del parámetro de salida se identifica con el nombre. Por lo tanto, se puede usar la llamada en cualquier sitio donde se pueda usar una variable. A continuación, se muestran los mismos ejemplos que ya se vieron en el tema 4:

- Cálculo del factorial de `n`[16].

fact()
```
def fact(n):
  # Dato: n, entero positivo

  f = 1
  for i in range(2, n+1):
     f = f * i

  return f
  # Resultado: f=n!, entero positivo
```

Para utilizar esta función en el cálculo de las combinaciones de `m` elementos, tomados de `n` en `n`,

$$C_n^m = \frac{m!}{n! * (m-n)!} .$$

se haría lo siguiente:

combinaciones.py
```
from ejemploT8 import fact

m = int(raw_input('Valor de m:'))
n = int(raw_input('Valor de n:'))
comb = fact(m)/(fact(n)*fact(m-n))
print 'Combinaciones de %d, de %d en %d, son %d' %(m,n,n,comb)
```

- Determinar si un valor entero `n` es primo.

primo()
```
def primo(n):
   # Dato: n, entero positivo

   i = 2
   while ((i <= n/2) and (n%i != 0)):
      i = i + 1
   return (n%i != 0)
   # Resultado: cierto si n es primo, falso (0) si no lo es
```

tal y como ya se hizo en el tema 4, esta función se puede utilizar para escribir la tabla de todos los números primos entre `1` y `K`,

[16]El factorial devuelve valores muy grandes; podéis encontrar ejemplos en Python, en los que la primera asignación se formule como `f=1L`, para que devuelva un entero grande, si bien en las últimas versiones de Python esta conversión se realiza automáticamente. También podría ser conveniente cambiar el `for` y utilizar `for i in xrange(2,n+1)`, tal y como se justificará en la seccción 8.5.

primos.py
```
from ejemploT8 import primo

K = int(raw_input('Valor de K:'))
for i  in range(1, K+1):
   if (primo(i)):
      print i
```

- Suma de divisores propios de un entero, n.

sumaDiv()
```
def sumaDiv(n):
   # Dato: n, entero positivo

   suma = 1
   for i in range(2, (n/2)+1):
      if (n%i == 0):
         suma = suma + i
   return suma
   # Resultado: suma, suma de los divisores propios de n
```

Ejemplo de su uso para determinar si dos enteros, n1 y n2 son o no son amigos; se asume que esta función también se define en el fichero `ejemploT8`:

amigos()
```
def amigos(n1,n2):
   # Datos: n1 y n2, enteros positivos
   lo_son = 0
   if  (sumadiv(n1) == n2):
      lo_son = (n1 == sumadiv(n2))

   return lo_son
   # Resultado: lo_son, que vale falso (0) si n1 y n2 no
   # son amigos y que vale cierto si lo son
```

- Máximo Común Divisor de dos enteros, u y v.

  Si se utiliza el algoritmo de Euclides, se obtiene el siguiente algoritmo:

maxComDiv()
```
def maxComDiv(u,v):
   # Datos: u y v, enteros positivos
   while (u != v):
       if (u > v):
          u = u - v
       else:
          v = v - u
   mcd = u
   return mcd
   # Resultado: mcd, máximo común divisor de u y v
```

y, se puede utilizar, por ejemplo, para comprobar si dos números son primos entre sí; si lo son, su mcd es `1`:

primosEntreSi.py
```
from ejemploT8 import MaxComDiv
```

```
x1 = int(raw_input('Valor de x1:'))
x2 = int(raw_input('Valor de x2:'))
if (MaxComDiv(x1,x2) == 1):
   print 'Son primos entre sí.'
else:
   print 'No son primos entre sí.'
```

Más de un parámetro de salida.

En este caso, se devuelve más de un valor. Por eso se agrupan los parámetros formales de salida entre corchetes, `[ y ]`, después de `return`. Como hay más de un valor, la llamada se hace en una instrucción en la que también se agrupan entre corchetes los parámetros reales de salida a los que se "asigna" el resultado de la llamada.

- Cálculo del cociente y el resto de la división entera de A y B.

divEnt()

```
def divEnt(A,B):
    # Datos: A y B, enteros, A>=0, B>0
    coc = 0
    resto = A
    while (resto>=B):
      resto = resto - B
      coc = coc + 1

    return [coc, resto]
    # Resultados: coc=A/B,entero; resto=A%B,entero
```

Para utilizarla, el programa debe incluir entre sus líneas, las siguientes:

```
from ejemploT8 import divEnt
....

# las instrucciones que toque ...
...
[x, y] = DivEnt(345, 28)
...
```

- Una acción no primitiva que calcule el máximo común dividor y el mínimo común múltiplo de dos enteros. Ya se disponde de una función que calcula el mcd; por lo tanto, el mcm se puede calcular como el producto de los dos enteros, dividido por el mcd.

mcdMcm()

```
def  mcdMcm(A,B):
   # Datos: A y B, enteros positivos
   mcd = MaxComDiv(A,B)
   mcm = (A * B)/mcd

   return [mcd, mcm]
   # Resultados: mcd, mcd=mcd(A,B) y mcm, mcm=mcm(A,B),
   # enteros positivos
```

Para usarla:

usodeMcdMcm.py
```
from ejemploT8 import mcdMcm

xx = int(raw_input('Valor de xx:'))
yy = int(raw_input('Valor de yy:'))
[max,min] = mcdMcm(xx,yy)
print 'El mcd es ', max
print 'El mcm es ', min
```

- Lectura del valor de tres ángulos, verificando que su suma es igual a 180 grados:

lectura_correcta()
```
def  lectura_correcta():
    # Datos: no hay.
    # Los resultados se leerán de teclado, a1, a2 y a3, reales

    a1 = float(raw_input('Primer ángulo:'))
    a2 = float(raw_input('Segundo ángulo:'))
    a3 = float(raw_input('Tercer ángulo:'))
    while ((a1 + a2 + a3) != 180.0):
       print 'Incorrecto. Los ángulos deben sumar 180 grados'
       a1 = float(raw_input('Primer ángulo:'))
       a2 = float(raw_input('Segundo ángulo:'))
       a3 = float(raw_input('Tercer ángulo:'))

    return [a1, a2, a3]
    # Resultados: a1, a2 y a3, reales, a1+a2+a3=180.0
```

En la siguiente subsección se verá un ejemplo del uso, al combinarlo con otra acción no primitiva que clasifica un triángulo por el valor de sus ángulos.

Ningún parámetro de Salida.

No se devuelve ningún valor. Se hace la llamada en una línea, simplemente mediante el nombre y los parámetros de entrada.

- El ejemplo típico de esta situación, podría ser un procedimiento para visualizar por pantalla cómo utilizar un programa, que además, permite presentar una peculiaridad de Python a la hora de trabajar con cadenas:

info()
```
def info():
    # Datos: no hay.
    print """
    Uso de este programa:
           1) debe introducir las medidas de los segmentos.
           2) debe introducir un valor infinitesimal, epsilón.
           3) para finalizar, introduzca tres ceros. """
    # Resultados: no hay parámetros.
    # El efecto de esta acción se visualiza en la pantalla.
```

Cuando se escribe en pantalla una cadena como ésta, entre triples comillas, el resultado es que la cadena se escribirá tal y como aparece: incluirá los tabuladores y los saltos de línea.

- Clasificación de un triángulo según el valor de sus ángulos: hay que ver si hay alguno recto, y entonces es rectángulo, si todos son agudos, y entonces es acutángulo. En otro caso, es obtusángulo.

clasificacion()

```
def clasificacion(ang1, ang2, ang3):
    # Datos: ang1,ang2,ang3,reales,ang1+ang2+ang3=180.0
    if ((ang1 == 90.0) or (ang2 == 90.0) or (ang3 == 90.0)):
       print 'Es un triángulo rectángulo'
    elif ((ang1 < 90.0) and (ang2 < 90.0) and (ang3 < 90.0)):
       print 'Es un triángulo acutángulo'
    else:
       print 'Es un triángulo obtusángulo'

    # Resultados: no hay parámetros.
    # El efecto de esta acción se visualiza en la pantalla.
```

Para usarlo, se puede combinar con la acción no primitiva que lee tres ángulos comprobando que su suma es igual a 180 grados.

lectClasfTriang.py

```
from ejemploT8 import lectura_correcta, clasificacion

print 'Introduzca el valor de tres ángulos y \
               clasificaré el triángulo.'
[angulo1, angulo2, angulo3] = lectura_correcta()
clasificacion(angulo1, angulo2, angulo3)
```

Al introducir la barra, \, en la cadena de un `print`, se consigue la impresión de la cadena sin salto de línea.

Para finalizar esta subsección se comenta una utilidad de Python que permite documentar fácilmente las acciones definidas por el usuario. Consiste en que la primera sentencia del cuerpo puede ser una constante de tipo cadena, que se denomina *cadena de documentación* o *docstring*. Si es de varias líneas debe ir entre triples comillas y la primera línea sin comillas, marca la indentación a seguir.

mcdMcm()

```
def  mcdMcm(A,B):
   """ Dados A y B enteros
   devuelve su mcd y su mcm """
   # Datos: A y B, enteros positivos
   mcd = MaxComDiv(A,B)
   mcm = (A * B)/mcd

   return [mcd, mcm]
   # Resultados: mcd, mcd=mcd(A,B) y mcm, mcm=mcm(A,B),
   # enteros positivos
```

Si se ejecuta en el terminal la orden `print mcdMcm.__doc__` se visualiza la cadena de documentación (ojo, son dos símbolos _ seguidos).

8.5. Python: Gestión de la Información.

Durante la sección anterior, 8.4, se mantuvo el mismo esquema que en los temas del 2 al 4. Siguiendo con la misma idea, en esta debería mantenerse el esquema de los temas del 5 al 7. Sin embargo, los tipos compuestos del lenguaje Python no se corresponden exactamente con los tipos compuestos de C. Por lo tanto, se hará referencia a los tipos de C cuando ello sea posible, así como a los ejemplos ya conocidos.

Antes de pasar a presentar los tipos compuestos de Python, hay que comentar que, además del tipo entero y del real, Python dispone del tipo complejo, complex. Se puede definir una variable de tipo complejo bien utilizando la notación <real>+<imag>j, o bien invocando a la función complex(<real>,<imag>).

La aritmética es la habitual de este tipo. Además, se puede consultar el valor de la parte real y de la parte imaginaria y obtener el módulo mediante abs(), que devuelve un valor real:

```
>>> x=0.7+3.5j
>>> y=complex(1.3,2.5)
>>> y
(1.3+2.5j)
>>> print x*y
(-7.84+6.3j)
>>> print x/y
(1.21662468514+0.352644836272j)
>>> print x.real
0.7
>>> print y.imag
2.5
>>> print abs(x)
3.56931365951
```

8.5.1. Tipos Compuestos: Secuencias.

En Python no hay vectores ni registros. Hay diversos tipos compuestos que se conocen genéricamente como *secuencias*. De hecho, ya se conoce una: cualquier cadena es una secuencia, definida como una serie de carácteres contenidos entre comillas.

Hay cuatro tipos de secuencias en Python y cada una de ellas queda definida, precisamente, por el tipo de delimitador utilizado parar expresarlas. Los cuatro tipos son:

- La cadena, asociada a los delimitadores " y ", o ' y '. Como caso especial, se considera la *cadena vacía*: "" o ''.
- La lista, asociada a los delimitadores [y]. La *lista vacía* se expresa como [].
- La tupla, asociada a los delimitadores (y). En este caso, () define una *tupla vacía*.
- Y el diccionario, asociado a los delimitadores { y }. El *diccionario vacío* se expresa como {}.

Cualquier secuencia vacía equivale en Python al valor FALSO, exactamente igual que el 0.

Estos tipos se dividen en *mutables* o *inmutables*. En los tipos mutables, se puede cambiar el valor a elementos individuales de la secuencia y en los inmutables, no. La cadena, como ya se sabe, es inmutable, al igual que la tupla. Las listas son mutables. Por su parte, el diccionario es una caso especial. Tal y como se verá, un elemento de un dicccionario es un par formado por una *clave* y un *valor*: las claves han de ser inmutables y los valores son mutables.

Por último, hay que comentar también que hay una serie de operaciones en Python que son comunes a cualquier secuencia: la concatenación y la repetición, la indexación y el operador *slice* y la función `len()`, que ya se introdujeron al hablar de las cadenas.

Listas.

En Python, una lista es cualquier secuencia de valores *no necesariamente del mismo tipo* separadas por comas y escritas entre corchetes, `[` y `]`; por ejemplo, la ejecución de la sentencia

```
a=['caramelos',180.5,23.8,100,'guindillas']
```

crea una lista `a`, donde hay 5 valores de distinto tipo.

Las listas y las operaciones de secuencia: Las operaciones comunes sobre secuencias, al ser aplicadas sobre las listas producen los siguientes resultados:

- La indexación y el operador *slice*: `a[i]` devuelve el elemento i-ésimo de la lista `a` (sin olvidar que se empieza a numerar desde el `0`), y `a[n:m]` devuelve una lista que contiene los elementos del `n` al `m-1`.
- La concatenación y la repetición: es posible unir dos listas con el operador `+` y concatenar repetidamente una lista consigo misma con el operador `*`.
- La longitud: la llamada a la función `len(a)` devuelve el número de elementos de la lista `a`.

El efecto, por lo tanto, es similar al que se produce al trabajar con cadenas. Pero, por supuesto hay una gran diferencia: al ser un tipo *mutable*, la lista permite la asignación de valores a sus elementos. Las siguientes operaciones sirven como ejemplo, partiendo del valor que se dió a `a` en el ejemplo anterior:

- La ejecución de `a[3]=a[3]+4` produce como efecto que `a` pase a ser la lista

  `['caramelos',180.5,23.8,104,'guindillas']`.
- La ejecución de `a[0:2]=[1,12]` produce como efecto que `a` pase a ser la lista

  `[1,12,23.8,104,'guindillas']`.
- También puede asignarse la lista vacía, `[]`: el efecto es "borrar" toda la lista o sólo una porción:

  ```
  >>> a[1:3]=[]
  >>> a
  [1,104,'guindillas']
  ```

- También se puede utilizar el operador *slice* para añadir elementos en una posición determinada:

  ```
  >>> a[1:1]=['cielos', 23.5]
  >>> a
  [1,'cielos',23.5,104,'guindillas']
  ```

- Pero, ojo, que el efecto es distinto cuando se hace esto:

  ```
  >>> a[1:2]=['yo', 35]
  >>> a
  [1,'yo',35,23.5,104,'guindillas']
  ```

  Se ha añadido el segundo elemento, pero el primero ha sobreescrito al que existía en la lista. ¿Por qué? La referencia `a[i:i]` denota en general a la lista vacía (ya que obliga a considerar los elementos del `i` al `i-1`) y a esa "lista vacía" que comienza en la posición `i` se le está asignando una lista. La referencia `a[i:i+1]` denota a la sublista de un único elemento que comienza en la posición `i`; y a esa lista, en el ejemplo `a[1:2]`, se le está asignando otra.

- Como último ejemplo:

  ```
  >>> a[:0]=a
  >>> a
  [1,'yo',35,23.5,104,'guindillas',1,'yo',35,23.5,104,'guindillas']
  ```

  ¿Qué ocurre? (pista: recordad que si no se indica el valor de `n`, el operador *slice* asume que vale `0`).

Operaciones propias de acceso y manipulación de la lista: Los ejemplos anteriores son una muestra del "juego" que ofrece la asignación combinada con el uso del operador *slice*, pero no permiten trabajar de forma eficiente con las listas. Así, para eliminar elementos es mucho más recomendable utilizar la función `del`, que se usa bien en combinación con índices o bien en combinación con el operador *slice*.

Si se usa con un índice, `del a[i]`, se elimina el elemento i-ésimo de la lista `a`. Con un rango, `del a[n:m]` elimina los elementos del `n` al `m-1`. Y si se utiliza sin especificar elementos concretos, `del a`, elimina completamente la lista `a`.

Además, existen una serie de *métodos* propios para un *objeto* de tipo lista[17], y que se pueden considerar como funciones optimizadas sobre listas:

`append(x)` Añade a la lista el elemento `x` al final; es equivalente a `a[len(a):]=[x]`.

```
>>> a=[1,'yo',35, 23.5,104,'guindillas']
>>> a.append(8)
>>> a
[1,'yo',35,23.5,104,'guindillas',8]
```

`extend(L)` Concatena a una lista los elementos de la lista `L`; es equivalente a `a[len(a):]=L`.

```
>>> b=[1,2,3]
>>> a.extend(b)
>>> a
[1,'yo',35,23.5,104,'guindillas',8,1,2,3]
```

[17] No olvidéis que Python es un lenguaje orientado a objetos.

insert(i,x) Inserta el elemento x antes de la posición i. Por lo tanto, a.insert(0,x) inserta x al principio de la lista y a.insert(len(a),x) sería equivalente a a.append(x).

```
>>> a.insert(0,20)
>>> a
[20,1,'yo',35,23.5,104,'guindillas',8,1,2,3]
```

remove(x) Elimina el primer elemento cuyo valor sea x de la lista. Si no lo encuentra se produce un error.

```
>>> a.remove('guindillas')
>>> a
[20,1,'yo',35,23.5,104,8,1,2,3]
```

pop(i) Elimina el elemento que ocupa la posición i de la lista. Si no se indica el valor de i, elimina el último. En cualquier caso, devuelve el valor de dicho elemento.

```
>>> a.pop(2)
>>> a
[20,1,35,23.5,104,8,1,2,3]
>>> c=a.pop()
>>> a
[20,1,35,23.5,104,8,1,2]
>>> c
3
```

index(x) Devuelve el índice del primer elemento cuyo valor sea x en la lista. Si no lo encuentra se produce un error.

```
>>> d=a.index(104)
>>> d
4
```

count(x) Devuelve el número de elementos de la lista cuyo valor coincide con x de la lista. Por lo tanto, su uso combinado con remove() e index(), puede prevenir que se produzcan errores.

```
>>> i=a.count(104)
>>> if (i!=0):
...     a.remove(104)
...
>>> a
[20,1,35,23.5,8,1,2]
```

sort() Ordena por orden creciente los elementos de la lista.

```
>>> a.sort()
>>> a
[1,1,2,8,20,23.5,35]
```

reverse() Invierte la lista.

```
>>> a.reverse()
>>> a
[35,23.5,20,8,2,1,1]
```

Nótese que todas estas operaciones tienen como parámetros o un índice o un valor: al ser un tipo predefinido del lenguaje, no hay necesidad de preocuparse por la gestión dinámica de la memoria. Y, por lo tanto, no hay que acceder a los elementos mediante un puntero tal y como ocurría en `C`. Con esta salvedad, se puede reproducir el comportamiento de las listas tal y como se estudiaron en `C`, sin más que autoimponerse la restricción de utilizar valores del *mismo tipo base*.

La función `range()`: Una función característica de las listas en Python, que ya se comentó al presentar el bucle `for`, es la función `range()`. Su uso general

```
range(n)
```

produce como resultado una lista cuyos elementos son los enteros del `0` al `n-1`.

Con dos parámetros, `range(n,m)` produce una lista cuyos elementos son los enteros del `n` al `m-1`. Y se puede añadir un tercer parámetro, `range(n,m,incr)`, para que los valores se produzcan del `n` al `m-1`, pero con un incremento `incr`.

El resultado de `range()` siempre es una lista. Por eso, al ser utilizada por un bucle `for` es conveniente, en ocasiones, substituirla por `xrange()`: en este caso, el índice del bucle, toma todos los valores indicados pero sin que se cree la lista explícitamente.

De acuerdo a lo anterior, un esquema típico de recorrido exhaustivo de una lista puede ser el siguiente:

```
for i in range(len(a)):
    #tratar a[i]
```

Por ejemplo, si lo que se deseea es imprimir todos los elementos se puede hacer

```
for i in range(len(a)):
    print 'Elemento %d es ' %i,
    print a[i]
```

También posible hacer listas con elementos que sean listas. Por ejemplo:

```
>>> a=[1,2,3,4]
>>> b=[1, a, 10]
>>> b
[1,[1,2,3,4],10]
```

En este caso, se puede utilizar doble índice para acceder a los elementos de la lista que hay en el segundo elemento de `b`; por ejemplo, `b[1][0]` ó `b[1][2]`.

¿Cómo trabajar con vectores en Python? El lenguaje Python no tiene vectores, pero la indexación permite que se pueda simular fácilmente con listas el trabajo con este estructura. Para ello, es necesario autoimponerse dos restricciones, trabajar con *elementos del mismo tipo base* y *no modificar el número de elementos* que lo forman.

Los siguientes ejemplos ilustran esta idea.

- Algoritmo para leer un vector real v de tamaño n dado por teclado:

```
v=[0]*n  # crea un "vector" de n 0s
for i in range(n):
    v[i]=float(raw_input('Introduzca valor %d:%i))
```

- Algoritmo para calcular el mínimo de un vector v de tamaño n:

```
min=v[0]
for i in range(1,n):
    if (v[i]<min):
        min=v[i]
```

- Algoritmo para calcular cuántos elementos de un vector v de tamaño n, son mayores que su valor medio:

```
med=0.0
for i in range(n):
    med=med+v[i]
med=med/n
cont=0
for i in range(n):
    if (v[i]>media):
        cont=cont+1
```

- Algoritmo para sobreescribir los elementos nulos de un vector v de tamaño n, con la media de los elementos anteriores:

```
acum=v[0]
for i in range(1,n):
    if (v[i]==0):
        v[i]=acum/i
    acum=acum+v[i]
```

Si se anidan “vectores” del mismo tamaño en otro “vector”, se podría simular el trabajo con una matriz. Por ejemplo, para leer sus valores y después mostrarlos por pantalla, se podría hacer lo siguiente:

```
# se dan valores a n (columnas) y m (filas)
M=[]
for i in range(m):
    M.append([0]*n) # crea una "matriz" m*n
for i in range(m):
    for j in range(n):
        M[i][j]=float(raw_input('M[%d][%d]='%(i,j)))
for i in range(m):
    for j in range(n):
        print '%5.2f'% M[i][j] ,
    print
```

Tuplas y Diccionarios.

Las **tuplas** son secuencias de valores entre paréntesis. Son inmutables, por lo que no es posible acceder a uno de sus elementos para asignarles un nuevo valor. Admite los operadores comunes a las otras secuencias y también permite el anidamiento.

```
1  >>> u=('uno',1,1.0)
2  >>> d=('dos',2,2.0)
3  >>> t=('tres',3,3.0)
4  >>> u+d+t
5  ('uno',1,1.0,'dos',2,2.0,'tres',3,3.0)
6  >>> todos=(u,d,t)
7  >>> todos
8  (('uno',1,1.0),('dos',2,2.0),('tres',3,3.0))
9  >>> todos[1]
10 ('dos',2,2.0)
11 >>> t[0:2]
12 ('tres',3)
13 >>> todos[0:1]
14 (('uno',1,1.0),)
```

En el ejemplo, se llama la atención sobre las líneas 9, una indexación, y 13, un rango de `0` a `0`, y el distinto efecto que producen. La indexación devuelve un elemento de una secuencia; en este caso, el segundo elemento de la tupla `todos`. El operador *slice*, devuelve siempre una subsecuencia; en este caso, la subtupla formada por los elementos indicados... como sólo hay uno, se devuelve una tupla de longitud `1`. Nótese que la línea 14 también ilustra que las tuplas de un sólo elemento están formadas por dicho elemento *seguido de coma*.

Por supuesto, de lo anterior se puede deducir que el concepto de tupla en Python no tiene absolutamente nada que ver con el concepto que se ha venido manejando en la asignatura y que en `C` se concretaba con la definición de un `struct`. Esta nueva forma de ver la tupla estaría más relacionada con la estructura del mismo nombre que se maneja en un entorno de bases de datos.

Como ya se dijo, los **diccionarios** están formados por pares de *clave* y *valor*. Esto los caracteriza como la única secuencia que no tiene sus elementos indexados por la posición, sino que están indexados por la clave. Además, tienen la característica de poseer claves inmutables y valores mutables[18]. Su indexación por claves, provoca también que no pueda utilizarse el operador *slice*.

El siguiente es un ejemplo de uso de un diccionario:

```
1  >>> tlfs={'miguel':8289,'mar':8288,'angeles':8352,'gloria':8289}
2  >>> tlfs['gloria']        # consulto valor
3  8289
4  >>> tlfs['gloria']=8299   # asigno valor
5  >>> tlfs
6  {'gloria':8299,'mar':8288,'miguel':8289,'angeles':8352}
7  >>> tlfs['javier']=8264   # añado clave y valor
8  >>> tlfs
9  {'gloria':8299,'mar':8288,'javier':8264,'miguel':8289,'angeles':8352}
```

[18]Por lo tanto, siempre se pueden usar cadenas o números como claves; nunca pueden utilizarse listas y sólo se pueden utilizar tuplas si estas no contienen algún elemento mutable.

```
10 >>> tlfs.keys()
11 ['gloria','mar','javier','miguel','angeles']
12 >>> del tlfs['miguel']    # elimino clave y valor
13 {'gloria':8299,'mar':8288,'javier':8264,'angeles':8352}
14 >>> tlfs.has_key('miguel')
15 0
```

Las líneas 2 y 4 son ejemplos de cómo utilizar la indexación por claves para acceder o asignar un valor a un elemento de un diccionario. En la línea 7, se añade un nuevo elemento, mientras que en la 12, usando `del` se elimina. Las líneas 10 y 14 muestran un ejemplo de los *métodos* de un *objeto* diccionario `keys()` y `has_key()`; el primero, devuelve una lista de las claves y el segundo permite consultar si una clave determinada pertenece o no al diccionario. Son útiles para evitar acceder a elementos mediante claves que no existan, lo que provoca un error

También existe una función `dict()` que permite construir un diccionario desde una lista de pares clave/valor almacenados como tuplas; es decir, el diccionario del ejemplo anterior podría haberse creado también de la siguiente forma,

```
tlfs=dict([('miguel',8289),('mar',8288),('angeles',8352),('gloria',8289)])
```

Se puede obtener un listado completo de todos los elementos de un diccionario, mediante el *método* `items()`:

```
>>> for c,v in tlfs.items():
...     print c, v
...
gloria 8299
mar 8288
javier 8264
angeles 8352
>>> # es equivalente a
>>> for clave in tlfs.keys():
...    print clave, tlfs[clave]
...
gloria 8299
mar 8288
javier 8264
angeles 8352
```

El diccionario es una estructura de datos muy especial: algunos lenguajes lo conocen como *memoria asociativa*, pero es más habitual la expresión *tabla hash* (seguro que oiréis esta expresión en cursos posteriores).

¿Cómo trabajar con vectores de `structs` en Python? Se ha visto que pueden definirse listas cuyos elementos sean listas. En general, cualquier secuencia puede anidarse dentro de una lista: por lo tanto, se pueden considerar distintas combinaciones.

Podría ser útil una lista (jugando de "vector") cuyos elementos fueran listas, todas con la misma estructura de tipos (jugando de "`structs`"). Por ejemplo, la definición hecha en `C`:

```
typedef struct {
  char nombre[80];
  char telefono[15];
  int edad;
} tPersona;
```

que luego permite definir un vector `alumnos` de ese tipo base, se podría traducir a Python como una lista de listas de la siguiente forma:

```
alumnos=[['pepe','999666',19],['jose','666999',18],['pepin','969696',18]]
```

La información sería accesible para consulta y asignación, sin más que tener en cuenta como funciona la indexación:

```
>>> print alumnos[0][2]
19
>>> alumnos[1][1]='666666'
>>> alumnos
[['pepe','999666',19],['jose','666666',18],['pepin','969696',18]]
```

También podría pensarse en una tupla de Python para simular la `struct` de `C`: son inmutables, pero una tupla que sea un elemento de una lista, se puede reasignar completamente. Si se parte de la situación:

```
alumnos=[('pepe','999666',19),('jose','666999',18),('pepin','969696',18)]
```

y se quiere modificar como antes el número del segundo elemento de `alumnos`, se podría hacer lo siguiente:

```
>>> alumnos[1]=('jose','666666',18)
>>> alumnos
[('pepe','999666',19),('jose','666666',18),('pepin','969696',18)]
```

Pero quizás la notación más similar a la que habitualmente se maneja, es la que permiten los diccionarios: una lista de diccionarios permite acceder mediante índices a los elementos del "vector" y en cada diccionario podrían usarse las claves para acceder a un elemento, de forma similar a como se usan los identificadores de los campos de un `struct`:

```
>>> uno={'nombre':'pepe', 'tlf': '999666', 'edad': 19}
>>> dos={'nombre':'jose', 'tlf': '666999', 'edad': 18}
>>> tres={'nombre':'pepin', 'tlf': '969696', 'edad': 18}
>>> alumnos=[uno,dos,tres]
>>> alumnos[1]['tlf']='666666'
>>> alumnos[1]
{'nombre':'jose', 'tlf': '666666', 'edad': 18}
```

El uso de la clave resulta más significativo que el segundo índice, tal y como ocurre con los identificadores de campo de un `struct`.

8.5.2. Ficheros en Python.

El lenguaje Python permite manejar ficheros de texto. Al igual que en otro lenguajes, el protocolo de manejo de ficheros supone primero, *abrir* el fichero, *trabajar* con los datos en él almacenados y, finalmente, *cerrar* el fichero (para preservar su integridad).

En la apertura, que sigue la siguiente sintaxis,

`f=open('fichero', 'modo')`

la cadena `'fichero'` se refiere, evidentemente, al nombre del fichero a abrir y la cadena `'modo'` se refiere a una de estas tres opciones: o bien es la cadena `'r'`, leer, o la cadena `'w'`, escribir, o la cadena `'a'`, añadir. Si no se indica ningún modo, la opción por defecto será `'r'`.

En `C`, al abrir un fichero se le asocia un descriptor, un puntero a una `struct` de tipo `FILE`; en Python la asignación anterior tiene como efecto definir un *objeto*, `f`, a través del cual se accede a la información contenida en el fichero.

Si el fichero se ha abierto para lectura, se pueden utilizar los siguientes *métodos*:

- `f.read(cantidad)` Se lee del fichero la `cantidad` de bytes indicada; teniendo en cuenta que el fichero es de texto, suele ser habitual indicar `1`, y leer, por lo tanto, carácter a carácter.

  Se puede utilizar con un valor negativo e, incluso, sin indicar ninguna cantidad. Pero se debe tener en cuenta que, en ambos casos, se lee *el fichero completo* (por lo tanto, si el fichero es muy extenso los riesgos son evidentes: es posible agotar la memoria del computador o, como poco, despilfarrarla).
- `f.readline()` Se lee del fichero una línea.

Cuando se alcanza el final del fichero, tanto `f.read` como `f.readline` devuelven la cadena vacía.

Si el fichero se ha abierto para escritura, el *método* `f.write(cadena)`, escribe la cadena especificada en el fichero. Si se desea que dicha cadena sea una línea (para leer posteriormente la información utilizando `readline()`, por ejemplo), la cadena debe incluir el carácter de control `'\n'`.

Y, por supuesto, no hay que olvidar utilizar el siguiente *método*, `f.close()`, que, tal y como se puede suponer, permite cerrar el fichero una vez se ha procesado la información.

Como ejemplo del uso de ficheros se retoma el ejemplo desarrollado en `C` en el tema 7: la definición de dos funciones, una que permita guardar los datos de una lista en un fichero y otra que permita recuperarlos y almacenarlos de nuevo en una lista. Cada dato de la lista se asume que es a su vez una lista con una cadena `nombre`, una cadena `tel` (teléfono) y un valor entero `edad`.

guardaLista.py

```
def guardaLista (lista,fichero):
# pre:  lista es una lista ya creada y fichero es una cadena
# con un identificador de fichero válido para escritura

    f = open(fichero , "w")

    for i in range(len(lista)):
        # por cada dato, una nueva línea
```

```
        # para facilitar la lectura con readline
        f.write(lista[i][0]+'\n')
        f.write(lista[i][1]+'\n')
        f.write(str(lista[i][2])+'\n')

    f.close()

# post: el fichero contiene los datos de personas
# incluidas en la lista, de forma que hay un dato
# por línea
```

recuperaLista.py

```
def leePersonas (fichero):
# pre:  fichero es una cadena que identifica a un
# fichero en el que se ha almacenado datos de una
# persona, un dato por línea

    personas=[]
    f = open(fichero , "r")
    nombre = f.readline()
    while (nombre!=''):
        tel = f.readline()
        edad = f.readline()
        # se elimina el "\n"
        nombre = nombre[:-1]
        tel = tel[:-1]
        edad = edad[:-1]
        # y se almacenan en la lista
        personas.append([nombre,tel,int(edad)])
        nombre = f.readline()
    f.close()
    return personas

# post: la lista personas está formada por los datos de persona
# almacenados inicialmente en fichero. Hay tantos elementos
# en la lista como líneas en el fichero dividido entre 3
```

8.6. Python: Actualizaciones.

La primera versión de estos apuntes se escribió en el curso 2002/2003. Desde que se escribieron han sido distribuidas nuevas versiones del lenguaje Python; la última es la versión 2.5, que aportan nuevas características (pero que no nos afectan si tenemos en cuenta que en esta asignatura aún somos un poco “novatos” - eso sí, lo que sabemos hacer lo hacemos muy bien -).

Si se repasa lo que anteriores *realeases* modificaban el contenido de las secciones anteriores, cabe reseñar que la versión 2.2 incorporó el operador `//` para el cálculo del cociente de la división entera. El comportamiento del operador `/` será el descrito en la sección 8.4 hasta la versión 3.0, en la que sólo denotará a la división real.

La versión 2.3 incorporó el tipo `bool`: las constantes `True` y `False` aparecen ahora donde antes se trabajaba con `1` y `0`. La función `bool()` realiza una conversión de tipo, de forma que `0` o

cualquier secuencia vacía es `False` y cualquier valor numérico distinto de `0` y cualquier secuencia no vacía es `True`. Y la 2.4 introdujo un nuevo tipo, el `decimal`, que unifica definitivamente los tipos entero y entero largo, tiene nuevos métodos para cadenas y el iterador `reversed`, entre otras características nuevas.

8.7. Agradecimientos.

Nuestro compañero, José Luis Llopis, nos ayudó a presentar de forma coherente los conceptos básicos de la Programación Orientada a Objetos. Y nuestros compañeros Andrés Marzal Varó e Isabel Gracia Luengo, han escrito un libro sobre Python [19] que ha facilitado mucho la elaboración de estos apuntes.

De entre los alumnos que fueron “acosados”, hay que destacar las fructíferas discusiones mantenidas con José Traver Ardura, Luis Peralta Nieto, Jordi Paraire Andrés y, especialmente, con Daniel Gómez Béjar.

8.8. Glosario.

API, *Application Program Interface* Método específico preescrito para ser utilizado en un S.O., o una determinada aplicación, que un programador debe utilizar para desarrollar nuevos programas, intercambiando información con dicho S.O., o con la aplicación.

applet Un pequeño programa que puede ser enviado junto con una página web a un usuario; normalmente, estará escrito en Java. Permite la ejecución de pequeñas tareas en la máquina del usuario, sin necesidad de reenviar una solicitud al servidor web.

atributo En POO, aquellos elementos de una clase cuyos valores definirán el estado del objeto (similar al concepto de “elemento de una estructura de datos” o, simplemente, un dato).

CGI, *Common Gateway Interface* Cuando un usuario solicita una página web, el servidor de web se la envía; sin embargo, cuando un usuario rellena un formulario en una página web y lo envía, está enviando información que debe ser procesada. El servidor de web traslada dicha información a una aplicación adecuada, devolviéndose un mensaje de confirmación.

El método que permite este intercambio de mensajes entre servidor y aplicación, es el CGI y forma parte del *Hypertext Transfer Protocol*, HTTP.

clase Entidad básica en POO: en ella se definen tipos de datos y los métodos que permiten manipularlos, además de indicar cuáles son de acceso público y cuáles no.

encapsular Asociar en una sola entidad datos que por naturaleza deben ir juntos y sus operaciones de manipulación. Será más útil cuanto más facilite el proceso de abstracción y permita al usuario no depender de la implementación de dicha entidad. Los lenguajes que siguen el POO garantizan el encapsulamiento, mientras que otros lenguajes (como `C`) sólo permiten simularlo.

escritorio Un área de visualización en un computador que representa los elementos habituales de un escritorio real: documentos, una agenda telefónica, herramientas de escritura, carpetas de proyectos... Puede ser parte de una ventana o puede ocupar toda la pantalla. Es posible (un ejemplo lo tenéis en KDE) tener varios escritorios y cambiar de uno a otro.

[19] Que podéis encontrar en `http://marmota.dlsi.uji.es/MTP/pdf/python.pdf`.

GUI, *Graphical User Interface* Interfaz gráfica de comunicación entre usuario y computador; el término surgió para diferenciar este tipo de interactuación con la de un entorno de consola. Las primeras fueron desarrolladas en el laboratorio de investigación de Xerox en Palo Alto, en la década de los 70.

`JSP`, *Java Server Page* Tecnología que permite controlar el contenido o la apariencia de páginas web, mediante el uso de *servlets*, pequeños programas especíificados en la página web y que se ejecutan en el servidor para modificarla antes de enviarla al usuario.

mensaje En POO, mecanismo de comunicación entre objetos (similar al concepto de "realizar una llamada a una función o procedimiento").

método En POO, código que efectúa alguna acción (como tal, similar al concepto de "función" o de "procedimiento").

objeto Una definición informal sería la de "aquello en lo que uno debe pensar primero para diseñar una aplicación dentro del POO"; es decir, la aplicación se diseña teniendo en mente qué objetos se deben manipular y cómo.

Formalmente, un objeto es una instancia sobre una clase.

script Hay dos definiciones para este término. La primera, más general, define el *script* como una secuencia de instrucciones que han de ser interpretadas o procesadas por otro programa, y no por el procesador del computador (a diferencia de lo que ocurre con un programa compilado).

La acepción más común es la que ofrece la segunda definición: una lista de comandos del sistema operativo, almacenados en un fichero y que son invocadas para ser ejecutados en secuencia por el intérprete de comandos del S.O. utilizando el nombre de dicho fichero como un nuevo comando.

widget Elemento de una GUI, que o bien permite visualizar información o bien proporciona al usuario una posibilidad de interactuar: iconos, botones, cajas de selección, indicadores de progreso, ventanas, esquinas de redimensionado de las ventanas, menús de selección por marcaje, barras de scroll, etc.

También se refiere al programa escrito para describir cómo debe ser la apariencia de uno de estos elementos y cómo deben comportarse.

8.9. Bibliografía.

1. "Guía de Aprendizaje de Python, Release 2.4.1a0", G. Van Rossum, F. L. Drake, editor, BeOpen PyhtonLabs.

    Esta referencia la encontraréis en la página oficial de documentación de Python en castellano, que es

    `http://pyspanishdoc.sourceforge.net/`

    y la página web de Python es:

    `http://www.python.org`

    donde podréis consultar, por ejemplo, todas las diferencias que se vayan produciendo entre lo que se cuenta aquí y las nuevas versiones de Python. Ojo, que a día de hoy (Octubre de 2006... o sea, que más bien es "a día de ayer":-) ya está la versión 2.5. En cualquier caso, la versión más actualizada del correspondiente tutorial suele estar aquí: `http://docs.python.org/tut/tut.html`

2. "How to think like a computer scientist", A. B. Downey, J. Elkner & C. Myers, Open Book Project. 2002.

   `http://www.ibiblio.org/obp/books`

3. "Introducción a la programación (vol.1)", A. Marzal Varó, I. Gracia Luengo. 2003.

   `http://marmota.dlsi.uji.es/MTP/pdf/python.pdf`