Departamento de Física, Facultad de Ciencias, Universidad de Chile.
Las Palmeras 3425, Ñuñoa. Casilla 653, Correo 1, Santiago
FONO: 562 978 7276 FAX: 562 271 2973
E-MAIL: secretaria@fisica.ciencias.uchile.cl

Apuntes de un curso de

PROGRAMACIÓN

Décima primera edición, revisión 111208-02

José Rogan C.
Víctor Muñoz G.

Agradecimientos:

A Xavier Andrade.
A Denisse Pastén.
De la clase del 2004 a: Daniel Asenjo y Max Ramírez.
De la clase del 2005 a: Alejandro Varas y María Daniela Cornejo.
De la clase del 2006 a: Nicolás Verschueren y Paulina Chacón,
Sergio Valdivia y Elizabeth Villanueva.
De la clase del 2007 a: Sebastián Godoy, Carola Cerda,
Rodrigo Pedrasa y Felipe Fuentes.
De la clase del 2008 a: Smiljan Vojkovic,
Víctor Araya y Juan Ignacio Pinto.
De la clase del 2009 a: María José Tapia,
Elizabeth Ambrosetti.
De la clase del 2010 a: Carlos Yañez,
Francisca Pino.

Índice

Índice de figuras

Capítulo 1

Elementos del sistema operativo UNIX.

versión revisada 8.0, diciembre de 2011

1.1. Introducción.

En este capítulo se intentará dar los elementos básicos para poder trabajar en un ambiente UNIX. Sin pretender cubrir todos los aspectos del mismo, nuestro interés se centra en entregar las herramientas al lector para que pueda realizar los trabajos del curso bajo este sistema operativo. Como comentario adicional, conscientemente se ha evitado la traducción de gran parte de la terminología técnica teniendo en mente que la documentación disponible se encuentra, por lo general, en inglés y nos interesa que el lector sea capaz de reconocer los términos.

El sistema operativo UNIX es el más usado en investigación científica, tiene una larga historia y muchas de sus ideas y métodos se encuentran presentes en otros sistemas operativos. Algunas de las características relevantes del UNIX moderno son:

- Multitarea (*Multitasking*): Cada programa tiene asignado su propio "espacio" de memoria. Es **imposible** que un programa afecte a otro sin usar los servicios del sistema operativo. Si dos programas escriben en la misma dirección de memoria cada uno mantiene su propia "idea" de su contenido.

- Multiusuario: Más de una persona puede usar la máquina al mismo tiempo. Programas de otros usuarios continúan ejecutándose a pesar de que un nuevo usuario entre a la máquina.

- Memoria grande, lineal y virtual: Un programa en una máquina de 32 Bits puede acceder y usar direcciones hasta los 4 GB en una máquina de sólo 4 MB de RAM. El sistema sólo asigna memoria auténtica cuando le hace falta, en caso de falta de memoria de RAM, se utiliza el disco duro (*swap*).

- Casi todo tipo de dispositivo puede ser accedido como un archivo.

- Existen muchas aplicaciones diseñadas para trabajar desde la línea de comandos. Además, la mayoría de las aplicaciones permiten que la salida de una pueda ser la entrada de otra aplicación.

- Permite compartir dispositivos (como disco duro) entre una red de máquinas.

Por su naturaleza de multiusuario, **nunca** se debería apagar impulsivamente una máquina UNIX[1], ya que una máquina apagada sin razón puede matar trabajos de días, perder los últimos cambios de tus archivos e ir degradando el sistema de archivos en los dispositivos de almacenamiento como los discos duros.

Entre los sistemas operativos UNIX actuales cabe destacar:

- Linux fue originalmente desarrollado primero para computadores personales PCs basados en x86 de 32 bits (386 o superiores). Hoy Linux corre sobre Intel x86, Alpha AXP, Sun SPARC, Motorola 68000, PowerPC, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, AMD x86-64, AXIS CRIS, Renesas M32R, Atmel AVR32, Renesas H8/300, NEC V850, Tensilica Xtensa y arquitecturas Analog Devices Blackfin; para muchas de estas arquitecturas en variantes de 32 y 64 bits.

- SunOS[2]: disponible para la familia 68K así como para la familia SPARC de estaciones de trabajo SUN

- Solaris[3] : disponible para la familia SPARC de SUN así como para la familia x86.

- OSF1[4]: disponible para Alpha.

- Ultrix: disponible para VAX de Digital

- SYSVR4[5]: disponible para la familia x86, vax.

- IRIX: disponible para MIPS.

- AIX[6]: disponible para RS6000 de IBM y PowerPC.

[2]SunOS 4.1.x también se conoce como Solaris 1.
[3]También conocido como SunOS 5.x, solaris 2 o Slowaris :-).
[4]También conocido como Dec Unix.
[5]También conocido como Unixware y Novell-Unix.
[6]También conocido como Aches.

1.2. Historia de Linux

1.2.1. Antecedentes

El núcleo Linux ha sido marcado por un crecimiento constante en cada momento de su historia. Desde la primera publicación de su código fuente en 1991, nacido desde un pequeño número de archivos en lenguaje C bajo una licencia que prohíbe la distribución comercial, a su estado actual de cerca de 296 MiBs de fuente bajo la Licencia pública general de GNU.

El nombre Linux ha generado gran controversia y nuevas alternativas no han tardado en aparecer para hacerle competencia.

En 1983 Richard Stallman inició el Proyecto GNU, con el propósito de crear un sistema operativo similar y compatible con UNIX y los estándares POSIX. Dos años más tarde creó la Fundación del Software Libre (FSF) y desarrolló la Licencia pública general de GNU (GNU GPL), para tener un marco legal que permitiera difundir libremente el *software*. De este modo el software de GNU fue desarrollado muy rápidamente, y por muchas personas. A corto plazo, se desarrolló una multiplicidad de programas, de modo que a principios de los 90 había bastante software disponible como para crear un sistema operativo completo. Sin embargo, todavía le faltaba un núcleo.

Esto debía ser desarrollado en el proyecto GNU Hurd, pero Hurd demostró desarrollarse muy inactivamente, porque encontrar y reparar errores (debugging en inglés) era muy difícil, debido a las características técnicas del diseño del micronúcleo.

Otro proyecto de sistema operativo software libre, en los años 1980 fue BSD. Este fue desarrollado en la Universidad de Berkeley desde la 6ª edición de Unix de AT&T. Puesto que el código de AT&T Unix estaba contenido en BSD, AT&T presentó una demanda a principios de los años 1990 contra la Universidad de Berkeley, la cual redujo el desarrollo de BSD. Así, a principios de los años 90 no produjo ningún sistema completo libre.

El futuro de BSD era incierto debido al pleito y detuvo el desarrollo. Además, el Proyecto GNU gradualmente se desarrollaba pero, este carecía de un bien formado núcleo UNIX. Esto dejó un nicho abierto, que Linux llenaría muy pronto.

1.2.2. Aparición de Linux

En 1991, en Helsinki, Linus Torvalds comenzó un proyecto que más tarde llegó a ser el núcleo Linux. Esto fue al principio un emulador terminal, al cual Torvalds solía tener acceso en los grandes servidores UNIX de la universidad. Él escribió el programa expresamente para el hardware que usaba, e independiente de un sistema operativo, porque quiso usar las funciones de su nueva computadora personal con un procesador 80386. El sistema operativo que él usó durante el desarrollo fue Minix, y el compilador inicial fue el GNU C, que aún es la opción principal para compilar Linux hoy (aunque Linux puede ser compilado bajo otros compiladores, tal como el Intel C Compiler).

Como Torvalds escribió en su libro "Solamente por diversión", él tarde o temprano com-

prendió que había escrito un núcleo de sistema operativo. El 25 de agosto de 1991, anunció este sistema en un envío a la red Usenet, en el newsgroup (grupo de noticias):

> Hola a todos aquellos que usan Minix -
>
> Estoy haciendo un sistema operativo (gratuito) (solamente una afición, no será grande ni profesional como el GNU) para clones 386(486) AT. Este ha estado gestándose desde abril, y está comenzando a estar listo. Me gustaría recibir cualquier comentario sobre las cosas que gustan/disgustan en minix, ya que mi SO (Sistema Operativo) se le parece un poco (la misma disposición física del sistema de archivos, debido a motivos prácticos, entre otras cosas).
>
> Actualmente he portado bash (1.08) y gcc (1.40), y las cosas parecen funcionar. Esto implica que conseguiré algo práctico dentro de unos meses, y me gustaría saber qué características quiere la mayoría de la gente. Cualquier sugerencia es bienvenida, pero no prometeré que las pondré en prÃ¡ctica :-)
>
> Linus Benedict Torvalds (torvalds@kruuna.helsinki.fi)
>
> PD. Sí, es libre de cualquier código de minix, y tiene un sistema de archivos multihilo. NO es portable (usa 386 una conmutación de tarea etc.), y probablemente nunca será soportada por nada más que los discos duros AT, porque es todo lo que tengo :-(.
>
> Linus Torvalds

1.2.3. El Nombre

Linus Torvalds había querido llamar su invención Freax, una combinación de *freak* (anormal o raro), *free* (libre), y "X", una alusión a Unix. Durante el inicio de su trabajo sobre el sistema, él almacenó los archivos bajo el nombre "Freax" por aproximadamente medio año. Torvalds ya había considerado el nombre "Linux" , pero al principio lo había descartado por ser demasiado egocéntrico o egoísta.

Para dar a otra gente la capacidad de cooperar en el sistema o sugerir mejoras, los archivos fueron colocados en el servidor ftp (ftp.funet.fi) de la Universidad de Helsinki, en septiembre de 1991. Ari Lemmke, colega de Torvalds en la Universidad, y encargado de los servidores en ese momento, no estuvo de acuerdo con el nombre Freax, prefiriendo el nombre Linux. Él simplemente llamó los archivos colocados sobre el servidor "Linux" sin consultar a Torvalds. Más tarde, sin embargo, Torvalds accedió a usar el nombre "Linux".

1.2.4. GNU/Linux

La designación "Linux" al principio fue usada por Torvalds sólo para el núcleo. El núcleo fue, sin embargo, con frecuencia usado junto con otro software, especialmente con el del

proyecto de GNU. Esta variante de GNU rápidamente se hizo la más popular, ya que no había ningún otro núcleo libre que funcionara en ese tiempo. Cuando la gente comenzó a referirse hacia esta recopilación como "Linux", Richard Stallman, el fundador del proyecto de GNU, solicitó que se usara el nombre GNU/Linux, para reconocer el rol del software de GNU. En junio de 1994, en el boletín de GNU, Linux fue mencionado como un "clon libre de UNIX", y el Proyecto Debian comenzó a llamar a su producto GNU/Linux. En mayo de 1996, Richard Stallman publicó al editor Emacs 19.31, en el cual el tipo de sistema fue renombrado de Linux a Lignux. Esta "escritura" fue pretendida para referirse expresamente a la combinación de GNU y Linux, pero esto pronto fue abandonado en favor de "GNU/Linux".

El producto terminado es más a menudo denominado simplemente como "Linux", como el más simple, el nombre original. Stallman anunció su demanda por un cambio de nombre sólo después de que el sistema ya se había hecho popular.

1.2.5. Cronología

- 1983: Richard Stallman crea el proyecto de GNU con el objetivo de crear un sistema operativo libre.

- 1989: Richard Stallman escribe la primera versión de la licencia GNU GPL.

- 1991: El núcleo Linux es anunciado públicamente,el 25 de agosto por el entonces estudiante finlandés de 21 años Linus Benedict Torvalds. El 17 de septiembre la primera versión pública aparece sobre un servidor de ftp. Algunos desarrolladores están interesados en el proyecto y contribuyen con mejoras y extensiones.

- 1992: El núcleo Linux es licenciado de nuevo bajo la GNU GPL. Las primeras distribuciones Linux son creadas.

- 1993: Más de 100 desarrolladores trabajan sobre el núcleo Linux. Con su ayuda el núcleo es adaptado al ambiente de GNU, que crea un espectro enorme de tipos de aplicaciones para el nuevo sistema operativo creado de la unión del software del proyecto GNU, variados programas de Software libre y el núcleo Linux. En este año, también el proyecto Wine comienza su desarrollo y la distribución más antigua actualmente activa, Slackware, es liberada por primera vez. Más tarde en el mismo año, el Proyecto Debian es establecido. Hoy esta es la comunidad más grande de una distribución.

- 1994: Torvalds considera que todos los componentes del núcleo Linux están totalmente maduros y presenta la versión 1.0 de Linux. Esta versión está, por primera vez, disponible en la red Internet. El proyecto XFree86 contribuye con una interfaz gráfica de usuario (GUI). En este año, las empresas Red Hat y SUSE también publican la versión 1.0.

- 1995: Aparece la siguiente rama estable de Linuz, la serie 1.2. Más tarde, Linux es transportado a las plataformas informáticas DEC y SUN SPARC. Durante los años siguientes es transportado a un número cada vez mayor de plataformas.

- 1996: La versión 2.0 del núcleo Linux es liberada. Éste ahora puede servir varios procesadores al mismo tiempo, y así se hace una alternativa seria para muchas empresas.

- 1997: Varios programas propietarios son liberados para Linux en el mercado, como la base de datos Adabas D, el navegador Netscape y las suites de oficina Applixware y StarOffice.

- 1998: Empresas importantes de informática como IBM, Compaq y Oracle anuncian soporte para Linux. Además, un grupo de programadores comienza a desarrollar la interfaz gráfica de usuario KDE, primera de su clase para Linux, con el objetivo de proveer facilidad de uso al usuario.

- 1999: Aparece la serie 2.2 del núcleo Linux, con el código de red y el soporte a SMP mejorados. Al mismo tiempo, un grupo de desarrolladores comienza el trabajo sobre el entorno gráfico GNOME, que competirá con KDE por la facilidad de uso y la eficiencia para el usuario. Durante ese año IBM anuncia un extenso proyecto para el soporte de Linux.

- 2000: La Suite de oficina StarOffice es ofrecida según los términos de la GNU GPL, abriendo así el camino para una Suite de oficina avanzada, y libre en Linux.

- 2001: En enero, se libera la serie 2.4 del núcleo Linux. El núcleo Linux ahora soporta hasta 64 Gb de RAM, sistemas de 64 bits, dispositivos USB y un si stema de archivos journaling.

- 2002: La comunidad OpenOffice.org libera la versión 1.0 de su Suite de oficina homónima. El navegador web libre Mozilla es también liberado. En septiembre, aparece el Slapper-worm el cual es el primer gusano informático Linux.

- 2003: La serie 2.6 del núcleo Linux es liberada, después de lo cual Linus Torvalds va a trabajar para el OSDL. Linux se usa más extensamente sobre sistemas integrados (embedded system).

- 2004: El equipo de XFree86 se desintegra y se forma la fundación X.Org, que provoca un desarrollo considerablemente más rápido del servidor X para Linux.

- 2005: El proyecto openSUSE es comenzado como una distribución libre de la comunidad de Novell. Además el proyecto OpenOffice.org proyecta la versión de lanzamiento 2.0 que soporta al estándar OASIS OpenDocument en octubre.

- 2006: El Xgl de Novell y el AIGLX de Red Hat permiten el uso de efectos acelerados por hardware sobre el escritorio Linux. Oracle publica su propia distribución de Red Hat. Novell y Microsoft anuncian una cooperación para la mejor interoperabilidad.

- 2007: Dell llega a ser el primer fabricante principal de computadoras en vender una computadora personal de escritorio con Ubuntu preinstalado.

1.3. El Proyecto Debian.

El proyecto Debian es una asociación de personas que han creado un sistema operativo gratis y de código abierto (*free*). Este sistema operativo se denomina Debian GNU/Linux o simplemente Debian.

Actualmente Debian ocupa el kernel Linux desarrollado por Linus Torvalds apoyado por miles de programadores de todo el mundo. También están implementados otros kernels como Hurd, desarrollado por GNU, NetBSD y FreeBSD.

La mayoría de las herramientas del sistema operativo Debian provienen del proyecto GNU y por ende son *free*. Cabe destacar que actualmente Debian tiene un total de más de 18733 paquetes (por paquetes entendemos *software* precompilado, para la versión estable, en un formato que permite su fácil instalación en nuestra máquina). Entre estos paquetes encontramos desde las herramientas básicas para procesar texto, hojas de cálculo, edición de imágenes, audio, video, hasta aplicaciones de gran utilidad científica. Es importante recordar que todo este *software* es *free* y por lo tanto está al alcance de todos sin la necesidad de comprar licencias ni pagar por actualizaciones. También existe la posibilidad de modificar el *software* ya que tenemos acceso al código fuente de los programas. Debian siempre mantiene activas al menos tres versiones que tienen las siguientes clasificaciones:

- *stable* (estable): Es la última versión oficial de Debian que ha sido probada para asegurar su estabilidad. Actualmente corresponde a la versión 4.0r0 cuyo nombre código es *lenny*.

- *testing* (en prueba): Esta es la versión que se está probando para asegurar su estabilidad y para luego pasar a ser versión estable. Nombre código *sqeeze*.

- *unstable* (inestable): Aquí es donde los programadores verdaderamente desarrollan Debian y por esto no es muy estable y no se recomienda para el uso diario. Esta versión se denomina siempre *sid*.

Para información sobre Debian y cómo bajarlo visite la página oficial `http://www.debian.org`.

1.4. Ingresando al sistema.

En esta sección comentaremos las operaciones de comienzo y fin de una sesión en UNIX así como la modificación de la contraseña asignada (que a menudo no es la deseada por el usuario, y que por lo tanto la puede olvidar con facilidad).

1.4.1. Terminales.

Para iniciar una sesión es necesario poder acceder a un terminal. Pueden destacarse dos tipos de terminales:

- Terminal de texto: consta de una pantalla y de un teclado. Como indica su nombre, en la pantalla sólo es posible imprimir caracteres de texto.
- Terminal gráfico: Consta de pantalla gráfica, teclado y *mouse*. Dicha pantalla suele ser de alta resolución. En este modo se pueden emplear ventanas que emulan el comportamiento de un terminal de texto (`xterm` o `gnome-terminal`).

1.4.2. Login.

El primer paso es encontrar un terminal libre donde aparezca el *login prompt* del sistema:

```
Debian GNU/Linux 6.0 hostname tty2

hostname login:
```

También pueden ocurrir un par de cosas:

- La pantalla no muestra nada.
  - Comprobar que la pantalla esté encendida.
  - Pulsar alguna tecla o mover el *mouse* para desactivar el protector de pantalla.
- Otra persona ha dejado una sesión abierta. En este caso existe la posibilidad de intentar en otra máquina o bien finalizar la sesión de dicha persona (si ésta no se halla en las proximidades).

  Una vez que se haya superado el paso anterior de encontrar el *login prompt* se procede con la introducción del *Username* al *prompt* de *login* y después la contraseña (*password*) correspondiente.

1.4.3. Passwords.

El *password* puede ser cualquier secuencia de caracteres que sea fácil de recordar por uno mismo. Si se olvida, deberá pasar un mal rato diciéndoselo al Administrador de Sistema. Para evitar que otra persona obtenga su *password* y tenga libre acceso a sus archivos, se seguirse considerar la siguiente pauta a la hora de elegir su *password*:

- Incluya mayúsculas y minúsculas, las cuales no son equivalentes, en ella.
- Use caracteres numéricos y no alfabéticos. Debe tenerse la precaución de usar caracteres que se puedan encontrar en todos los terminales desde los que se pretenda acceder.
- Evite usar palabras de diccionario.

Debe cambiar su *password* si cree que es conocido por otras personas, o descubre que algún intruso[7] está usando su cuenta. En general, es recomendable cambiar el *password* con regularidad.

La instrucción para cambiar el *password* en UNIX es `passwd`. A menudo, cuando existen varias máquinas que comparten recursos (discos duros, impresoras, correo electrónico, ...), para facilitar la administración de dicho se unifica los usuarios en una sola base de datos común. Dicho sistema se conoce como NIS (*Network Information Service*)[8]. Si el sistema usa este servicio, la modificación de la contraseña en una máquina requiere el comando `yppasswd` y supone la modificación en todas las máquinas que constituyan el dominio NIS.

1.4.4. Cerrando la sesión.

Es importante que nunca se deje abierta una sesión, pues algún "intruso" podría tener libre acceso a archivos de propiedad del usuario y manipularlos de forma indeseable. Para evitar todo esto basta teclear `logout` o `exit` y se habrá terminado la sesión en dicha máquina[9].

1.5. Archivos y directorios.

Los datos en los dispositivos de almacenamientos están organizados en archivos[10]. Loa archivos están organizados en directorios. EL conjunto completo de directorios y archivos el sistema de archivos (*filesystem*). En UNIX la *filesystem* es única y puede incluir uno o más dispositivos físicos. Aunque las diferentes distribuciones ubiquen sus programas en diferentes partes, la estructura básica de directorios en una máquina Linux es más o menos la misma:

[8] Antiguamente se conocía como YP (*Yellow Pages*), pero debido a un problema de marca registrada de *United Kingdom of British Telecomunications* se adoptaron las siglas NIS.

[9] En caso que se estuviera trabajando bajo X-Windows debes cerrar la sesión con `Log Out Usuario de Gnome`.

[10] Un conjunto ordenados de bytes.

```
/-|--> bin
  |--> boot
  |--> cdrom
  |--> dev
  |--> emul
  |--> etc
  |--> home
  |--> initrd
  |--> lib
  |--> lib32
  |--> lib64
  |--> media
  |--> mnt
  |--> opt
  |--> proc
  |--> root
  |--> sbin
  |--> selinux
  |--> sys
  |--> tmp
  |--> usr--|--> bin
            |--> games
            |--> include
            |--> lib
            |--> lib32
            |--> lib64
            |--> local -|--> bin
                        |--> include
                        |--> lib
                        |...
            |--> sbin
            |--> share
            |--> src --> linux
            |--> X11R6
            |...
  |--> var--|--> lock
            |--> log
            |--> mail
            |--> www
            |...
  |...
```

El árbol que observamos muestra el típico árbol de directorios en Linux. Pueden haber pequeñas variaciones en algunos de los nombres de estos directorios dependiendo de la distribución o versión de Linux que se esté usando. Entre los directorios más destacados tenemos:

- `/home` - Espacio reservado para las cuentas de los usuarios.
- `/bin`, `/usr/bin` - Binarios (ejecutables) básicos de UNIX.
- `/etc`, aquí se encuentran los archivos de configuración de todo los diferentes *softwares* de la máquina.
- `/proc`, es un sistema de archivos virtuales. Contiene archivos que residen en memoria y no en el disco duro. Hace referencia a los programas que están corriendo en este momento en el sistema.
- `/dev` (*device*) (dispositivo). Aquí se guardan los archivos asociados a los dispositivos. Se usan para acceder los dispositivos físicos del sistema y recursos tales como discos duros, *modems*, memoria, *mouse*, etc. Algunos dispositivos:
  - `hd`: `hda1` será el disco duro IDE, primario (`a`), y la primera partición (`1`).
  - `fd`: los archivos que empiecen con las letras `fd` se referirán a los controladores de las disketteras: `fd0` sería la primera diskettera, `fd1` sería la segunda y así sucesivamente.
  - `ttyS`: se usan para acceder a los puertos seriales como por ejemplo `ttyS0`, que es el puerto conocido como `com1`.
  - `sd`: son los dispositivos SCSI y/o SATA. Su uso es muy similar al del `hd`. También se usa para denominar a los dispositivos de almacenamiento conectados vía USB (*pendrives*).
  - `lp`: son los puertos paralelos. `lp0` es el puerto conocido como `LPT1`.
  - `null`: éste es usado como un agujero negro, ya que todo lo que se dirige allí desaparece.
  - `tty`: hacen referencia a cada una de las consolas virtuales. Como es de suponer, `tty1` será la primera consola virtual, `tty2` la segunda, etc.
- `/usr/local` - Zona con las aplicaciones no comunes a todos los sistemas UNIX, pero no por ello menos utilizadas.
- `/usr/share/doc` aquí se puede encontrar información relacionada con aplicaciones (en forma de páginas de manual, texto, html o bien archivos dvi, Postscript o pdf). También encontramos archivos de ejemplo, tutoriales, *HOWTO*, etc.

1.6. Los archivos y directorios.

En un sistema computacional la información se encuentra en archivos que pueden contener (documentos, datos, textos ASCII, fuentes en diferentes lenguajes (Python, Fortran, C++), ejecutables, imagenes, sonidos, figuras, ...). Para organizar toda esta información se dispone de una entidad denominada directorio, que permite el almacenamiento en su interior tanto de archivos como de otros directorios[11].

Los nombres de los archivos y directorios UNIX siguen las mismas convenciones y reglas.

- Los nombres diferencian mayúsculas y minúsculas.
- Los nombres pueden tener hasta 256 caracteres.
- Casi cualquier combinación de letras y símbolos es aceptable (el caracter `/` no se permite). Sin embargo, hay algunes caracteres **no** recomendables, ya que incluirlos en el nombre podría hacer el archino no borrable o inaccesible. Los caracteres no recomendados `| ; : , ! @ # $ ( ) < > / \ " ’ ‘ ~ { } [ ] = + & ^ <space> <tab>`.

Se dice que la estructura de directorios en UNIX es jerárquica o arborescente, debido a que todos los directorios nacen en un mismo punto (denominado directorio raíz). De hecho, la zona donde uno trabaja es un nodo de esa estructura de directorios, pudiendo uno a su vez generar una estructura por debajo de ese punto.

Un archivo se encuentra situado siempre en un directorio y su acceso se realiza empleando el camino que conduce a él en el **Árbol de Directorios del Sistema**. Este camino es conocido como el *path*. El acceso a un archivo se puede realizar empleando:

- Path Absoluto, aquél que empieza con `/`

  Por ejemplo: `/etc/printcap`

- Path Relativo, aquél que NO empieza con `/`

  Por ejemplo: `../examples/rc.dir.01`

Algunos caracteres especiales para el acceso a archivos son:	
`.`	Directorio actual
`..`	Directorio superior en el árbol
`~`	Directorio `$HOME`
`~user`	Directorio `$HOME` del usuario user

[11] Normalmente se acude a la imagen de una carpeta que puede contener informes, documentos o bien otras carpetas, y así sucesivamente.

1.7. Los comandos.

Para ejecutar un comando, basta con teclear su nombre en el `prompt` del sistema (se debe tener permiso para ejecutarlo). Las opciones o modificadores empiezan normalmente con el caracter `-` (p. ej. `comando -l`). Para especificar más de una opción, se pueden agrupar en una sola cadena de caracteres (`comando -l -h` es equivalente a `comando -lh`). Algunos comandos aceptan también opciones dadas por palabras completas, en cuyo caso usualmente comienzan con `--` (`comando --help`). Después de las opciones van el o los argumentos de ser necesarios.

1.7.1. Órdenes relacionadas con directorios.

`ls` (LiSt)
Este comando permite mostrar el contenido de un determinado directorio. Si no se le suministra argumento, muestra el contenido, archivos y directorios, del directorio actual. Si se añade el nombre de un directorio muestra el contenido del directorio suministrado como argumento. Existen varias opciones que modifican su funcionamiento entre las que destacan:

- `-l` (Long listing) proporciona un listado extenso, que consta de los permisos[12] de cada archivo, el usuario, el tamaño del archivo, . . . , etc. Adicionalmente la opción `-h` imprime los tamaños en un formato fácil de leer (Human readable).
- `-a` (list All) lista también los archivos ocultos.
- `-R` (Recursive) lista recursivamente el contenido de todos los directorios que encuentre.
- `-t` ordena los archivos por tiempo de modificación.
- `-S` ordena los archivos por tamaño.
- `-r` invierte el sentido de un ordenamiento.
- `-p` agrega un caracter al final de cada nombre de archivo, indicando el tipo de archivo (por ejemplo, los directorios son identificados con un `/` al final).

Los archivos cuyo nombre comiencen por `.` se denominan **ocultos**, así por ejemplo en el directorio de partida de un usuario.

```
bash$ ls -a  user
.                .alias       .fvwmrc  .login    .xinitrc
..               .cshrc       .joverc  .profile
.Xdefaults       .enviroment  .kshrc   .tcshrc
```

[12] Se comentará posteriormente este concepto.

Los caracteres comodín (*wildcard*) pueden ser empleados para acceder a un conjunto de archivos con características comunes. El signo `*` puede sustituir cualquier conjunto de caracteres[13] y el signo `?` a cualquier caracter individual. Por ejemplo:[14]

```
bash$ ls
f2c.1          flexdoc.1    rcmd.1    rptp.1   zforce.1
face.update.1  ftptool.1    rlab.1    rxvt.1   zip.1
faces.1        funzip.1     robot.1   zcat.1   zipinfo.1
flea.1         fvwm.1       rplay.1   zcmp.1   zmore.1
flex.1         rasttoppm.1  rplayd.1  zdiff.1  znew.1
bash$ ls rp*
rplay.1        rplayd.1     rptp.1
bash$ ls *e??
face.update.1  zforce.1     zmore.1
```

1.7.2. Los comandos para directorios.

`pwd` (Print Working Directory)
Este comando proporciona el nombre del directorio actual.

`cd` (Change Directory)
Permite moverse a través de la estructura de directorios. Si no se le proporciona argumento se provoca un salto al directorio `$HOME`. El argumento puede ser un *path* absoluto o relativo de un directorio. `cd -` vuelve al último directorio visitado.

`mkdir` (MaKe DIRectory)
Crea un directorio con el *path* (absoluto o relativo) proporcionado.

`rmdir` (ReMove DIRectory)
Elimina un directorio con el *path* (absoluto o relativo) suministrado. Dicho directorio debe de estar vacío.

1.7.3. Viendo archivos en pantalla.

Este conjunto de órdenes permite visualizar el contenido de un archivo sin modificar su contenido.

`cat` (ConcATenate)
Concatena dos o más archivos y los muestra en pantalla. Si el comando es llamado con un sólo archivo de argumento muestra (vuelca) el contenido del mismo en pantalla.

[13] Incluido el punto '.', UNIX no es DOS.
[14] `bash$` es el *prompt* en todos los ejemplos.

`more`
Muestra el contenido de un archivodividiendolo en páginas.

`less`
Es una versión mejorada del anterior. Permite moverse en ambas direcciones. Otra ventaja es que no lee el archivo entero antes de arrancar.

1.7.4. Copiando, moviendo y borrando archivos.

`cp` (CoPy)
Copia un archivo(s) con otro nombre y/o a otro directorio, por ejemplo, el comando para copiar el `archivo1.txt` con el nombre `archivo2.txt` es:

```
cp archivo1.txt archivo2.txt
```

Veamos algunas opciones:

- `-a` copia en forma recursiva, no sigue los *link* simbólicos y preserva los atributos de lo copiado.
- `-i` (interactive), impide que la copia provoque una pérdida del archivo destino si éste existe[15].
- `-R` (recursive), copia un directorio y toda la estructura que cuelga de él.

`mv` (MoVe)
Mueve un archivo(s) a otro nombre y/o a otro directorio, por ejemplo, el comando para mover el `archivo1.txt` al nombre `archivo2.txt` es:

```
mv archivo1.txt archivo2.txt
```

Este comando dispone de opciones análogas al anterior.

`rm` (ReMove)
Borra un archivo(s). En caso de que el argumento sea un directorio y se haya suministrado la opción `-r`, es posible borrar el directorio y todo su contenido. La opción `-i` pregunta antes de borrar.

[15] Muchos sistemas tienen esta opción habilitada a través de un alias, para evitar equivocaciones.

1.7.5. Espacio de disco.

El recurso de almacenamiento en el disco es siempre limitado. A continuacion, se comentan un par de comandos relacionados con la ocupación de este recurso:

`du` (Disk Usage)
Permite ver el espacio de disco ocupado (en bloques de disco[16]) por el archivo o directorio suministrado como argumento. La opción `-s` impide que cuando se aplique recursividad en un directorio se muestren los subtotales. La opción `-h` imprime los tamaños en un formato fácil de leer (Human readable).

`df` (Disk Free)
Muestra los sistemas de archivos que están montados en el sistema, con las cantidades totales, usadas y disponibles para cada uno. `df -h` muestra los tamaños en formato fácil de leer.

1.7.6. Links.

`ln` (LiNk)
Un enlace (*link*), permite el uso de un archivo en otro disinto al original sin necesidad de copiarlo, con el consiguiente ahorro de espacios. Un enlace puede ser:

- *hard link*: se puede realizar sólo entre archivos del mismo sistema de archivos. El archivo enlazado apunta a la zona de disco donde se ubica el archivo original. Por tanto, si se elimina el archivo original, el enlace sigue teniendo acceso a dicha información. Es el enlace por omisión.

- *symbolic link*: permite enlazar archivos/directorios[17] de diferentes sistemas de archivos. El archivo enlazado apunta al nombre del original. Así si se elimina el archivo original el enlace apunta hacia un nombre sin información asociada. Para realizar este tipo de enlace debe emplearse la opción `-s`.

Un enlace permite el uso de un archivo en otro directorio distinto del original sin necesidad de copiarlo, con el consiguiente ahorro de espacio. Veamos un ejemplo. Creemos un enlace clásico en Linux, al directorio existente `linux-3.1.4` nombrémoslo sencillamente linux.

```
mitarro:/usr/src# ln -s linux-3.1.4 linux
```

[16] 1 bloque normalmente es 1 Kbyte.
[17] Debe hacerse notar que los directorios sólo pueden ser enlazados simbólicamente.

1.8. Protección de archivos y permisos.

Dado que el sistema de archivos UNIX es compartido por un conjunto de usuarios, surge el problema de la necesidad de privacidad. Sin embargo, dado que existen conjuntos de personas que trabajan en común, es necesaria la posibilidad de que un conjunto de usuarios puedan tener acceso a una serie de archivos (que puede estar limitado para el resto de los usuarios). Cada archivo y directorio del sistema dispone de un propietario, un grupo al que pertenece y unos **permisos**. Existen tres tipos fundamentales de permisos:

- **lectura** (**r**-*Read*): en el caso de un archivo, significa poder examinar el contenido del mismo; en el caso de un directorio significa poder listar el contenido de dicho directorio.
- **escritura** (**w**-*Write*): en el caso de un archivo significa poder modificar su contenido; en el caso de un directorio permite crear nuevos archivos o directorios en su interior.
- **ejecución** (**x**-*eXecute*): en el caso de un archivo significa que ese archivo se pueda ejecutar (archivo de procedimientos o binario); en el caso de un directorio permite acceder a los archivos dentro de él.

Se distinguen tres conjuntos de usuarioss sobre las que se deben especificar permisos:

- **user**: el usuario propietario del archivo.
- **group**: el grupo propietario del archivo (al que pertenece el usuario). Cada usuario puede pertenecer a uno o varios grupos y el archivo generado pertenece a uno de los mismos.
- **other**: el resto de los usuarios (excepto el usuario y los usuarios que pertenezcan al grupo)

También se puede emplear *all* que es la unión de todos los anteriores. Para visualizar las protecciones de un archivo o directorio se emplea la orden `ls -l`, cuya salida es de la forma:

```
-rw-r--r-- ...otra información... nombre
```

Los 10 primeros caracteres muestran las protecciones de dicho archivo:

- El primer caracter indica el tipo de archivo de que se trata:
  - `-` archivo
  - `d` directorio
  - `l` enlace (*link*)
  - `c` dispositivo de caracteres (p.e. puerta serial)
  - `b` dispositivo de bloques (p.e. disco duro)
  - `s` socket (conexión de red)

- Los caracteres 2, 3, 4 son los permisos de usuario
- Los caracteres 5, 6, 7 son los permisos del grupo
- Los caracteres 8, 9, 10 son los permisos del resto de usuarios

Así en el ejemplo anterior `-rw-r--r--` se trata de un archivo donde el usuario puede leer y escribir, mientras que el grupo y el resto de usuarios sólo pueden leer. Estos suelen ser los permisos por omisión para un archivo creado por un usuario. Para un directorio los permisos por omisión suelen ser: `drwxr-xr-x`, donde se permite al usuario "entrar" en el directorio y ejecutar órdenes desde él.

`chmod` (CHange MODe)
Esta orden permite modificar los permisos de un archivo.

```
yo@mitarro:~$chmod permisos archivos
```

con opción `-R` es recursiva.

Existen dos modos de especificar los permisos, el modo absoluto o numérico y el modo simbólico o literal.

Modo absoluto o numérico. Se realiza empleando un número que resulta de la `OR` binario de los siguientes modos:

400	lectura por el propietario.
200	escritura por el propietario.
100	ejecución (búsqueda) por el propietario.
040	lectura por el grupo.
020	escritura por el grupo.
010	ejecución (búsqueda) por el grupo.
004	lectura por el resto.
002	escritura por el resto.
001	ejecución (búsqueda) por el resto.
4000	Set User ID, cuando se ejecuta el proceso corre con los permisos del dueño del archivo.

Por ejemplo:

```
yo@mitarro:~$chmod 640 *.txt
```

Permite la lectura y escritura por el usuario, lectura para el grupo y ningún permiso para el resto, de un conjunto de archivos que acaban en `.txt`

Modo simbólico o literal. Se realiza empleando una cadena (o cadenas separadas por comas) para especificar los permisos. Esta cadena se compone de los siguientes tres elementos: `who operation permission`

- `who` : es una combinación de:
  - `u` : user
  - `g` : group
  - `o` : others
  - `a` : all (equivalente a `ugo`)

  Si se omite este campo se supone `a`, con la restricción de no ir en contra de la máscara de creación (`umask`).
- `operation`: es una de las siguientes operaciones:
  - `+` : añadir permiso.
  - `-` : eliminar permiso.
  - `=` : asignar permiso, el resto de permisos de la misma categoría se anulan.
- `permission`: es una combinación de los caracteres:
  - `r` : *read*.
  - `w` : *write*.
  - `x` : *execute*.
  - `s` : en ejecución fija el usuario o el grupo.

Por ejemplo:

```
yo@mitarro:~$chmod u+x tarea,sh
```

Permite ejecución por parte del usuario[18] del archivo `tarea.sh`.

```
yo@mitarro:~$chmod u=rx, go=r *.txt
```

Permite la lectura y ejecución del usuario, y sólo la lectura por parte del grupo y el resto de usuarios.

[18] Un error muy frecuente es la creación de un archivo de órdenes (*script file*) y olvidar permitir la ejecución del mismo.

`umask`

Esta es una orden intrínseca del Shell que permite asignar los permisos que se desea tengan los archivos y directorios por omisión. El argumento que acompaña a la orden es un número octal que aplicará una XOR sobre los permisos por omisión (`rw-rw-rw-`) para archivos y (`rwxrwxrwx`) para directorios. El valor por omisión de la máscara es `022` que habilita al usuario para lectura-escritura, al grupo y al resto para lectura. Sin argumentos muestra el valor de la máscara.

`chgrp` (CHange GRouP)

Cambia el grupo propietario de una serie de archivos/directorios

```
yo@mitarro:~$chgrp grupo archivos
```

El usuario que efectúa esta orden debe pertenecer al grupo mencionado.

`chown` (CHange OWNer)

Cambia el propietario y el grupo de una serie de archivos/directorios

```
yo@mitarro:~$chown usuario:grupo archivos
```

La opción `-r` hace que la orden se efectúe recursivamente.

`id`

Muestra la identificación del usuario[19], así como el conjunto de grupos a los que el usuario pertenece.

```
yo@mitarro:~$id
uid=1001(yo)gid=1001(yo) groups=1001(yo),24(cdrom),44(video)
user@hostname:~$
```

`echo`

Despliega en pantalla su argumento como mensaje, sin argumento despliega una línea en blanco. La opción `-n` elimina el cambio de línea al final del mensaje.

```
yo@mitarro:~$ echo Hola curso
Hola curso
yo@mitarro:~$ echo

yo@mitarro:~$ echo -n Hola curso
Hola cursoyo@mitarro:~$
```

[19] A pesar de que el usuario se identifica por una cadena denominada *username*, también existe un número denominado UID que es un identificativo numérico de dicho usuario.

1.9. Shells.

El sistema operativo UNIX soporta varios intérpretes de comandos o *shells*, que ayudan a que la interacción con el sistema sea lo más cómoda y amigable posible. La elección de cuál es la *shell* más cómoda es algo personal; en este punto sólo indicaremos algunos de los más populares:

- **sh** : Bourne SHell, el *shell* básico, no pensado para uso interactivo.
- **csh** : C-SHell, *shell* con sintaxis como el lenguaje "C". El archivo de configuración es `.cshrc` (en el directorio `$HOME`).
- **ksh** : Korn-SHell, *shell* diseñada por David Korn en los Laboratorios AT&T Bell. Es un intento para una *shell* interactiva y para uso en *script*. Su lenguaje de comandos es un superconjunto de el lenguaje de *shell* sh.
- **tcsh** : alTernative C-Shell (Tenex-CSHell), con editor de línea de comando. El archivo de configuración es `.tcshrc`, o en caso de no existir, `.cshrc` (en el directorio `$HOME`).
- **bash** : Bourne-Again Shell, con lo mejor de sh, ksh y tcsh. El archivo de configuración es `.bash_profile` cuando se entra a la cuenta por primera vez, y después el archivo de configuración es `.bashrc` siempre en el directorio `$HOME`. La línea de comando puede ser editada usando comandos (secuencias de teclas) del editor `emacs`. En bash el modo de completado (*file completion*) es automático usando `TAB` cuando el *shell* está en modo interactivo. Es el *shell* por defecto de Linux.

1.9.1. Archivos de configuración.

En los archivo de configuración de los distintos *shells* se encuentran las definiciones de algunas variables de entorno o ambiente como el camino de búsqueda `PATH`, además de los aliases y otras configuraciones personales.

1.9.2. Cambiar de shell.

Si queremos cambiar de *shell* en un momento dado, sólo será necesario que escribamos, en al actual *shell*, el nombre del mismo y lo ejecutamos para estar usando dicho *shell*.

Si queremos usar de forma permanente otro *shell* del que tenemos asignado por omisión[20] podemos emplear la orden `chsh` que permite realizar esta acción.

[20] Por omisión se asigna bash.

1.9.3. Caracteres especiales.

Veamos unos caracteres con especial significado para el Shell:

- ` [21] permite que el *output* de un comando reemplace al nombre del comando. Por ejemplo:

```
yo@mitarro:~$ echo Estoy en‘pwd‘
Estoy en /home/yo
yo@mitarro:~$
```

- ' [22] preserva el significado literal de cada uno de los caracteres de la cadena que delimita.

```
yo@mitarro:~$ echo 'Estoy en‘pwd‘'
Estoy en ‘pwd‘
yo@mitarro:~$
```

- " [23] preserva el significado literal de todos los caracteres de la cadena que delimita, salvo $, ‘, \.

```
yo@mitarro:~$ echo "Estoy en‘pwd‘"
Estoy en /home/yo
yo@mitarro:~$
```

- ; permite la ejecución de más de una orden en una sola línea de comando.

```
yo@mitarro:~$ mkdir tmp;cd tmp
yo@mitarro:~/tmp$
yo@mitarro:~/tmp$ echo Hola;echo
Hola
yo@mitarro:~/tmp$
```

1.9.4. Variables de entorno.

Las variables de entorno permiten la configuración, por defecto, de muchos programas cuando ellos buscan datos o preferencias. Se encuentran definidas en los archivos de configuración anteriormente mencionados. Para referenciar a las variables se debe poner el símbolo $ delante, por ejemplo, para mostrar el camino al directorio por defecto del usuario:

[21] Acento grave o inclinado hacia atrás, *backquote*.
[22] Acento agudo o inclinado hacia adelante, *single quote*.
[23] *double quote*.

```
yo@mitarro:~$ echo $HOME
/home/yo
yo@mitarro:~$
```

Las variables de entorno más importantes son:

- `HOME` - El directorio por defecto del usuario.
- `PATH` - El camino de búsqueda, una lista de directorios separados con ':' para buscar programas.
- `EDITOR` - El editor por defecto del usuario.
- `DISPLAY` - Bajo el sistema de X windows, el nombre de máquina y pantalla que está usando. Si esta variable toma el valor `:0` el despliegue es local.
- `TERM` - El tipo de terminal. En la mayoría de los casos bajo el sistema X windows se trata de `xterm` y en la consola en Linux es `linux`. En otros sistemas puede ser `vt100`.
- `SHELL` - La *shell* por defecto.
- `MANPATH` - Camino para buscar páginas de manuales.
- `PAGER` - Programa de paginación de texto (`less` o `more`).
- `TMPDIR` - Directorio para archivos temporales.

1.9.5. Aliases.

Para facilitar la entrada de algunas órdenes o realizar operaciones complejas, los *shells* interactivos permiten el uso de aliases.

La orden `alias` permite ver qué aliases hay definidos y también definir nuevos. Es corriente definir el alias `rm ='rm -i'`, de esta forma la orden siempre pide confirmación para borrar un archivo.

Si alguna vez quieres usar `rm` sin alias, sólo hace falta poner delante el símbolo \, denominado *backslash* . Por ejemplo `\rm` elimina los alias aplicados a `rm`.

1.9.6. Ejecución de comandos.

- Si el comando introducido corresponde a un alias del *shell*, primero se reemplaza y luego se trata de ejecutar. Si el comando es propio del *shell* (*built-in*), se ejecuta directamente.
- En caso contrario:

- Si el comando contiene /, el *shell* lo considera un *path* absoluto e intenta resolverlo (entrar en cada directorio especificado para encontrar el comando y ejecutarlo).
- En caso contrario el *shell* busca en una tabla *hash table* que contiene los nombres de los comandos que se han encontrado en los directorios especificados en la variable de ambiente `PATH`, cuando ha arrancado el *shell*.

1.9.7. Comandos del shell.

`bash` se considera un *shell* interactivo, permitiendo la edición de la línea de comandos, y el acceso a la historia de órdenes (*readline*). En uso normal (historia y editor de línea de comandos) `bash` es compatible con otras *shells* como `tcsh` y `ksh`, ver apéndice. En `bash` el modo de completado (*file completion*) es automático (usando `TAB` sólo) si el *shell* está en modo interactivo.

`help`
Ayuda interna sobre los comandos del *shell*.

`set`
Muestra el valor de todas las variables.

`VARIABLE=VALUE`
Permite asignar el valor de una variable de entorno. Para que dicha variable sea "heredada" es necesario emplear: `export VARIABLE` o bien combinarlas: `export VARIABLE=VALUE`.

`alias`
En `bash`, `alias` sólo sirve para substitución simple de una cadena por otra. Por ejemplo: `alias ls='ls -F'`. Para crear alias con argumentos se usan funciones, ver la documentación.

`unalias name`
Elimina un alias asignado.

`for var in wordlist do comandos done`
A la variable `var`, que puede llamarse de cualquier modo, se le asignan sucesivamente los valores de la cadena `wordlist`, y se ejecuta el conjunto de comandos. El contenido de dicha variable puede ser empleado en los comandos: `$var`. Ejemplo:

```
$ for i in 1 2 tres 4; do echo $i; done
1
2
tres
4
```

`history`
Muestra las últimas órdenes introducidas en el *shell*. Algunos comandos relacionados con el *Command history* son:

- `!!`
  Repite la última orden.
- `!n`
  Repite la orden n-ésima.
- `!string`
  Repite la orden más reciente que empiece por la cadena `string`.
- `!?string`
  Repite la orden más reciente que contenga la cadena `string`.
- `^str1^str2` o `!!:s/str1/str2/`
  (*substitute*) Repite la última orden reemplanzando la primera ocurrencia de la cadena `str1` por la cadena `str2`.
- `!!:gs/str1/str2/`
  (*global substitute*) Repite la última orden reemplazando todas las ocurrencias de la cadena `str1` por la cadena `str2`.
- `!$`
  Es el último argumento de la orden anterior que se haya tecleado.

`source file`
Ejecuta las órdenes del fichero `file` en el *shell* actual.

`umask value`
Asigna la máscara para los permisos por omisión.

Los comandos `umask`, `source`, `history`, `unalias` y `hash` [24], funcionan igual en la *shell* `tcsh`.

[24] En `bash` y `sh` la *hash table* se va generando dinámicamente a medida que el usuario va empleando las órdenes. Así el arranque del *shell* es más rápido, y el uso de orden equivalente `hash -r` casi nunca hace falta.

1.9.8. Redirección.

Cuando un programa espera que se introduzca algo (se teclee), aquello que el usuario teclea se conoce como el *Standard Input*: `stdin`. Los caracteres que el programa retorna por pantalla es lo que se conoce como *Standard Output*: `stdout` (o *Standard Error*: `stderr`[25]).

El signo `<` permite que un programa reciba el `stdin` desde un archivo en vez de la interacción con el usuario. Por ejemplo: `mail root@mitarro < file`, invoca el comando `mail` con argumento (destinatario del mensaje) `root@mitarro`, siendo el contenido del mensaje el contenido del archivo `file` en vez del texto que usualmente teclea el usuario.

Más a menudo aparece la necesidad de almacenar en un archivo la salida de un comando. Para ello se emplea el signo `>`. Por ejemplo, `man bash > file`, invoca el comando `man` con argumento (información deseada) `bash` pero indicando que la información debe ser almacenada en el archivo `file` en vez de ser mostrada por pantalla.

En otras ocasiones uno desea que la salida de un programa sea la entrada de otro. Esto se logra empleando los denominados *pipes*, para ello se usa el signo `|`. Este signo permite que el `stdout` de un programa sea el `stdin` del siguiente. Por ejemplo:
`cat manual.txt | more` Invoca la orden de concatenación, con un solo archivo, vuelca el archivo sobre el `stdout` hacia el paginador `more`, de forma que podamos ver página a página el archivo.

A parte de los símbolos mencionados existen otros que permiten acciones tales como:

- `>>` Añadir el `stdout` al final del archivo indicado (*append*).[26]
- `&>` o `>&` (sólo `csh`, `tcsh` y `bash`) Redireccionar el `stdout` y `stderr`.
- `2>` Redirecciona sólo el `stderr`.
- `>>&` Igual que `>&` pero en modo *append*.

1.9.9. Archivos de *script*.

Un archivo de *script* es una sucesión de comandos de la *shell* que se ejecutan secuencialmente. Veamos un ejemplo simple:

```
#!/bin/bash
variable="/home/yo"
cp $1 /tmp/$2
rm $1
cd $variable
# Hecho por mi
```

[25] Si estos mensajes son de error.
[26] En `bash`, si el archivo no existe, es creado.

La primera línea declara la *shell* específica que se quiere usar. En la segunda línea hay una declaración de una variable interna. La tercera contiene los dos primeros argumentos con que fue llamado el *script*. Por ejemplo, si el anterior *script* está en un archivo llamado `ejemplo`, el comando `ejemplo file1 file2` asocia `$1` a `file1` y `$2` a `file2`. La línea 5 hace uso de la variable interna dentro de un comando. La última línea, que comienza con un `#` corresponde a un comentario. Notemos que la primera también es un comentario, pero la combinación `#!` en la primera línea fuerza a que se ejecute esa *shell*.

Esto sólo es una mínima pincelada de una herramienta muy poderosa y útil. Los comandos disponibles en la *shell* conforman un verdadero lenguaje de programación en sí, y los *scripts* pueden diseñarse para realizar tareas monótonas y complejas. Éste es un tema que le será útil profundizar.

1.9.10. Filtros.

Existe un conjunto de órdenes en UNIX que permiten el procesamiento de archivos de texto. Se denominan **filtros** (*Unix Filters*), porque normalmente se trabaja empleando redirección recibiendo datos por su `stdin`[27] y retornándolos modificados por su `stdout`[28].

`cat` (ConcATenate)
Concatena dos o más archivos y los muestra en pantalla. Si el comando es llamado con un sólo archivo de argumento muestra (vuelca) el contenido del mismo en el `stdout`.

```
yo@mitarro:~$ cat /etc/hostname
mitarro
yo@mitarro:~$
```

También lo podemos usar para crear un archivo

```
yo@mitarro:~$ cat  > archivo.txt
Este es mi archivo
con muchas lineas
^d
yo@mitarro:~$
```

El caracter final `^d` corresponde a fin de archivo y termina el ingreso.

Para facilitar la comprensión de los ejemplos siguientes crearemos cuatro archivos llamados: `milista.txt`, `tulista.txt`, `tercero.txt` y `cuarto.txt` usando el ejemplo anterior:

```
milista.txt       tulista.txt    tercero.txt   cuarto.txt
1 190             1 190          11  b         daniella Maria.
2 280             2 281          33  c         maria   Magdalena
3 370             3 370          222 a         NICOLLE    sarA
```

[27] Entrada estándar.
[28] Salida estándar.

`Tee` (Tee)
Lee desde `stdin` y escribe en el `stdout` y en una archivo.

```
yo@mitarro:~$ cat milista.txt |tee lista.txt
1 190
2 280
3 370
yo@mitarro:~$ cat lista.txt
1 190
2 280
3 370
yo@mitarro:~$
```

Con la opción `-a` agrega al archivo pasado como argumento (modo *append*).

`seq`
Genera una secuencia de números naturales consecutivos.

```
yo@mitarro:~$ seq 4 8
4
5
6
7
8
yo@mitarro:~$ seq -s, 4 8
4,5,6,7,8
```

Con la opción -s uno puede indicar el separador, en este caso el caracter `,`. Por defecto el separador es `\n`.

`cut`
Para un archivo compuesto por columnas de datos, permite escribir sobre la salida cierto intervalo de columnas. La opción `-b N-M` permite indicar el intervalo en bytes que se escribirán en la salida. La opción `-b N,M` permite indicar el los bytes que se escribirán en la salida.

```
yo@mitarro:~$ cut -b 3-4 milista.txt
19
28
37
yo@mitarro:~$ cut -b 3,5 milista.txt
10
20
30
```

`paste`

Mezcla líneas de distintos archivos. Escribe líneas en el `stdout` pegando secuencialmente las líneas correspondientes de cada uno de los archivo separadas por tab. Ejemplo, supongamos que tenemos nuestros archivos `milista.txt` y `tulista.txt` y damos el comando

```
yo@mitarro:~$ paste milista.txt tulista.txt
1 190   1 190
2 280   2 281
3 370   3 370
yo@mitarro:~$
```

`sed`

Es un editor de flujo. Veamos algunos ejemplos

```
yo@mitarro:~$ sed = milista.txt
1
1 190
2
2 280
3
3 370
yo@mitarro:~$
```

Numera las líneas.

```
yo@mitarro:~$ sed -n '3p' milista.txt
3 370
yo@mitarro:~$
```

Sólo muestra la línea 3. El modificador `-n` suprime la impresión de todas las líneas excepto aquellas especificadas por `p`. Separando por coma damos un rango en el número de líneas.

```
yo@mitarro:~$ sed '2q' milista.txt
1 190
2 280
yo@mitarro:~$
```

Muestra hasta la línea 2 y luego se sale de `sed`.

```
yo@mitarro:~$ sed 's/0/a/g' milista.txt
1 19a
2 28a
3 37a
yo@mitarro:~$
```

Reemplaza todos los 0 del archivo por la letra a. Éste es uno de los usos más comunes.

```
yo@mitarro:~$ sed '/2 2/s/0/a/g' milista.txt
1 190
2 28a
3 370
yo@mitarro:~$
```

Busca las líneas con la secuencia `2 2` y en ellas reemplaza todos los 0 por la letra a.

```
yo@mitarro:~$ sed 's/1/XX/2' milista.txt
1 XX90
2 280
3 370
yo@mitarro:~$
```

Reemplaza la segunda aparición de un 1 en una línea por los caracteres XX.

A continuación, mostramos otras posibilidades del comando `sed`

- Para remover una línea especifica (`X`) del archivo `file.txt`

  ```
  yo@mitarro:~$ sed 'Xd' file.txt
  ```

- Para remover un intervalo de líneas del archivo `file.txt`

  ```
  yo@mitarro:~$ sed 'X,Yd' file.txt
  ```

- Para mostrar sólo las líneas `X` e `Y` del archivo `file.txt`

  ```
  yo@mitarro:~$ sed -n 'Xp;Yp' file.txt
  ```

- Para mostrar el archivo `file.txt`, salvo las líneas que contengan `key`

  ```
  yo@mitarro:~$ sed '/key/d' file.txt
  ```

- Para mostrar el archivo `file.txt`, sólo las líneas que contengan `key`

  ```
  yo@mitarro:~$ sed -n '/key/p' file.txt
  ```

- Para mostrar el archivo `file.txt` salvo las líneas que comienzan con `#`

  ```
  yo@mitarro:~$ sed '/^#/d' file.txt
  ```

Expresiones regulares:

- `^` Matches al comienzo de la línea
- `$` Matches al final de la línea, se pone después del caracter a buscar.
- `.` Matches cualquier caracter.
- `[]` Matches con todos los caracteres dentro de los paréntesis

`diff`

Permite comparar el contenido de dos archivos o directorios

```
yo@mitarro:~$ diff milista.txt tulista.txt
2c2
< 2 280
---
> 2 281
yo@mitarro:~$
```

Hay una diferencia entre los archivos en la segunda fila.

`sort`

Permite ordenar alfabéticamente

```
yo@mitarro:~$ sort tercero.txt
11  b
222 a
33  c
yo@mitarro:~$ sort -r tercero.txt
33  c
222 a
11  b
yo@mitarro:~$ sort -n tercero.txt
11  b
33  c
222 a
yo@mitarro:~$ sort -k 2 tercero.txt
222 a
11  b
33  c
yo@mitarro:~$
```

La opción `-n` considera los valores numéricos y la opción `-r` invierte el orden. La opción `-k` permite especificar la columna a usar para hacer el `sort`.

`find`

Permite la búsqueda de un archivo en la estructura de directorios

```
yo@mitarro:~$ find . -name file.dat -print
```

Comenzando en el directorio actual(`.`) recorre la estructura de directorios buscando el archivo `file.dat`, cuando lo encuentre imprime el path al mismo, actualmente es innecesaria la opción `print`.

```
yo@mitarro:~$ find . -name '*~' -exec rm '{}' \;
```

Esta es otra aplicación de `find` que busca en la estructura de directorios un archivo que termine en `~` y lo borra.

El comando `xargs` ordena repetir orden para cada argumento que se lea desde *stdin*. Este lo podemos combinar con `find`.

```
yo@mitarro:~$ find . -name '*.mp3' -print | xargs mv /home/yo/musica \;
```

Logrando un comando que busca en la estructura de directorios todos los archivos que termines en `.mp3`, y los mueve al directorio `/home/yo/musica`.

`grep`

Permite la búsqueda de una cadena de caracteres en uno o varios archivos, imprimiendo el nombre del archivo y la línea en que se encuentra la cadena.

```
yo@mitarro:~$ grep 1 *list.txt
milista.txt:1 190
tulista.txt:1 190
tulista.txt:2 281
yo@mitarro:~$
```

Algunas opciones útiles

- `-c` Elimina la salida normal y sólo cuenta el número de apariciones de la cadena en cada archivo.
- `-i` Ignora para la comparación entre la cadena dada y el archivo, si la cadena está en mayúsculas o minúsculas.
- `-n` Incluye el número de líneas en que aparece la cadena en la salida normal.
- `-r` Hace la búsqueda recursiva.
- `-v` Invierte la búsqueda mostrando todas las líneas donde no aparece la cadena pedida.

`head`

Muestra las primeras diez líneas de un archivo.
`head -30 file` Muestra las 30 primeras líneas de *file*.

```
yo@mitarro:~$ head -1  milista.txt
1 190
yo@mitarro:~$
```

`tail`

Muestra las diez últimas líneas de un archivo.
`tail -30 file` Muestra las 30 últimas líneas de *file*.
`tail +30 file` Muestra desde la línea 30 en adelante de *file*.

```
yo@mitarro:~$ tail -1  milista.txt
3 370
yo@mitarro:~$
```

La opción `-f` permite que se actualice la salida cuando el archivo crece.

`awk`

Es un procesador de archivos de texto que permite la manipulación de las líneas de forma tal que tome decisiones en función del contenido de la misma. Ejemplo, supongamos que tenemos nuestro archivo `milista.txt` con sus dos columnas

```
yo@mitarro:~$ awk '{print }'  milista.txt
1 190
2 280
3 370
yo@mitarro:~$
```

Funciona como el comando `cat`

```
yo@mitarro:~$ awk '{print $2, $1 }'  milista.txt
190 1
280 2
370 3
yo@mitarro:~$
```

Imprime esas dos columnas en orden inverso.

```
yo@mitarro:~$ awk '{print ``a'', 8*$1, $2-1 }'  milista.txt
a 8 189
a 16 279
a 24 369
yo@mitarro:~$
```

Permite operar sobre las columnas.

```
yo@mitarro:~$ awk '{ if (NR>1 && NR < 3) print}' milista.txt
2 280
yo@mitarro:~$
```

Sólo imprime la línea 2.

```
yo@mitarro:~$ awk '{print $NF}' archivo.txt
```

Imprime en pantalla la última columna de cada fila del archivo `archivo.txt`.

```
yo@mitarro:~$ awk '{print NF}' archivo.txt
```

Imprime en pantalla el número respectivo a la última columna de cada fila, que equivalentemente es la cantidad de columnas por fila en el archivo `archivo.txt`.

Supongamos que tenemos el archivo `notas.txt` que contiene lo siguiente:

```
hugo 4.0 5.0 5.0
paco 3.0 6.0 4.0
luis 2.0 5.0 7.0
```

El comando

```
yo@mitarro:~$ awk '{print "Las notas del alumno",$0}' notas.txt
Las notas del alumno hugo 4.0 5.0 5.0
Las notas del alumno paco 3.0 6.0 4.0
Las notas del alumno luis 2.0 5.0 7.0
yo@mitarro:~$
```

El comando

```
yo@mitarro:~$ awk '{printf "%s %1.1f\n", $1,($2+$3+$4)/3}' notas.txt
hugo 4.7
paco 4.3
luis 4.7
yo@mitarro:~$
```

El comando

```
$awk '{ print >$N }'  archivo.txt
```

Crea archivos cuyos nombres correspondan a las palabras de la columna `N` en cada fila. Además, cada archivo contiene la fila correspondiente. Por ejemplo si aplicamos este filtro a nuestro archivo `notas.txt` por la primera columna, es decir,

```
$awk '{ print >$1 }'  archivo.txt
```

Crea tres archivos: `hugo`, `paco` y `luis`. El archivo `hugo` contiene

```
hugo 4.0 5.0 5.0
```

el archivo `paco`

```
paco 3.0 6.0 4.0
```

y el archivo `luis`

```
luis 2.0 5.0 7.0
```

`wc` (Word Count) Contabiliza el número de líneas, palabras y caracteres de un archivo.

```
yo@mitarro:~$ wc  milista.txt
      3       6      18 milista.txt
yo@mitarro:~$
```

El archivo tiene 3 líneas, 6 palabras, considerando cada número como una palabra *i.e.* 1 es la primera palabra y 190 la segunda, y finalmente 18 caracteres. ¿Cuáles son los 18 caracteres?

`uniq`
Reporta, omite o cuenta líneas repetidas, **adyacentes**.

```
yo@mitarro:~$ cat milista.txt tulista.txt |sort| uniq
1 190
2 280
2 281
3 370
yo@mitarro:~$ cat milista.txt tulista.txt |sort| uniq -d
1 190
3 370
yo@mitarro:~$ cat milista.txt tulista.txt |sort| uniq -c
2 1 190
1 2 280
1 2 281
2 3 370
yo@mitarro:~$
```

`tr` (TRanslate)
Traduce "colapsa" y/o borra caracteres del `stdin` escribiendolos en el `stdout`.

```
yo@mitarro:~$ tr [:upper:] [:lower:]<cuarto.txt|tee 4.txt
daniella maria.
maria
magdalena
nicolle
sara
yo@mitarro:~$ cat 4.txt|tr -s l|tr -s [:blank:]|tr -d .
daniela maria
maria magdalena
nicole sara
yo@mitarro:~$
```

La opción `-s` colapsa a sólo una las apariciones del caracter. La opción `-d` borra el caracter.

1.9.11. Utilitarios.

`users` `who` `w`

Para ver quién está conectado en la máquina.

`cal`

Muestra el calendario del mes actual. Con la opción `-y` y el año presenta el calendario del año completo.

```
yo@mitarro:~$ cal
   December 2011
Su Mo Tu We Th Fr Sa
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

yo@mitarro:~$
```

`date`

Muestra el día y la hora actual.

`bc`

Calculadora de precisión arbitraria.

```
yo@mitarro:~$ echo 3.76*17.8|bc}
66.92
```

```
yo@mitarro:~$ echo "scale=3;s(2.0)"|bc -l
0.909
yo@mitarro:~$
```

Permite calcular expresiones directamente desde la línea de comando mediante la calculadora de precisión arbitraria `bc`. En el segundo caso se fijan los decimales de salida a 3 y se evalua la función sen(2.0), la opción `-l` permite el uso de funciones especiales.

1.10. Ayuda y documentación.

Para obtener ayuda sobre comandos de UNIX, se puede emplear la ayuda *on-line*, en la forma de páginas de manual.

Así `man comando` proporciona la ayuda sobre el `comando` deseado. Por ejemplo, para leer el manual de los shells, puedes entrar: `man sh csh tcsh bash`, la orden formatea las páginas y te permite leer los manuales en el orden pedido.

En el caso de `bash` se puede usar el comando `help`, por ejemplo, `help alias`.

Además, para muchos comandos y programas se puede obtener información tipeando `info comando`.

Finalmente, algunos comandos tienen una opción de ayuda (`--help` o `-h`), para recordar rápidamente las opciones más comunes disponibles (`ls --help`).

1.11. Diferencias entre sistemas.

Cuando se transfieren archivos de texto entre WINDOWS y UNIX sin las precauciones adecuadas pueden aparecer los siguientes problemas:

- En UNIX no existe obligatoriedad respecto a que los archivos llevan extensión. Incluso pueden tener más de una extensión `algo.v01.tar.gz`, esto puede complicar a otros sistemas que usan sólo una extensión de tres caracteres.
- El cambio de línea en un archivo de texto WINDOWS se compone de *Carriage Return* y *Line Feed*. Sin embargo, en UNIX sólo existe el *Carriage Return*. Así un archivo de UNIX visto desde WINDOWS parece una única línea. El caso inverso es la aparición del caracter `^M` al final de cada línea. Además, el fin de archivo en WINDOWS es `^Z` y en UNIX es `^D`.

Usando el comando `tr` se puede transformar un archivo con cambios de líneas para `DOS` en uno para UNIX. Sabiendo que `^M` es ASCII 13 decimal, pero 15 en octal:

```
tr -d '\015' < datafile > TEMPFILE
mv -f TEMPFILE datafile
```

En Debian, instalando el paquete `dos2unix`, queda instalado el comando `dos2unix` que también lo hace.

1.12. Procesos.

En una máquina existen una multitud de procesos que pueden estar ejecutándose simultáneamente. La mayoría de ellos no corresponden a ninguna acción realizada por el usuario y no merecen que se les preste mayor atención. Estos procesos corresponden a programas ejecutados en el arranque del sistema y tienen que ver con el funcionamiento global del servidor. En general, los programas suelen tener uno de estos dos modos de ejecución:

- **foreground**: Son aquellos procesos que requieren de la interacción y/o atención del usuario mientras se están ejecutando, o bien en una de sus fases de ejecución (*i.e.* introducción de datos). Así por ejemplo, la consulta de una página de manual es un proceso que debe ejecutarse claramente en *foreground*.

- **background**: Son aquellos procesos que no requieren de la interacción con el usuario para su ejecución. Si bien el usuario desearía estar informado cuando este proceso termine. Un ejemplo de este caso sería la impresión de un archivo.

Sin embargo, esta división que a primera vista pueda parecer tan clara y concisa, a menudo en la práctica aparece la necesidad de conmutar de un modo al otro, detención de tareas indeseadas, etc. Así por ejemplo, puede darse el caso de que estemos leyendo una página de manual y de repente necesitemos ejecutar otra tarea. Un proceso viene caracterizado por:

- *process number*

- *job number*

Veamos algunas de las órdenes más frecuentes para la manipulación de procesos:

- `comando &` Ejecución de un comando en el *background*. [29]

- `Ctrl-Z` Detiene el proceso que estuviera ejecutándose en el *foreground* y lo coloca detenido en el *background*.

- `Ctrl-C` Termina un proceso que estaba ejecutándose en *foreground*.

- `Ctrl-\` Termina de forma definitiva un proceso que estaba ejecutándose en *foreground*.

- `ps x` Lista todos los procesos que pertenezcan al usuario, incluyendo los que no están asociados a un terminal.

- `jobs` Lista los procesos que se hayan ejecutado desde el *shell* actual, mostrando el *job number*.

[29] Por omisión un comando se ejecuta siempre en el *foreground*.

- `fg (job number)` Pasa a ejecución en *foreground* un proceso que se hallase en *background*.
- `bg (job number)` Pasa a ejecución en *background* un proceso que se hallase detenido con `Ctrl-Z`.
- `kill (process number)` Envía una señal[30] a un proceso UNIX. En particular, para enviar la señal de término a un programa, damos el comando `kill -KILL`, pero no hace falta al ser la señal por defecto.

Cuando se intenta abandonar una sesión con algún proceso aún detenido en el *background* del *shell*, se informa de ello con un mensaje del tipo: `There are stopped jobs` si no importa, el usuario puede intentar abandonar de nuevo el *shell* y éste matará los *jobs*, o puedes utilizar `fg` para traerlos al *foreground* y ahí terminar el mismo.

1.13. Editores.

Un editor es un programa que permite crear y/o modificar un archivo. Existen una multitud de editores diferentes, y al igual que ocurre con los *shells*, cada usuario tiene alguno de su predilección. Mencionaremos algunos de los más conocidos:

- **vi** - El editor standard de UNIX.
- **emacs (xemacs)** - Editor muy configurable escrito en lenguaje Lisp. Existen muchos modos para este editor (lector de mail, news, www,...) que lo convierten en un verdadero *shell* para multitud de usuarios. Las últimas versiones del mismo permiten la ejecución desde X-windows o terminal indistintamente con el mismo binario. Posee un tutorial en línea, comando `C-H t` dentro del editor. El archivo de configuración personalizada es: `$HOME/.emacs`.
- **jove** - Basado en Emacs, (Jonathan's Own Version of Emacs). Posee tutorial en una utilidad asociada: `teachjove`. El archivo de configuración personalizada es: `$HOME/.joverc`.
- **jed** - Editor configurable escrito en S-Lang. Permite la emulación de editores como emacs y Wordstar. Posee una ayuda en línea `C-H C-H`. El archivo de configuración personalizada es: `$HOME/.jedrc`.
- **gedit** - Un pequeño y liviano editor de texto para Gnome
- **xjed** - Versión de jed para el X-windows system. Presenta como ventaja que es capaz de funcionar en muchos modos: lenguaje C, Fortran, TeX, etc., reconociendo palabras clave y signos de puntuación, empleando un colorido distinto para ellos. El archivo de configuración personalizada es el mismo que el de jed.

[30] Para ver las señales disponibles entra la orden `kill -l` (l por *list*).

Dado que los editores del tipo de **gedit** disponen de menús auto explicativos, daremos a continuación unas ligeras nociones sólo de `vi` y `emacs`.

1.13.1. El editor vi.

El `vi` es un editor de texto muy poderoso pero un poco difícil de usar. Lo importante de este editor es que se puede encontrar en cualquier sistema UNIX y sólo hay unas pocas diferencias entre un sistema y otro. Explicaremos lo básico solamente. Comencemos con el comando para invocarlo:

```
localhost:/# vi
```

```
~
~
~
/tmp/vi.9Xdrxi: new file: line 1
```

La sintaxis para editar un archivo es:

```
localhost:/# vi nombre.de.archivo
```

```
~
~
~
nombre.de.archivo: new file: line 1
```

Insertar y borrar texto en vi.

Cuando se inicia el `vi`, editando un archivo, o no, se entra en un modo de órdenes, es decir, que no se puede empezar a escribir directamente. Si se quiere entrar en modo de inserción de texto se debe presionar la tecla `i`. Entrando en el modo de inserción, se puede empezar a escribir. Para salir del modo de inserción de texto y volver al modo de órdenes se apreta `ESC`.

```
Aqui ya estamos escribiendo porque apretamos
la tecla 'i' al estar en modo ordenes.
~
~
```

La tecla `a` en el modo de órdenes también entra en modo de inserción de texto, pero en vez de comenzar a escribir en la posición del cursor, empieza un espacio después.

La tecla `o` en el modo de órdenes inserta texto pero desde la línea que sigue a la línea donde se está ubicado.

Para borrar texto, hay que salir al modo órdenes, y presionar la tecla `x` que borrará el texto que se encuentre sobre el cursor. Si se quiere borrar las líneas enteras, entonces se debe presionar dos veces la tecla `d` sobre la línea que deseo eliminar. Si se presionan las teclas `dw` se borra la palabra sobre la que se está ubicado.

La letra `R` sobre una palabra se puede escribir encima de ella. Esto es una especie de modo de inserción de texto pero sólo se podrá modificar la palabra sobre la que se está situado. La tecla `~` cambia de mayúscula a minúscula la letra sobre la que se está situado.

Moverse dentro de vi.

Estando en modo órdenes podemos movernos por el archivo que se está editando usando las flechas hacia la izquierda, derecha, abajo o arriba. Con la tecla `0` nos movemos al comienzo de la línea y con la tecla `$` nos movemos al final de la misma.

Con las teclas `w` y `b` nos movemos al comienzo de la siguiente palabra o al de la palabra anterior respectivamente. Para moverme hacia la pantalla siguiente la combinación de teclas `CTRL F` y para volver a la pantalla anterior `CTRL B`. Para ir hasta el principio del archivo se presiona la tecla `G`.

Opciones de comandos.

Para entrar al menú de comandos se debe presionar la tecla `:` en el modo de órdenes. Aparecerán los dos puntos (`:`). Aquí se pueden ingresar ordenes para guardar, salir, cambiar de archivo entre otras cosas. Veamos algunos ejemplos:

- `:w` Guardar los cambios.
- `:w otherfile.txt` Guardar con el nuevo nombre `otherfile.txt`
- `:wq` Guardar los cambios y salir.
- `:q!` Salir del archivo sin guardar los cambios.
- `:e file1.txt` Si deseo editar otro archivo al que se le pondrá por nombre `file1.txt`.
- `:r file.txt` Si se quiere insertar un archivo ya existente, por ejemplo `file.txt`.
- `:r! comando` Si se quiere ejecutar algún comando del *shell* y que su salida aparezca en el archivo que se está editando.

1.13.2. Editores modo emacs.

El editor `GNU Emacs`, escrito por Richard Stallman de la *Free Software Foundation*, es uno de los que tienen mayor aceptación entre los usuarios de UNIX, estando disponible bajo licencia **GNU GPL**[31] para una gran cantidad de arquitecturas. También existe otra versión de emacs llamada `XEmacs` totalmente compatible con la anterior pero presentando mejoras significativas respecto al `GNU Emacs`. Dentro de los "inconvenientes" que presenta es que no viene por defecto incluido en la mayoría de los sistemas UNIX. Las actuales distribuciones de Linux y en particular Debian GNU/Linux contienen ambas versiones de emacs, tanto `GNU Emacs` como `XEmacs`, como también versiones de `jove`, `jed`, `xjed` y muchos otros editores. Para mayor información ver Apéndice.

1.14. El sistema X Windows.

El *X Windows system* es el sistema estándar de ventanas en las estaciones de trabajo. Lo usual actualmente es que el sistema de ventanas sea arrancado automáticamente cuando la máquina parte. En el sistema *X Windows* deben distinguirse dos conceptos:

- **server** : Es un programa que se encarga de escribir en el dispositivo de video y de capturar las entradas (por teclado, ratón, etc.). Asimismo se encarga de mantener los recursos y preferencias de las aplicaciones. Sólo puede existir un server para cada pantalla.
- **client** : Es cualquier aplicación que se ejecute en el sistema *X Windows*. No hay límite (en principio) en el número de clientes que pueden estarse ejecutando simultáneamente. Los clientes pueden ser locales o remotos.

Window Manager (WM) Es un cliente con "privilegios especiales": controla el comportamiento (forma, tamaño,...) del resto de clientes. Existen varios, destacando:

- **icewm** : *Ice Window Manager*, uno de los *window managers* gnome compatible.
- **sawfish** : *Window managers* gnome compatible, altamente configurable y muy integrado al gnome *desktop*.
- **Metacity** : *Window managers* gnome 2 compatible.

El *look and feel* (o GUI) de *X Windows* es extremadamente configurable, y puede parecer que dos máquinas son muy distintas, pero esto se debe al `WM` que se esté usando y no a que las aplicaciones sean distintas.

[31] La licencia de GNU, da el permiso de libre uso de los programas con sus fuentes, pero los autores mantienen el *Copyright* y no es permitido distribuir los binarios sin acceso a sus fuentes. Los programas derivados de dichos fuentes heredan la licencia GNU.

Para configurar tu sesión es necesario saber qué programas estás usando y ver las páginas de manual. Los archivos principales son:

- `.xinitrc` o `.xsession` archivo leído al arrancar *X Windows*. Aquí se pueden definir los programas que aparecen al inicio de tu sesión.
- `.fvwmrc` archivo de configuración del `fvwm`. Ver las páginas del manual de `fvwm`.
- `.olwmrc` archivo de configuración del `olwm`. Ver las páginas del manual de `olwm`.
- `.Xdefaults` Configuración general de las aplicaciones de X Windows. Aquí puedes definir los *resources* que encontrarás en los manuales de las aplicaciones de X.

1.15. Uso del ratón.

El ratón es un dispositivo esencial en el uso de programas X, sin embargo, la función que realiza en cada uno de ellos no está normalizada.

Comentaremos la pauta seguida por la mayoría de las aplicaciones, pero debe tenerse presente que es muy frecuente encontrar aplicaciones que no las respetan.[32]

- **Botón izquierdo** (LB): Seleccionar. Comienza el bloque de selección.
- **Botón central** (MB): Pegar. Copia la selección en la posición del cursor.
- **Botón derecho** (RB): Habitualmente ofrece un menú para partir aplicaciones.

Existen dos modos para determinar cuál es la **ventana activa**, aquélla que recibe las entradas de teclado:

- *Focus Follows Mouse*: La ventana que contenga al ratón es la que es activa. No usado por defecto actualmente.
- *Click To Focus*: La ventana seleccionada es la activa. El modo que esté activo depende de la configuración del *Window Manager*.

[32] Las aplicaciones que son conscientes de un uso anormal y están realizadas por programadores inteligentes, muestran en pantalla la función de cada botón cuando son posibles varias alternativas.

1.16. Internet.

En esta sección denominaremos `unix1` a la máquina local (desde donde ejecutamos la orden) y `unix2` a la máquina remota (con la que interaccionamos). Ambos son los `hostnames` de las respectivas máquinas. Existen algunos conceptos que previamente debemos comentar:

- `IP-number`: es un conjunto de 4 números separados por puntos (p.e. 200.89.74.6) que se asocia a cada máquina. No puede haber dos máquinas conectadas en la misma red con el mismo número.

- `hostname`: es el nombre que tiene asociada la máquina (p.e. `macul`). A este nombre se le suelen añadir una serie de sufijos separados por puntos que constituyen el denominado dominio (p.e. `macul.ciencias.uchile.cl`). Una máquina por tanto puede tener más de un nombre reconocido (se habla en este caso de alias). Se denomina resolución a la identificación entre un `hostname` y el `IP-number` correspondiente. La consulta se realiza inicialmente en el archivo `/etc/hosts`, donde normalmente se guardan las identificaciones de las máquinas más comúnmente empleadas. En caso de que no se logre se accede al servicio `DNS` (*Domain Name Service*), que permite la identificación (resolución) entre un `hostname` y un `IP-number`.

- `mail-address`: es el nombre que se emplea para enviar correo electrónico. Este nombre puede coincidir con el nombre de una máquina, pero se suele definir como un alias, con objeto de que la dirección no deba de cambiarse si la máquina se estropea o se cambia por otra.

1.16.1. Otras máquinas.

`ping`

Verifica si una máquina está conectada a la red y si el camino de Internet hasta la misma funciona correctamente.

`finger`

`finger user`, muestra información[33] sobre el usuario `user` en la máquina local.

`finger user@hostname`, muestra información sobre un usuario llamado `user` en una máquina `hostname`.

`finger @hostname`, muestra los usuarios conectados en la máquina `hostname`.

Este comando suele estar desabilitado en las máquinas actuales.

[33] La información proporcionada es el nombre completo del usuario, las últimas sesiones en dicha máquina, si ha leído o no su correo y el contenido de los archivos `.project` y `.plan` del usuario.

1.16.2. Acceso a la red.

Existen muchos programas para la conexión de la red, los más usados son:

- `telnet unix2`, hace un login en la máquina `unix2`, debe ingresarse el usuario y su respectiva `passwd`. Además, permite especificar el puerto en conexión en la máquina remota.

- `ssh nombre@unix2`, muy similar a `telnet` pero se puede especificar el usuario, si no se especifica se usa el nombre de la cuenta local. Además, el `passwd` pasa encriptado a través de la red. `ssh nombre@unix2 comando`, muy similar a `rsh`, el `passwd` pasa encriptado y ejecuta el comando en la máquina remota, mostrando el resultado en la máquina local.

- `scp file1 usuario2@unix2:path/file`, copia el archivo `file1`, del usuario1, que se encuentra en el directorio local en la máquina `unix1`, en la cuenta del `usuario2` en la máquina `unix2` en `$HOME/path/file`. Si no se especifica el nombre del usuario se usa el nombre de la cuenta local. Si se quiere copiar el archivo `file2` del `usuario3` en `unix2` en la cuenta actual de `unix1` el comando sería: `scp usuario3@unix2:file2 .`. Antes de realizar cualquiera de las copias el sistema preguntará por el `passwd` del usuario en cuestión en la máquina `unix2`. Nuevamente, el `passwd` pasa encriptado a través de la red.

- `talk usuario1@unix2`, intenta hacer una conexión para hablar con el `usuario1` en la máquina `unix2`. Existen varias versiones de `talk` en los diferentes sistemas operativos, de forma que no siempre es posible establecer una comunicación entre máquinas con sistemas operativos diferentes.

- `ftp unix2`, (file transfer protocol) aplicación para copiar archivos entre máquinas de una red. `ftp` exige un nombre de cuenta y password para la máquina remota. Algunas de las opciones más empleadas (una vez establecida la conexión) son:

  - `bin`: Establece el modo de comunicación binario. Es decir, transfiere una imagen exacta del archivo.
  - `asc`: Establece el modo de comunicación `ascii`. Realiza las conversiones necesarias entre las dos máquinas en comunicación. Es el modo por defecto.
  - `cd`: Cambia directorio en la máquina remota.
  - `lcd`: Cambia directorio en la máquina local.
  - `ls`: Lista el directorio remoto.
  - `!ls`: Lista el directorio local.
  - `prompt` : No pide confirmación para transferencia múltiple de archivos.

- `get rfile [lfile]`: transfiere el archivo `rfile` de la máquina remota a la máquina local denominándolo `lfile`. En caso de no suministrarse el segundo argumento supone igual nombre en ambas máquinas.
- `put lfile [rfile]` : transfiere el archivo `lfile` de la máquina local a la máquina remota denominándolo `rfile`. En caso de no suministrarse el segundo argumento supone igual nombre en ambas máquinas. También puede usarse `send`.
- `mget rfile` : igual que `get`, pero con más de un archivo (`rfile` puede contener caracteres comodines).
- `mput lfile` : igual que `put`, pero con más de un archivo (`lfile` puede contener caracteres comodines).

Existen versiones mejoradas de `ftp` como `ncftp`. También existen versiones gráficas de clientes `ftp` donde la elección de archivo, el sentido de la transferencia y el modo de ésta, se elige con el *mouse* (p.e. `wxftp`).

- `rlogin -l nombre unix2`, (*remote login*), hace un `login` a la máquina `unix2` como el usuario `nombre` por defecto, sin los argumentos `-l nombre` `rlogin` usa el nombre de la cuenta local. Normalmente `rlogin` pide el *password* de la cuenta remota, pero con el uso del archivo `.rhosts` o `/etc/hosts.equiv` esto no es siempre necesario.
- `rsh -l nombre unix2 orden`, (*remote shell*), ejecuta la orden en la máquina `unix2` como usuario `nombre`. Es necesario que pueda entrar en la máquina remota sin *password* para ejecutar una orden remota. Sin especificar orden actúa como `rlogin`.

1.16.3. Exportando el DISPLAY.

En caso de que tengas que correr una aplicación de X que no esté disponible en la máquina que estás usando, eso no representa ningún problema. Las órdenes necesarias son (por ejemplo, para arrancar un `gnome-terminal` remoto):

```
user@hostname1:~$ ssh -XC userB@hostname2
userB@hostname2's password:
userB@hostname2:~$ gnome-terminal &
```

Las opciones `XC` en el comando `ssh` corresponden a que exporte el `DISPLAY` y que comprima, respectivamente. La forma antigua

```
userA@hostname1:~$ xhost +hostname2
hostname2 being added to access control list
user@hostname1:~$ ssh userB@hostname2
userB@hostname2's password:
userB@hostname2:~$ export DISPLAY=hostname1:0
userB@hostname2:~$ gnome-terminal &
```

Si todo está previamente configurado, es posible que no haga falta dar el *password*.

1.16.4. El correo electrónico.

El correo electrónico (`e-mail`) es un servicio para el envío de mensajes entre usuarios, tanto de la misma máquina como de diferentes máquinas.

Direcciones de correo electrónico.

Para mandar un `e-mail` es necesario conocer la dirección del destinatario. Esta dirección consta de dos campos que se combinan intercalando entre ellos el @ (*at*): `user@domain`

- `user` : es la identificación del usuario (*i.e.* `login`) en la máquina remota.
- `domain` : es la máquina donde recibe correo el destinatario. A menudo, es frecuente que si una persona tiene acceso a un conjunto de máquinas, su dirección de correo no corresponda con una máquina sino que corresponda a un alias que se resolverá en un nombre específico de máquina en forma oculta para el que envía.

Si el usuario es local, no es necesario colocar el campo `domain` (ni tampoco el @).

Nomenclatura.

Veamos algunos conceptos relacionados con el correo electrónico:

- **Subject** : Es una parte de un mensaje que piden los programas al comienzo y sirve como título para el mensaje.
- **Cc** (Carbon Copy) : Permite el envío de copias del mensaje que está siendo editado a terceras personas.
- **Reply** : Cuando se envía un mensaje en respuesta a otro se suele añadir el comienzo del *subject:* `Re:`, con objeto de orientar al destinatario sobre el tema que se responde. Es frecuente que se incluya el mensaje al que se responde para facilitar al destinatario la comprensión de la respuesta.
- **Forward** : Permite reenviar un mensaje completo (con modificaciones o sin ellas) a una tercera persona. Notando que Forward envía también los archivos adjuntos, mientras que la opción Reply no lo hace.
- **Forwarding Mail** : Permite a un usuario que disponga de cuentas en varias máquinas no relacionadas, de concentrar su correo en una cuenta única[34]. Para ello basta con tener un archivo `$HOME/.forward` que contenga la dirección donde desea centralizar su correo.

[34]Este comando debe usarse con conocimiento pues en caso contrario podría provocar un *loop* indefinido y no recibir nunca correo.

- **Mail group** : Un grupo de correo es un conjunto de usuarios que reciben el correo dirigido a su grupo. Existen órdenes para responder a un determinado correo recibido por esa vía de forma que el resto del grupo sepa lo que ha respondido un miembro del mismo.

- **In-Box** : Es el archivo donde se almacena el correo que todavía no ha sido leído por el usuario. Suele estar localizado en `/var/spool/mail/user`.

- **Mailer-Daemon** : Cuando existe un problema en la transmisión de un mensaje se recibe un mensaje proveniente del *Mailer-Daemon* que indica el problema que se ha presentado.

Aplicación mail.

Es posiblemente la aplicación más simple. Para la lectura de mail teclear simplemente: `mail` y a continuación aparece un índice con los diferentes mensajes recibidos. Cada mensaje tiene una línea de identificación con número. Para leer un mensaje basta teclear su número y a continuación `return`. Para enviar un mensaje: `mail (address)` se pregunta por el `Subject:` y a continuación se introduce el mensaje. Para acabar se teclea sólo un punto en una línea o bien `Ctr-D`. Por último, se pregunta por `Cc:`. Es posible personalizar el funcionamiento mediante el archivo `$HOME/.mailrc`.

Para enviar un archivo de texto a través del correo se suele emplear la redirección de entrada:

```
yo@mitarro:~$ mail tu@otrotarro < carta.txt
yo@mitarro:~$
```

Si queremos enviar un archivo binario en forma de `attach` en el mail, el comando es

```
yo@mitarro:~$ mpack informe.pdf tu@otrotarro
yo@mitarro:~$
```

1.16.5. Ftp anonymous.

Existen servidores que permiten el acceso por `ftp` a usuarios que no disponen de cuenta en dichas máquinas. Para ello se emplea como `login` de entrada el usuario `anonymous` y como `passwd` la dirección de *e-mail* personal. Existen servidores que no aceptan conexiones desde máquinas que no están declaradas correctamente en el servicio de nombre (*dns*), así como algunas que no permiten la entrada a usuarios que no se identifican correctamente. Dada la sobrecarga que existe, muchos de los servidores tienen limitado el número de usuarios que pueden acceder simultáneamente.

1.16.6. WWW.

WWW son las siglas de *World-Wide Web*. Este servicio permite el acceso a información entrelazada (dispone de un texto donde un término puede conducir a otro texto): *hyperlinks*. Los archivos están escritos, principalmente, en un lenguaje denominado *html*. Para acceder a este servicio es necesario disponer de un lector de dicho lenguaje conocido como *browser* o navegador. Destacan actualmente: Google-Chrome,Iceweasel, Firefox, Opera, Safari (para MAC) y el simple pero muy rápido Lynx.

1.17. Impresión.

Cuando se quiere obtener una copia impresa de un archivo se emplea el comando `lpr`.

`lpr file` - Envía el archivo `file` a la cola de impresión por defecto. Si la cola está activada, la impresora lista y ningún trabajo por encima del enviado, nuestro trabajo será procesado de forma automática.

A menudo existen varias posibles impresoras a las que poder enviar los trabajos. Para seleccionar una impresora en concreto (en vez de la por defecto) se emplea el modificador: `lpr -Pimpresora`, siendo `impresora` el nombre lógico asignado a esta otra impresora. Para recibir una lista de las posibles impresoras de un sistema, así como su estado, se puede emplear el comando `/usr/sbin/lpc status`. La lista de impresoras y su configuración también está disponible en el archivo `/etc/printcap`.

Otras órdenes para la manipulación de la cola de impresión son:

- `lpq [-Pimpresora]`, permite examinar el estado de una determinada cola (para ver la cantidad de trabajos sin procesar de ésta, por ejemplo).
- `lprm [-Pimpresora] jobnumber`, permite eliminar un trabajo de la cola de impresión.

Uno de los lenguajes de impresión gráfica más extendidos en la actualidad es *PostScript*. La extensión de los archivos *PostScript* empleada es `.ps`. Un archivo *PostScript* puede ser visualizado e impreso mediante los programas: `gv`, `gnome-gv` o `ghostview`. Por ello muchas de las impresoras actuales sólo admiten la impresión en dicho formato.

En caso de desear imprimir un archivo `ascii` deberá previamente realizarse la conversión a *PostScript* empleando la orden `a2ps`: `a2ps file.txt` Esta orden envía a la impresora el archivo `ascii file.txt` formateado a 2 páginas por hoja. Otro programa que permite convertir un archivo `ascii` en postscript es `enscript`.

Otro tipo de archivos ampliamente difundido y que habitualmente se necesita imprimir es el conocido como *Portable Document Format*. Este tipo de archivo poseen una extensión `.pdf` y pueden ser visualizados e impresos usando aplicaciones tales como: `xpdf`, `acroread` o `evince`.

1.18. Compresión.

`tar`

Este comando permite la creación/extracción de archivos contenidos en un único archivo denominado `tarfile` (o `tarball`). Este `tarfile` suele ser luego comprimido con `gzip`, la versión de compresión **gnu**,[35] o bien con `bzip2`.

La acción a realizar viene controlada por el primer argumento:

- `c` (Create) creación
- `x` (eXtract) extracción
- `t` (lisT) mostrar contenido
- `r` añadir al final
- `u` (Update) añadir aquellos archivos que no se hallen en el tarfile o que hayan sido modificados con posterioridad a la versión que aparece.

A continuación se muestran algunas de las opciones:

- `v` Verbose (indica qué archivos son agregados a medida que son procesados)
- `z` Comprimir o descomprimir el contenido con `gzip`.
- `j` Comprimir o descomprimir el contenido con `bzip2`.
- `f` File: permite especificar el archivo para el tarfile.

Veamos algunos ejemplos:

```
tar cvf simul.tar *.dat
```

Genera un archivo `simul.tar` que contiene todos los archivos que terminen en `.dat` del directorio actual. A medida que se va realizando indica el tamaño en bloques de cada archivo añadido modo *verbose*.

```
tar czvf simul.tgz *.dat
```

Igual que en el caso anterior, pero el archivo generado simul.tgz ha sido comprimido empleando gzip.

```
tar tvf simul.tar
```

Muestra los archivos contenidos en el tarfile `simul.tar`.

```
tar xvf simul.tar
```

Extrae todos los archivos contenidos en el tarfile `simul.tar`.

[35] **gnu** es un acrónimo recursivo, significa: **gnu**'s Not UNIX! **gnu** es el nombre del producto de la *Free Software Foundation*, una organización dedicada a la creación de programas compatibles con UNIX algunos mejorado respecto a los estándars, y de libre distribución. La distribución de Linux **gnu** es **debian**.

1.18.1. Distintos algoritmos de compresión.

A menudo necesitamos comprimir un archivo para disminuir su tamaño, o bien crear un respaldo (*backup*) de una determinada estructura de directorios. Se comentan a continuación una serie de comandos que permiten ejecutar dichas acciones.

El compresor `compress` está relativamente fuera de uso[36] pero aún podemos encontrarnos con archivos comprimidos por él.

- `compress file` : comprime el archivo, creando el archivo `file.Z`. Destruye el archivo original.
- `uncompress file.Z` : descomprime el archivo, creando el archivo `file`. Destruye el archivo original.
- `zcat file.Z` : muestra por el `stdout` el contenido descomprimido del archivo (sin destruir el original).

Otra alternativa de compresor mucho más usada es `gzip`, el compresor de `GNU` que posee una mayor razón de compresión que `compress`. Veamos los comandos:

- `gzip file` : comprime el archivo, creando el archivo `file.gz`. Destruye el archivo original.
- `gunzip file.gz` : descomprime el archivo, creando el archivo `file`. Destruye el archivo original.
- `zless file.gz` : muestra por el `stdout` el contenido descomprimido del archivo paginado por `less`.

La extensión empleada en los archivos comprimidos con `gzip` suele ser `.gz`, pero a veces se usa `.gzip`. Adicionalmente el programa `gunzip` también puede descomprimir archivos creados con `compress`.

La opción con mayor tasa de compresión que `gzip` es `bzip2` y su descompresor `bunzip2`.

- `bzip2 file` : comprime el archivo, creando el archivo `file.bz2`. Destruye el archivo original.
- `bunzip2 file.bz2` : descomprime el archivo, creando el archivo `file`. Destruye el archivo original.
- `bzcat file.bz2` : muestra por el `stdout` el contenido descomprimido del archivo. Debemos usar un paginador, adicionalmente, para verlo por páginas.

[36] Este comando no se incluye en la instalación básica. Debemos cargar el paquete `ncompress` para tenerlo

La extensión usada en este caso es `.bz2`. El *kernel* de Linux se distribuye en formato `bzip2`. También existe una versión paralelizada llamada `pbzip2`. Uno de los mejores algoritmos de compresión está disponible para Linux en el programa `p7zip`. Veamos un ejemplo: un archivo `linux-2.6.18.tar` que contiene el kernel 2.6.18 de Linux que tiene un tamaño de 230 Mb. Los resultados al comprimirlo con `compress`, `gzip`, `bzip2` y `7za` son: `linux-2.6.18.tar.Z` 91 Mb, `linux-2.6.18.tar.gz` 51 Mb, `linux-2.6.18.tar.bz2` 40 Mb y `linux-2.6.18.tar.7z` 33 Mb.[37]

Compresor	Tamaño
Sin comprimir	230 Mb
`compress`	91 Mb
`gzip`	51 Mb
`bzip2`	40 Mb
`7z`	33 Mb

Cuadro 1.1: Tabla de compresión del archivo `linux-2.6.18.tar` que contiene el kernel 2.6.18 de Linux.

Existen también versiones de los compresores compatibles con otros sistemas operativos: `zip`, `unzip`, `unarj`, `lha`, `rar` y `zoo`.

En caso que se desee crear un archivo comprimido con una estructura de directorios debe ejecutarse la orden:
`tar cvzf nombre.tgz directorio` o bien `tar cvjf nombre.tbz directorio`
En el primer caso comprime con `gzip` y en el segundo con `bzip2`. Para descomprimir y restablecer la estructura de directorio almacenada se usan los comandos:
`tar xvzf nombre.tgz directorio`
si se realizó la compresión con `gzip` o bien
`tar xvjf nombre.tbz directorio`
si se realizó la compresión con `bzip2`.

[37] Los comandos `gzip` y `bzip2` fueron dados con la opción `-best` para lograr la mayor compresión. El comando usado para la compresión con `7z` fue: `7za a -t7z -m0=lzma -mx=9 -mfb=64 -md=32m -ms=on file.tar.7z file.tar`, note la nueva extensión `7z`. Para descomprimir con `7z` basta `7z e file.tar.7z`

Capítulo 2

Gráfica.

versión 4.12, 24 de Octubre del 2003

En este capítulo quisiéramos mostrar algunas de las posibilidades gráficas presentes en Linux. Nuestra intensión es cubrir temas como la visualización, conversión, captura y creación de archivos gráficos. Sólo mencionaremos las aplicaciones principales en cada caso centrándonos en sus posibilidades más que en su utilización específica, ya que la mayoría posee una interfase sencilla de manejar y una amplia documentación.

2.1. Visualización de archivos gráficos.

Si disponemos de un archivo gráfico conteniendo algún tipo de imagen lo primero que es importante determinar es en qué tipo de formato gráfico está codificada. Existe un número realmente grande de diferentes tipos de codificaciones de imágenes, cada una de ellas se considera un formato gráfico. Por razones de reconocimiento inmediato del tipo de formato gráfico se suelen incluir en el nombre del archivo, que contiene la imagen, un trío de letras finales, conocidas como la extensión, que representan el formato. Por ejemplo: bmp, tiff, jpg, ps, eps, fig, gif entre muchas otras, si uno quiere asegurarse puede dar el comando:

```
jrogan@huelen:~$file mono.jpg
mono.jpg: JPEG image data, JFIF standard 1.01, resolution (DPCM), 72 x 72
```

¿De qué herramientas disponemos en Linux para visualizar estas imágenes? La respuesta es que en Linux disponemos de variadas herramientas para este efecto.

Si se trata de archivos de tipo *PostScript* o *Encapsulated PostScript*, identificados por la extensión `ps` o `eps`, existen las aplicaciones `gv`, `gnome-gv` o `kghostview`, todos programas que nos permitirán visualizar e imprimir este tipo de archivos. Si los archivos son tipo *Portable Document Format*, con extensión `pdf`, tenemos las aplicaciones `gv`, `acroread` o `xpdf`, Con todas ellas podemos ver e imprimir dicho formato. Una mención especial requieren los archivos *DeVice Independent* con extensión `dvi` ya que son el resultado de la compilación de un documento TEX o LATEX, para este tipo de archivo existen las aplicaciones `xdvi`, `advi`, `gxdvi` y `kdvi` por nombrar las principales. La aplicación `xdvi` sólo permite visualizar estos archivos y no imprimirlos, la mayoría de las otras permiten imprimirlo directamente. Si usa `xdvi` y

desea imprimir el documento debe transformar a `ps` vía `dvips` y luego se imprime como cualquier otro *Postscript*.

Para la gran mayoría de formatos gráficos más conocidos y usualmente usados para almacenar fotos existen otra serie se programas especializados en visualización que son capaces de entender la mayoría de los formatos más usados. Entre estos programas podemos mencionar: *Eye of Gnome* (`eog`), *Electric Eyes* (`eeyes`), `kview` o `display`. Podemos mencionar que aplicaciones como `display` entienden sobre ochenta formatos gráficos distintos entre los que se encuentran la mayoría de los formatos conocidos más otros como `ps`, `eps`, `pdf`, `fig`, `html`, entre muchos otros. Una mención especial merece el utilitario `gthumb` que nos permite hacer un *preview* de un directorio con muchas imagenes de manera muy fácil.

2.2. Modificando imágenes

Si queremos modificaciones como rotaciones, ampliaciones, cortes, cambios de paleta de colores, filtros o efectos sencillos, `display` es la herramienta precisa. Pero si se desea intervenir la imagen en forma profesional, el programa `gimp` es el indicado. El nombre `gimp` viene de *GNU Image Manipulation Program*. Se puede usar esta aplicación para editar y manipular imágenes. Pudiendo cargar y salvar en una variedad de formatos, lo que permite usarlo para convertir entre ellos. La aplicación `gimp` puede también ser usado como programa de pintar, de hecho posee una gran variedad de herramientas en este sentido, tales como brocha de aire, lápiz clonador, tijeras inteligentes, curvas bezier, etc. Además, permite incluir *plugins* que realizan gran variedad de manipulaciones de imagen. Como hecho anecdótico podemos mencionar que la imagen oficial de Tux, el pingüino mascota de Linux, fue creada en `gimp`. Sin embargo, si `gimp` le parece muy profesional o usted sólo necesita un programa para dibujar en el cual se entretenga su hermano menor `tuxpaint` es la alternativa.

2.3. Conversión entre formatos gráficos.

El problema de transformar de un formato a otro es una situación usual en el trabajo con archivos gráficos. Muchos *softwares* tienen salidas muy restringidas en formato o bien usan formatos arcaicos (`gif`) por ejemplo. De ahí que se presenta la necesidad de convertir estos archivos de salida en otros formatos que nos sean más manejables o prácticos. Como ya se mencionó, `gimp` puede ser usado para convertir entre formatos gráficos. También `display` permite este hecho. Sin embargo, en ambos casos la conversión es vía menús, lo cual lo hace engorroso para un gran número de conversiones e imposible para conversiones de tipo automático. Existe un programa llamado `convert` que realiza conversiones desde la línea de comando. Este programa junto con `display`, `import` y varios otros forman la *suite* gráfica *ImageMagick*, una de las más importantes en UNIX, en general, y en especial en Linux y que ya ha sido migrada a otras plataformas. Además, de la clara ventaja de automatización que

proporciona `convert`, posee otro aspecto interesante, puede convertir un grupo de imágenes asociadas en una secuencia de animación o película. Veamos la sintaxis para este programa:

```
user@host:~/imagenes$convert cockatoo.tiff cockatoo.jpg

user@host:~/secuencias$convert -delay 20 dna*.png dna.mng
```

En el primer caso convierte el archivo `cockatoo` de formato `tiff` a formato `jpg`. En el segundo, a partir de un conjunto de archivos con formato `png` llamados `dna` más un número correlativo, crea una secuencia animada con imágenes que persisten por 20 centésimas de segundos en un formato conocido como `mng`.

2.4. Captura de pantalla.

A menudo se necesita guardar imágenes que sólo se pueden generar a tiempo de ejecución, es decir, mientras corre nuestro programa genera la imagen pero no tiene un mecanismo propio para exportarla o salvarla como imagen. En este caso necesitamos capturar la pantalla y poderla almacenar en un archivo para el cual podamos elegir el formato. Para estos efectos existe un programa, miembro también de la *suite ImageMagick*, llamado `import` que nos permite hacer el trabajo. La sintaxis del comando es

```
import figure.eps

import -window root root.jpg
```

En el primer caso uno da el comando en un terminal y queda esperando hasta que uno toque alguna de las ventanas, la cual es guardada en este caso en un archivo `figure.eps` en formato *PostScript*. La extensión le indica al programa qué formato usar para almacenar la imagen. En el segundo caso uno captura la pantalla completa en un archivo `root.jpeg`. Este comando puede ser dado desde la consola de texto para capturar la imagen completa en la pantalla gráfica.

2.5. Creando imágenes.

Para imágenes artísticas sin duda la alternativa es `gimp`, todo le que se dijo respecto a sus posibilidades para modificar imágenes se aplica también en el caso de crearlas. En el caso de necesitar imágenes más bien técnicas como esquemas o diagramas o una ilustración para aclarar un problema las alternativas pueden ser `xfig`, `sodipodi` o `sketch` todas herramientas vectoriales muy poderosa. Este tipo de programas son manejados por medio de menús y permiten dibujar y manipular objetos interactivamente. Las imágenes pueden ser salvadas, en formato propios y posteriormente editadas. La gran ventaja de estos programas es que

trabaja con objetos y no con bitmaps. Además, puede exportar las imágenes a otros formatos: *PostScript* o *Encapsulated PostScript* o bien `gif` o `jpeg`.

Habitualmente los dibujos necesarios para ilustrar problemas en Física en tareas, pruebas y apuntes son realizados con *software* de este tipo, principalmente `xfig`, luego exportados a *PostScript* e incluídos en los respectivos archivos LaTeX. También existe una herramienta extremadamente útil que permite convertir un archivo *PostScript*, generado de cualquier manera, a un archivo `fig` que puede ser editado y modificado. Esta aplicación que transforma se llama `pstoedit` y puede llegar a ser realmente práctica. Otra herramienta interesante es `autotrace` que permite pasar una figura en *bitmap* a forma vectorial.

Una aparente limitación de este tipo de *software* es que se podría pensar que no podemos incluir curvas analíticas, es decir, si necesitamos ilustrar una función gaussiana no podemos pretender "dibujarla" con las herramientas de que disponen. Sin embargo, este problema puede ser resuelto ya que *software* que grafica funciones analíticas, tales como `gnuplot`, permite exportar en formato que entienden los programas vectoriales (`fig`, por ejemplo) luego podemos leer el archivo y editarlo. Además, `xfig` permite importar e incluir imágenes del tipo *bitmap*, agregando riqueza a los diagramas que puede generar.

Una característica importante de este tipo de programas es que trabajen por capas, las cuales son tratadas independientemente, uno puede poner un objeto sobre otro o por debajo de otro logrando diferentes efectos. Algunos programas de presentación gráficos basados en LaTeX y `pdf` están utilizando esta capacidad en `xfig` para lograr animaciones de imágenes.

Finalmente el programa `xfig` permite construir una biblioteca de objetos reutilizables ahorrando mucho trabajo. Por ejemplo, si uno dibuja los elementos de un circuito eléctrico y los almacena en el lugar de las bibliotecas de imágenes podrá incluir estos objetos en futuros trabajos. El programa viene con varias bibliotecas de objetos listas para usar.

2.6. Graficando funciones y datos.

Existen varias aplicaciones que permiten graficar datos de un archivo, entre las más populares están: `gnuplot`, `xmgrace` y `SciGraphica`. La primera está basada en la línea de comando y permite gráficos en 2 y 3 dimensiones, pudiendo además, graficar funciones directamente sin pasar por un archivo de datos. Las otras dos son aplicaciones basadas en menús que permiten un resultado final de mucha calidad y con múltiples variantes. La debilidad en el caso de `xmgrace` es que sólo hace gráficos bidimensionales.

El programa `gnuplot` se invoca de la línea de comando y da un *prompt* en el mismo terminal desde el cual se puede trabajar, veamos una sesión de `gnuplot`:

```
jrogan@huelen:~$ gnuplot

        G N U P L O T
        Version 3.7 patchlevel 2
        last modified Sat Jan 19 15:23:37 GMT 2002
```

```
        System: Linux 2.4.19

        Copyright(C) 1986 - 1993, 1998 - 2002
        Thomas Williams, Colin Kelley and many others

        Type ‘help‘ to access the on-line reference manual
        The gnuplot FAQ is available from
        http://www.gnuplot.info/gnuplot-faq.html

        Send comments and requests for help to <info-gnuplot@dartmouth.edu>
        Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>


Terminal type set to 'x11'
gnuplot> plot sqrt(x)
gnuplot> set xrange[0:5]
gnuplot> set xlabel" eje de las x"
gnuplot> replot
gnuplot> set terminal postscript
Terminal type set to 'postscript'
Options are 'landscape noenhanced monochrome dashed defaultplex "Helvetica" 14'
gnuplot> set output "mygraph.ps"
gnuplot> replot
gnuplot> set terminal X
Terminal type set to 'X11'
Options are '0'
gnuplot> set xrange[-2:2]
gnuplot> set yrange[-2:2]
gnuplot> splot exp(-x*x-y*y)
gnuplot> plot "myfile.dat" w l
gnuplot> exit
jrogan@huelen:~$
```

En el caso de `xmgrace` y `SciGraphica` mucho más directo manejarlo ya que está basado en menús. Además, existe abundante documentación de ambos *softwares*. El *software* `SciGraphica` es una aplicación de visualización y análisis de data científica que permite el despliegue de gráficos en 2 y 3 dimensiones, además, exporta los resultados a formato *PostScript*. Realmente esta aplicación nació como un intento de clonar el programa comercial `origen` no disponible para Linux.

Capítulo 3

El sistema de preparación de documentos TEX .

versión 5.0, 30 de Julio del 2003

3.1. Introducción.

TEX es un procesador de texto o, mejor dicho, un avanzado sistema de preparación de documentos, creado por Donald Knuth, que permite el diseño de documentos de gran calidad, conteniendo textos y fórmulas matemáticas. Años después, LATEX fue desarrollado por Leslie Lamport, facilitando la preparación de documentos en TEX, gracias a la definición de "macros" o conjuntos de comandos de fácil uso.

LATEX tuvo diversas versiones hasta la 2.09. Actualmente, LATEX ha recibido importantes modificaciones, siendo la distribución actualmente en uso y desarrollo LATEX 2_ε, una versión transitoria en espera de que algún día se llegue a la nueva versión definitiva de LATEX, LATEX3. En estas páginas cuando digamos LATEX nos referiremos a la versión actual, LATEX 2_ε. Cuando queramos hacer referencia a la versión anterior, que debería quedar progresivamente en desuso, diremos explícitamente LATEX 2.09.

3.2. Archivos.

El proceso de preparación de un documento LATEX consta de tres pasos:

1. Creación de un archivo con extensión `tex` con algún editor.
2. Compilación del archivo `tex`, con un comando del tipo `latex <archivo>.tex` o `latex <archivo>`. Esto da por resultado tres archivos adicionales, con el mismo nombre del archivo original, pero con extensiones distintas:

   a) `dvi`. Es el archivo procesado que podemos ver en pantalla o imprimir. Una vez compilado, este archivo puede ser enviado a otro computador, para imprimir en

otra impresora, o verlo en otro monitor, independiente de la máquina (de donde su extensión `dvi`, *device independent*).

b) `log`. Aquí se encuentran todos los mensajes producto de la compilación, para consulta si es necesario (errores encontrados, memoria utilizada, mensajes de advertencia, etc.).

c) `aux`. Contiene información adicional que, por el momento, no nos interesa.

3. Visión en pantalla e impresión del archivo procesado a través de un programa anexo (`xdvi` o `dvips`, por ejemplo), capaz de leer el `dvi`.

3.3. Input básico.

3.3.1. Estructura de un archivo.

En un archivo no pueden faltar las siguientes líneas:

```
\documentclass[12pt]{article}

\begin{document}

\end{document}
```

Haremos algunas precisiones respecto a la primera línea más tarde. Lo importante es que una línea de esta forma debe ser la primera de nuestro archivo. Todo lo que se encuentra antes de `\begin{document}` se denomina *preámbulo*. El texto que queramos escribir va entre `\begin{document}` y `\end{document}`. Todo lo que se encuentre después de `\end{document}` es ignorado.

3.3.2. Caracteres.

Pueden aparecer en nuestro texto todos los caracteres del código ASCII no extendido (teclado inglés usual): letras, números y los signos de puntuación:

```
.  :  ;  ,  ?  !  ‘  ’  (  )  [  ]  -  /  * @
```

Los caracteres especiales:

```
#  $  %  &  ~  _  ^  \  {  }
```

tienen un significado específico para LaTeX. Algunos de ellos se pueden obtener anteponiéndoles un *backslash*:

```
#  \#    $  \$    %  \%    &  \&    {  \{    }  \}
```

Los caracteres

```
+  =  |  <  >
```

generalmente aparecen en fórmulas matemáticas, aunque pueden aparecer en texto normal. Finalmente, las comillas dobles (") casi nunca se usan.

Los espacios en blanco y el fin de línea son también caracteres (invisibles), que LaTeX considera como un mismo carácter, que llamaremos espacio, y que simbolizaremos ocasionalmente como ␣ .

Para escribir en castellano requeriremos además algunos signos y caracteres especiales:

```
ñ  \~n     á  \'a     í  \'{\i}     ü  \"u     ¡  !‘     ¿  ?‘
```

3.3.3. Comandos.

Todos los comandos comienzan con un backslash, y se extienden hasta encontrar el primer carácter que no sea una letra (es decir, un espacio, un número, un signo de puntuación o matemático, etc.).

3.3.4. Algunos conceptos de estilo.

LaTeX es consciente de muchas convenciones estilísticas que quizás no apreciamos cuando leemos textos bien diseñados, pero las cuales es bueno conocer para aprovecharlas.

a) Observemos la siguiente palabra: fino. Esta palabra fue generada escribiendo simplemente `fino`, pero observemos que las letras 'f' e 'i' no están separadas, sino que unidas artísticamente. Esto es una *ligadura*, y es considerada una práctica estéticamente preferible. LaTeX sabe esto e inserta este pequeño efecto tipográfico sin que nos demos cuenta.

b) Las comillas de apertura y de cierre son distintas. Por ejemplo: 'insigne' (comillas simples) o "insigne" (comillas dobles). Las comillas de apertura se hacen con uno o con dos acentos graves (‘), para comillas simples o dobles, respectivamente, y las de cierre con acentos agudos (’): `‘insigne’`, `‘‘insigne’’`. No es correcto entonces utilizar las comillas dobles del teclado e intentar escribir `"insigne"` (el resultado de esto es el poco estético ïnsigne").

c) Existen tres tipos de guiones:

Corto	Saint-Exupéry	-	(entre palabras, corte en sílabas al final de la línea)
Medio	páginas 1–2	--	(rango de números)
Largo	un ejemplo —como éste	---	(puntuación, paréntesis)

d) LaTeX inserta después de un punto seguido un pequeño espacio adicional respecto al espacio normal entre palabras, para separar sutilmente frases. Pero, ¿cómo saber que un punto termina una frase? El criterio que utiliza es que todo punto termina una frase cuando va precedido de una minúscula. Esto es cierto en la mayoría de los casos, así como es cierto que generalmente cuando un punto viene después de una mayúscula no hay fin de frase:

> China y U.R.S.S. estuvieron de acuerdo. Sin embargo...

Pero hay excepciones:

> En la pág. 11 encontraremos noticias desde la U.R.S.S. Éstas fueron entregadas...

Cuando estas excepciones se producen, nosotros, humanos, tenemos que ayudarle al computador, diciéndole que, aunque hay un punto después de la "g", no hay un fin de frase, y que el punto después de la última "S" sí termina frase. Esto se consigue así:

```
En la p\'ag.\ 11 encontraremos noticias desde la
U.R.S.S\@. \'Estas fueron entregadas...
```

d) Énfasis de texto:

Éste es un texto *enfatizado*.	`\'Este es un texto {\em enfatizado}.`
Otro texto *enfatizado*.	`Otro texto \emph{enfatizado}.`

Al enfatizar, pasamos temporalmente a un tipo de letra distinto, la *itálica*. Esta letra es ligeramente inclinada hacia adelante, lo cual puede afectar el correcto espaciado entre palabras. Comparemos, por ejemplo:

Quiero *hoy* mi recompensa.	`Quiero {\em hoy} mi recompensa.`
Quiero *hoy* mi recompensa.	`Quiero {\em hoy\/} mi recompensa.`
Quiero *hoy* mi recompensa.	`Quiero \emph{hoy} mi recompensa.`

La segunda y tercera frase tienen un pequeño espacio adicional después de "hoy", para compensar el espacio entre palabras perdido por la inclinación de la itálica. Este pequeño espacio se denomina *corrección itálica*, y se consigue usando `\emph`, o, si se usa `\em`, agregando `\/` antes de cerrar el paréntesis cursivo. La corrección itálica es innecesaria cuando después del texto enfatizado viene un punto o una coma. LaTeX advierte esto y omite el espacio adicional aunque uno lo haya sugerido.

3.3.5. Notas a pie de página.

```
Insertemos una nota a pie de p\'agina.\footnote{Como \'esta.}
```

LaTeX colocará una nota a pie de página[1] en el lugar apropiado.

3.3.6. Fórmulas matemáticas.

LaTeX distingue dos modos de escritura: un modo de texto, en el cual se escriben los textos usuales como los ya mencionados, y un modo matemático, dentro del cual se escriben las fórmulas. Cualquier fórmula *debe* ser escrita dentro de un modo matemático, y si algún símbolo matemático aparece fuera del modo matemático el compilador acusará un error.

Hay tres formas principales para acceder al modo matemático:

a) `$x+y=3$`

b) `$$xy=8$$`

c)
```
\begin{equation}
x/y=5
\end{equation}
```

Estas tres opciones generan, respectivamente, una ecuación en el texto: $x + y = 3$, una ecuación separada del texto, centrada en la página:

$$xy = 8$$

y una ecuación separada del texto, numerada:

$$x/y = 5 \tag{3.1}$$

Es importante notar que al referirnos a una variable matemática en el texto debemos escribirla en modo matemático:

Decir que la incógnita es x es incorrecto. No: la incógnita es x.	`Decir que la inc{\'o}gnita es x es incorrecto. No: la inc{\'o}gnita es $x$.`

3.3.7. Comentarios.

Uno puede hacer que el compilador ignore parte del archivo usando `%`. Todo el texto desde este carácter hasta el fin de la línea correspondiente será ignorado (incluyendo el fin de línea).

Un pequeño comentario.	`Un peque{\~n}o co% Texto ignorado mentario.`

[1] Como ésta.

3.3.8. Estilo del documento.

Las características generales del documento están definidas en el preámbulo. Lo más importante es la elección del *estilo*, que determina una serie de parámetros que al usuario normal pueden no importarle, pero que son básicas para una correcta presentación del texto: ¿Qué márgenes dejar en la página? ¿Cuánto dejar de sangría? ¿Tipo de letra? ¿Distancia entre líneas? ¿Dónde poner los números de página? Y un largo etcétera.

Todas estas decisiones se encuentran en un *archivo de estilo* (extensión `cls`). Los archivos standard son: `article`, `report`, `book` y `letter`, cada uno adecuado para escribir artículos cortos (sin capítulos) o más largos (con capítulos), libros y cartas, respectivamente.

La elección del estilo global se hace en la primera línea del archivo:[2]

```
\documentclass{article}
```

Esta línea será aceptada por el compilador, pero nos entregará un documento con un tamaño de letra pequeño, técnicamente llamado de 10 puntos ó 10pt (1pt = 1/72 pulgadas). Existen tres tamaños de letra disponibles: 10, 11 y 12 pt. Si queremos un tamaño de letra más grande, como el que tenemos en este documento, se lo debemos indicar en la primera línea del archivo:

```
\documentclass[12pt]{article}
```

Todas las decisiones de estilo contenidas dentro del archivo `cls` son modificables, existiendo tres modos de hacerlo:

a) Modificando el archivo `cls` directamente. Esto es poco recomendable, porque dicha modificación (por ejemplo, un cambio de los márgenes) se haría extensible a todos los archivos compilados en nuestro computador, y esto puede no ser agradable, ya sea que nosotros seamos los únicos usuarios o debamos compartirlo. Por supuesto, podemos deshacer los cambios cuando terminemos de trabajar, pero esto es tedioso.

b) Introduciendo comandos adecuados en el preámbulo. Ésta es la opción más recomendable y la más usada. Nos permite dominar decisiones específicas de estilo válidas sólo para el archivo que nos interesa.

c) Creando un nuevo archivo `cls`. Esto es muy recomendable cuando las modificaciones de estilo son abundantes, profundas y deseen ser reaprovechadas. Se requiere un poco de experiencia en LaTeX para hacerlo, pero a veces puede ser la única solución razonable.

En todo caso, la opción a usar en la gran mayoría de los casos es la b) (Sec. 3.9).

[2] En LaTeX 2.09 esta primera línea debe ser `\documentstyle[12pt]article`, y el archivo de estilo tiene extensión `sty`. Intentar compilar con LaTeX 2.09 un archivo que comienza con `\documentclass` da un error. Por el contrario, la compilación con LaTeX 2ε de un archivo que comienza con `\documentstyle` no genera un error, y LaTeX entra en un *modo de compatibilidad*. Sin embargo, interesantes novedades de LaTeX 2ε respecto a LaTeX 2.09 se pierden.

3.3.9. Argumentos de comandos.

Hemos visto ya algunos comandos que requieren argumentos. Por ejemplo: `\begin{equation}`, `\documentclass[12pt]{article}`, `\footnote{Nota}`. Existen dos tipos de argumentos:

1. **Argumentos obligatorios.** Van encerrados en paréntesis cursivos: `\footnote{Nota}`, por ejemplo. Es obligatorio que después de estos comandos aparezcan los paréntesis. A veces es posible dejar el interior de los paréntesis vacío, pero en otros casos el compilador reclamará incluso eso (`\footnote{}` no genera problemas, pero `\documentclass{}` sí es un gran problema).

   Una propiedad muy general de los comandos de LaTeX es que las llaves de los argumentos obligatorios se pueden omitir cuando dichos argumentos tienen sólo un carácter. Por ejemplo, `\~n` es equivalente a `\~{n}`. Esto permite escribir más fácilmente muchas expresiones, particularmente matemáticas, como veremos más adelante.

2. **Argumentos opcionales.** Van encerrados en paréntesis cuadrados. Estos argumentos son omitibles, `\documentclass[12pt]...` . Ya dijimos que `\documentclass{article}` es aceptable, y que genera un tamaño de letra de 10pt. Un argumento en paréntesis cuadrados es una opción que modifica la decisión default del compilador (en este caso, lo obliga a usar 12pt en vez de sus instintivos 10pt).

3.3.10. Título.

Un título se genera con:

```
\title{Una breve introducci\'on}
\author{V\'{\i}ctor Mu\~noz}
\date{30 de Junio de 1998}
\maketitle
```

`\title`, `\author` y `\date` pueden ir en cualquier parte (incluyendo el preámbulo) antes de `\maketitle`. `\maketitle` debe estar después de `\begin{document}`. Dependiendo de nuestras necesidades, tenemos las siguientes alternativas:

a) Sin título:

```
\title{}
```

b) Sin autor:

```
\author{}
```

c) Sin fecha:

```
\date{}
```

d) Fecha actual (en inglés): omitir `\date`.

e) Más de un autor:

```
\author{Autor_1 \and Autor_2 \and Autor_3}
```

Para artículos cortos, LaTeX coloca el título en la parte superior de la primera página del texto. Para artículos largos, en una página separada.

3.3.11. Secciones.

Los títulos de las distintas secciones y subsecciones de un documento (numerados adecuadamente, en negrita, como en este texto) se generan con comandos de la forma:

```
\section{Una secci\'on}
\subsection{Una subsecci\'on}
```

Los comandos disponibles son (en orden decreciente de importancia):

`\part`	`\subsection`	`\paragraph`
`\chapter`	`\subsubsection`	`\subparagraph`
`\section`		

Los más usados son `\chapter`, `\section`, `\subsection` y `\subsubsection`. `\chapter` sólo está disponible en los estilos `report` y `book`.

3.3.12. Listas.

Los dos modos usuales de generar listas:

a) Listas numeradas (ambiente `enumerate`):

1. Nivel 1, ítem 1.
2. Nivel 1, ítem 2.
    a) Nivel 2, ítem 1.
        1) Nivel 3, ítem 1.
3. Nivel 1, ítem 3.

```
\begin{enumerate}
\item Nivel 1, \'{\i}tem 1.
\item Nivel 1, \'{\i}tem 2.
\begin{enumerate}
\item Nivel 2, \'{\i}tem 1.
\begin{enumerate}
\item Nivel 3, \'{\i}tem 1.
\end{enumerate}
\end{enumerate}
\item Nivel 1, \'{\i}tem 3.
\end{enumerate}
```

b) Listas no numeradas (ambiente `itemize`):

- Nivel 1, ítem 1.
- Nivel 1, ítem 2.
  - Nivel 2, ítem 1.
    - Nivel 3, ítem 1.
- Nivel 1, ítem 3.

```
\begin{itemize}
\item Nivel 1, {\'\i}tem 1.
\item Nivel 1, {\'\i}tem 2.
\begin{itemize}
\item Nivel 2, {\'\i}tem 1.
\begin{itemize}
\item Nivel 3, {\'\i}tem 1.
\end{itemize}
\end{itemize}
\item Nivel 1, {\'\i}tem 3.
\end{itemize}
```

Es posible anidar hasta tres niveles de listas. Cada uno usa tipos distintos de rótulos, según el ambiente usado: números arábes, letras y números romanos para `enumerate`, y puntos, guiones y asteriscos para `itemize`. Los rótulos son generados automáticamente por cada `\item`, pero es posible modificarlos agregando un parámetro opcional:

a) Nivel 1, ítem 1.
b) Nivel 1, ítem 2.

```
\begin{enumerate}
\item[a)] Nivel 1, \'{\i}tem 1.
\item[b)] Nivel 1, \'{\i}tem 2.
\end{enumerate}
```

`\item` es lo primero que debe aparecer después de un `\begin{enumerate}` o `\begin{itemize}`.

3.3.13. Tipos de letras.

Fonts.

Los fonts disponibles por default en LaTeX son:

roman	*italic*	Small Caps
boldface	*slanted*	`typewriter`
sans serif		

Los siguientes modos de cambiar fonts son equivalentes:

texto	`{\rm texto}`	`\textrm{texto}`
texto	`{\bf texto}`	`\textbf{texto}`
texto	`{\sf texto}`	`\textsf{texto}`
texto	`{\it texto}`	`\textit{texto}`
texto	`{\sl texto}`	`\textsl{texto}`
Texto	`{\sc Texto}`	`\textsc{texto}`
`texto`	`{\tt texto}`	`\texttt{texto}`

`\rm` es el default para texto normal; `\it` es el default para texto enfatizado; `\bf` es el default para títulos de capítulos, secciones, subsecciones, etc.

`\textrm`, `\textbf`, etc., sólo permiten cambiar porciones definidas del texto, contenido entre los paréntesis cursivos. Con `\rm`, `\bf`, etc. podemos, omitiendo los paréntesis, cambiar el font en todo el texto posterior:

Un cambio local de fonts *y uno global, interminable e infinito...*	`Un cambio {\sf local} de fonts \sl y uno global, interminable e infinito...`

También es posible tener combinaciones de estos fonts, por ejemplo, ***bold italic***, pero no sirven los comandos anteriores, sino versiones modificadas de `\rm`, `\bf`, etc.:

```
\rmfamily
\sffamily
\ttfamily
\mdseries
\bfseries
\upshape
\itshape
\slshape
\scshape
```

Por ejemplo:

texto	`{\bfseries\itshape texto}`
texto	`{\bfseries\upshape texto}` (= `{\bf texto}`)
TEXTO	`{\ttfamily\scshape texto}`
texto	`{\sffamily\bfseries texto}`
texto	`{\sffamily\mdseries texto}` (= `{\sf texto}`)

Para entender el uso de estos comandos hay que considerar que un font tiene tres *atributos*: `family` (que distingue entre `rm`, `sf` y `tt`), `series` (que distingue entre `md` y `bf`), y `shape` (que distingue entre `up`, `it`, `sl` y `sc`). Cada uno de los comandos `\rmfamily`, `\bfseries`, etc., cambia sólo uno de estos atributos. Ello permite tener versiones mixtas de los fonts, como un *slanted sans serif*, imposible de obtener usando los comandos `\sl` y `\sf`. Los defaults para el texto usual son: `\rmfamily`, `\mdseries` y `\upshape`.

Tamaño.

Los tamaños de letras disponibles son:

ó	\'o	õ	\~o	ǒ	\v o	o̧	\c o
ò	\`o	ō	\=o	ő	\H o	ọ	\d o
ô	\^o	ȯ	\. o	o͡o	\t{oo}	o̲	\b o
ö	\"o	ŏ	\u o	o̊	\r o		

Cuadro 3.1: Acentos.

†	\dag	œ	\oe	ł	\l
‡	\ddag	Œ	\OE	Ł	\L
§	\S	æ	\ae	ß	\ss
¶	\P	Æ	\AE	SS	\SS
©	\copyright	å	\aa	¿	?‘
ⓐ	\textcircled a	Å	\AA	¡	!‘
␣	\textvisiblespace	ø	\o		
£	\pounds	Ø	\O		

Cuadro 3.2: Símbolos especiales y caracteres no ingleses.

texto	\tiny	texto	\normalsize	texto	\LARGE
texto	\scriptsize	texto	\large	texto	\huge
texto	\footnotesize	texto	\Large	texto	\Huge
texto	\small				

Se usan igual que los comandos de cambio de font \rm, \sf, etc., de la sección 3.3.13.

\normalsize es el default para texto normal; \scriptsize para sub o supraíndices; \footnotesize para notas a pie de página.

3.3.14. Acentos y símbolos.

LaTeX provee diversos tipos de acentos, que se muestran en la Tabla 3.1 (como ejemplo consideramos la letra “o”, pero cualquiera es posible, por supuesto). (Hemos usado acá el hecho de que cuando el argumento de un comando consta de un carácter, las llaves son omitibles.)

Otros símbolos especiales y caracteres no ingleses disponibles se encuentran en la Tabla 3.2.

3.3.15. Escritura de textos en castellano.

LaTeX emplea sólo los caracteres ASCII básicos, que no contienen símbolos castellanos como ¿, ¡, ñ, etc. Ya hemos visto que existen comandos que permiten imprimir estos caracteres,

y por tanto es posible escribir cualquier texto en castellano (y otros idiomas, de hecho).

Sin embargo, esto no resuelve todo el problema, porque en inglés y castellano las palabras se cortan en "sílabas" de acuerdo a reglas distintas, y esto es relevante cuando se debe cortar el texto en líneas. LaTeX tiene incorporados algoritmos para cortar palabras en inglés y, si se ha hecho una instalación especial de LaTeX en nuestro computador, también en castellano u otros idiomas (a través del programa **babel**, que es parte de la distribución standard de LaTeX 2_ε). En un computador con babel instalado y configurado para cortar en castellano basta incluir el comando `\usepackage[spanish]{babel}` en el preámbulo para poder escribir en castellano cortando las palabras en sílabas correctamente.[3]

Sin embargo, ocasionalmente LaTeX se encuentra con una palabra que no sabe cortar, en cuyo caso no lo intenta y permite que ella se salga del margen derecho del texto, o bien toma decisiones no óptimas. La solución es sugerirle a LaTeX la silabación de la palabra. Por ejemplo, si la palabra conflictiva es `matem\'aticas` (generalmente hay problemas con las palabras acentuadas), entonces basta con reescribirla en la forma: `ma\-te\-m\'a\-ti\-cas`. Con esto, le indicamos a LaTeX en qué puntos es posible cortar la palabra. El comando `\-` no tiene ningún otro efecto, de modo que si la palabra en cuestión no queda al final de la línea, LaTeX por supuesto ignora nuestra sugerencia y no la corta.

Consideremos el siguiente ejemplo:

Podemos escribir matemáticas. O matemáticas.

```
Podemos escribir matem\'aticas.
O matem\'aticas.
```

Podemos escribir matemáticas. O matemáticas.

```
Podemos escribir
ma\-te\-m\'a\-ti\-cas.
O ma\-te\-m\'a\-ti\-cas.
```

En el primer caso, LaTeX decidió por sí mismo dónde cortar "matemáticas". Como es una palabra acentuada tuvo problemas y no lo hizo muy bien, pues quedó demasiado espacio entre palabras en esa línea. En el segundo párrafo le sugerimos la silabación y LaTeX pudo tomar una decisión más satisfactoria. En el mismo párrafo, la segunda palabra "matemáticas" también tiene sugerencias de corte, pero como no quedó al final de línea no fueron tomadas en cuenta.

3.4. Fórmulas matemáticas.

Hemos mencionado tres formas de ingresar al modo matemático: `$...$` (fórmulas dentro del texto), `$$...$$` (fórmulas separadas del texto, no numeradas) y `\begin{equation} ... \end{equation}` (fórmulas separadas del texto, numeradas). Los comandos que revisaremos en esta sección sólo pueden aparecer dentro del modo matemático.

[3] Esto resuelve también otro problema: los encabezados de capítulos o índices, por ejemplo, son escritos "Capítulo" e "Índice", en vez de "Chapter" e "Index", y cuando se usa el comando `\date`, la fecha aparece en castellano.

3.4.1. Sub y supraíndices.

x^{2y} `x^{2y}` x^{y^2} `x^{y^{2}}` (ó `x^{y^2}`) x_1^y `x^y_1` (ó `x_1^y`)
x_{2y} `x_{2y}` x^{y_1} `x^{y_{1}}` (ó `x^{y_1}`)

`\textsuperscript` permite obtener supraíndices fuera del modo matemático:

La 3ª es la vencida.

```
La 3\textsuperscript{a}
es la vencida.
```

3.4.2. Fracciones.

a) Horizontales

$n/2$ `n/2`

b) Verticales

$\frac{1}{2}$ `\frac{1}{2}`, `\frac 1{2}`, `\frac{1}2` ó `\frac 12`

$x = \frac{y + z/2}{y^2+1}$ `x = \frac{y + z/2}{y^2+1}`

$\frac{x+y}{1 + \frac{y}{z+1}}$ `\frac{x+y}{1 + \frac y{z+1}}`

La forma a) es más adecuada y la preferida para fracciones dentro del texto, y la segunda para fórmulas separadas. `\frac` puede aparecer en fórmulas dentro del texto ($\frac{1}{2}$ con `$\frac 12$`), pero esto es inusual y poco recomendable estéticamente, salvo estricta necesidad.

3.4.3. Raíces.

$\sqrt{n}$ `\sqrt{n}` ó `\sqrt n`

$\sqrt{a^2 + b^2}$ `\sqrt{a^2 + b^2}`

$\sqrt[n]{2}$ `\sqrt[n]{2}`

3.4.4. Puntos suspensivos.

a) ... `\ldots`

Para fórmulas como

$a_1 a_2 \ldots a_n$ `a_1 a_2 \ldots a_n`

b) $\cdots$ `\cdots`

Entre símbolos como $+$, $-$, $=$:

$$x_1 + \cdots + x_n \qquad \texttt{x\_1 + \cdots + x\_n}$$

c) $\vdots$ `\vdots`

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

d) $\ddots$ `\ddots`

$$I_{n\times n} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

`\ldots` puede ser usado también en el texto usual:

Arturo quiso salir...pero se detuvo.	`Arturo quiso salir\ldots pero se detuvo.`

No corresponde usar tres puntos seguidos (...), pues el espaciado entre puntos es incorrecto.

3.4.5. Letras griegas.

Las letras griegas se obtienen simplemente escribiendo el nombre de dicha letra (en inglés): `\gamma`. Para la mayúscula correspondiente se escribe la primera letra con mayúscula: `\Gamma`. La lista completa se encuentra en la Tabla 3.3.[4]

No existen símbolos para α, β, η, etc. mayúsculas, pues corresponden a letras romanas (A, B, E, etc.).

[4]Un ejemplo del uso de variantes de letras griegas, en el idioma griego σ se usa dentro de una palabra y ς se usa al finalizar una palabra. El nombre Felipe en griego, es Felipós, y se escribe de la forma: $\Phi\varepsilon\lambda\eta\pi\acute{o}\varsigma$. El nombre José, sería algo como Josué en griego: *Ιοσυή* o la palabra Física: *Φυσική*

Minúsculas

α	\alpha	θ	\theta	o	o	τ	\tau
β	\beta	ϑ	\vartheta	π	\pi	υ	\upsilon
γ	\gamma	ι	\iota	ϖ	\varpi	ϕ	\phi
δ	\delta	κ	\kappa	ρ	\rho	φ	\varphi
ϵ	\epsilon	λ	\lambda	ϱ	\varrho	χ	\chi
ε	\varepsilon	μ	\mu	σ	\sigma	ψ	\psi
ζ	\zeta	ν	\nu	ς	\varsigma	ω	\omega
η	\eta	ξ	\xi				

Mayúsculas

Γ	\Gamma	Λ	\Lambda	Σ	\Sigma	Ψ	\Psi
Δ	\Delta	Ξ	\Xi	Υ	\Upsilon	Ω	\Omega
Θ	\Theta	Π	\Pi	Φ	\Phi		

Cuadro 3.3: Letras griegas.

3.4.6. Letras caligráficas.

Letras caligráficas mayúsculas $\mathcal{A}$, $\mathcal{B}$, ..., $\mathcal{Z}$ se obtienen con `\cal`. `\cal` se usa igual que los otros comandos de cambio de font (`\rm`, `\it`, etc.).

Sea $\mathcal{F}$ una función con $\mathcal{F}(x) > 0$.	`Sea $\cal F$ una funci\'on con ${\cal F}(x) > 0$.`

No son necesarios los paréntesis cursivos la primera vez que se usan en este ejemplo, porque el efecto de `\cal` está delimitado por los `$`.

3.4.7. Símbolos matemáticos.

LaTeX proporciona una gran variedad de símbolos matemáticos (Tablas 3.4, 3.5, 3.6, 3.7). La negación de cualquier símbolo matemático se obtiene con `\not`:

$x \not< y$	`x \not < y`
$a \notin \mathcal{M}$	`a \not \in {\cal M}`

[Notemos, sí, en la Tabla 3.5, que existe el símbolo $\neq$ (`\neq`).]

Algunos símbolos tienen tamaño variable, según aparezcan en el texto o en fórmulas separadas del texto. Se muestran en la Tabla 3.8.

Estos símbolos pueden tener índices que se escriben como sub o supraíndices. Nuevamente, la ubicación de estos índices depende de si la fórmula está dentro del texto o separada de él:

$\pm$	\pm	$\cap$	\cap	$\diamond$	\diamond	$\oplus$	\oplus
$\mp$	\mp	$\cup$	\cup	$\triangle$	\bigtriangleup	$\ominus$	\ominus
$\times$	\times	$\uplus$	\uplus	$\triangledown$	\bigtriangledown	$\otimes$	\otimes
$\div$	\div	$\sqcap$	\sqcap	$\triangleleft$	\triangleleft	$\oslash$	\oslash
$*$	\ast	$\sqcup$	\sqcup	$\triangleright$	\triangleright	$\odot$	\odot
$\star$	\star	$\vee$	\lor	$\bigcirc$	\bigcirc		
$\circ$	\circ	$\wedge$	\land	$\dagger$	\dagger		
$\bullet$	\bullet	$\setminus$	\setminus	$\ddagger$	\ddagger		
$\cdot$	\cdot	$\wr$	\wr	$\amalg$	\amalg		

Cuadro 3.4: Símbolos de operaciones binarias.

$\leq$	\leq	$\geq$	\geq	$\equiv$	\equiv	$\models$	\models
$\prec$	\prec	$\succ$	\succ	$\sim$	\sim	$\perp$	\perp
$\preceq$	\preceq	$\succeq$	\succeq	$\simeq$	\simeq	$\mid$	\mid
$\ll$	\ll	$\gg$	\gg	$\asymp$	\asymp	$\parallel$	\parallel
$\subset$	\subset	$\supset$	\supset	$\approx$	\approx	$\bowtie$	\bowtie
$\subseteq$	\subseteq	$\supseteq$	\supseteq	$\cong$	\cong	$\neq$	\neq
$\smile$	\smile	$\sqsubseteq$	\sqsubseteq	$\sqsupseteq$	\sqsupseteq	$\doteq$	\doteq
$\frown$	\frown	$\in$	\in	$\ni$	\ni	$\propto$	\propto
$\vdash$	\vdash	$\dashv$	\dashv				

Cuadro 3.5: Símbolos relacionales.

$\leftarrow$	\gets	$\longleftarrow$	\longleftarrow	$\uparrow$	\uparrow
$\Leftarrow$	\Leftarrow	$\Longleftarrow$	\Longleftarrow	$\Uparrow$	\Uparrow
$\rightarrow$	\to	$\longrightarrow$	\longrightarrow	$\downarrow$	\downarrow
$\Rightarrow$	\Rightarrow	$\Longrightarrow$	\Longrightarrow	$\Downarrow$	\Downarrow
$\Leftrightarrow$	\Leftrightarrow	$\Longleftrightarrow$	\Longleftrightarrow	$\Updownarrow$	\Updownarrow
$\mapsto$	\mapsto	$\longmapsto$	\longmapsto	$\nearrow$	\nearrow
$\hookleftarrow$	\hookleftarrow	$\hookrightarrow$	\hookrightarrow	$\searrow$	\searrow
$\leftharpoonup$	\leftharpoonup	$\rightharpoonup$	\rightharpoonup	$\swarrow$	\swarrow
$\leftharpoondown$	\leftharpoondown	$\rightharpoondown$	\rightharpoondown	$\nwarrow$	\nwarrow
$\rightleftharpoons$	\rightleftharpoons				

Cuadro 3.6: Flechas

$\aleph$	\aleph	$\prime$	\prime	$\forall$	\forall	∞	\infty
$\hbar$	\hbar	$\emptyset$	\emptyset	$\exists$	\exists	$\triangle$	\triangle
$\imath$	\imath	∇	\nabla	$\neg$	\lnot	$\clubsuit$	\clubsuit
$\jmath$	\jmath	$\surd$	\surd	$\flat$	\flat	$\diamondsuit$	\diamondsuit
ℓ	\ell	$\top$	\top	$\natural$	\natural	$\heartsuit$	\heartsuit
$\wp$	\wp	$\bot$	\bot	$\sharp$	\sharp	$\spadesuit$	\spadesuit
$\Re$	\Re	$\|$	|	$\backslash$	\backslash		
$\Im$	\Im	$\angle$	\angle	∂	\partial		

Cuadro 3.7: Símbolos varios.

Σ	$\sum$	\sum	$\cap$	$\bigcap$	\bigcap	$\sqcup$	$\bigsqcup$	\bigsqcup
Π	$\prod$	\prod	$\cup$	$\bigcup$	\bigcup	$\odot$	$\bigodot$	\bigodot
$\amalg$	$\coprod$	\coprod	$\uplus$	$\biguplus$	\biguplus	$\otimes$	$\bigotimes$	\bigotimes
$\int$	$\int$	\int	$\vee$	$\bigvee$	\bigvee	$\oplus$	$\bigoplus$	\bigoplus
$\oint$	$\oint$	\oint	$\wedge$	$\bigwedge$	\bigwedge			

Cuadro 3.8: Símbolos de tamaño variable.

```
\arccos  \cos   \csc  \exp  \ker     \limsup  \min  \sinh
\arcsin  \cosh  \deg  \gcd  \lg      \ln       \Pr   \sup
\arctan  \cot   \det  \hom  \lim     \log      \sec  \tan
\arg     \coth  \dim  \inf  \liminf  \max      \sin  \tanh
```

Cuadro 3.9: Funciones tipo logaritmo

$$\sum_{i=1}^{n} x_i = \int_0^1 f$$ `$$\sum_{i=1}^n x_i = \int_0^1 f $$`

$\sum_{i=1}^{n} x_i = \int_0^1 f$ `$\sum_{i=1}^n x_i = \int_0^1 f $`

3.4.8. Funciones tipo logaritmo.

Observemos la diferencia entre estas dos expresiones:

$x = logy$ `$x  = log y$`

$x = \log y$ `$x  = \log y$`

En el primer caso LaTeX escribe el producto de cuatro cantidades, l, o, g e y. En el segundo, representa correctamente nuestro deseo: el logaritmo de y. Todos los comandos de la Tabla 3.9 generan el nombre de la función correspondiente, en letras romanas.

Algunas de estas funciones pueden tener índices:

$$\lim_{n\to\infty} x_n = 0$$ `$$\lim_{n\to\infty} x_n = 0 $$`

$\lim_{n\to\infty} x_n = 0$ `$\lim_{n\to\infty} x_n = 0 $`

3.4.9. Matrices.

Ambiente `array`.

Se construyen con el ambiente `array`. Consideremos, por ejemplo:

$$\begin{array}{ccc} a+b+c & uv & 27 \\ a+b & u+v & 134 \\ a & 3u+vw & 2.978 \end{array}$$

La primera columna está alineada al centro (`c`, center); la segunda, a la izquierda (`l`, left); la tercera, a la derecha (`r`, right). `array` tiene un argumento obligatorio, que consta de tantas letras como columnas tenga la matriz, letras que pueden ser `c`, `l` o `r` según la alineación que queramos obtener. Elementos consecutivos de la misma línea se separan con `&` y líneas consecutivas se separan con `\\`. Así, el ejemplo anterior se obtiene con:

(	(	)	)	↑	\uparrow
[	[	]	]	↓	\downarrow
{	\{	}	\}	↕	\updownarrow
⌊	\lfloor	⌋	\rfloor	⇑	\Uparrow
⌈	\lceil	⌉	\rceil	⇓	\Downarrow
⟨	\langle	⟩	\rangle	⇕	\Updownarrow
/	/	\	\backslash		
|	|	‖	\\|		

Cuadro 3.10: Delimitadores

```
\begin{array}{clr}
a+b+c & uv & 27 \\
a+b & u + v & 134 \\
a & 3u+vw & 2.978
\end{array}
```

Delimitadores.

Un delimitador es cualquier símbolo que actúe como un paréntesis, encerrando una expresión, apareciendo a la izquierda y a la derecha de ella. La Tabla 3.10 muestra todos los delimitadores posibles.

Para que los delimitadores tengan el tamaño correcto para encerrar la expresión correspondiente hay que anteponerles `\left` y `\right`. Podemos obtener así expresiones matriciales:

$$\left(\begin{array}{cc} a & b \\ c & d \end{array} \right)$$

```
\left(\begin{array}{cc}
a&b\\
c&d
\end{array}\right)
```

$$v = \left(\begin{array}{c} 1 \\ 2 \\ 3 \end{array} \right)$$

```
v = \left(\begin{array}{c}
1\\
2\\
3
\end{array}\right)
```

$$\Delta = \left| \begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right|$$

```
\Delta = \left|\begin{array}{cc}
a_{11} & a_{12}\\
a_{21} & a_{22}
\end{array}\right|
```

`\left` y `\right` deben ir de a pares, pero los delimitadores no tienen por qué ser los mismos:

$$\left(\begin{array}{c} a \\ b \end{array} \right[$$

```
\left(\begin{array}{c}
a\\
b
\end{array}\right[
```

Tampoco es necesario que los delimitadores encierren matrices. Comparemos, por ejemplo:

$$(\vec{A} + \vec{B}) = (\frac{d\vec{F}}{dx})_{x=a}$$

```
(\vec A + \vec B) =
( \frac{d \vec F}{dx} )_{x=a}
```

$$\left(\vec{A} + \vec{B}\right) = \left(\frac{d\vec{F}}{dx}\right)_{x=a}$$

```
\left(\vec A + \vec B\right) =
\left( \frac{d \vec F}{dx} \right)_{x=a}
```

El segundo ejemplo es mucho más adecuado estéticamente.

Algunas expresiones requieren sólo un delimitador, a la izquierda o a la derecha. Un punto (.) representa un delimitador invisible. Los siguientes ejemplos son típicos:

$$\int_a^b dx \frac{df}{dx} = f(x)\Big|_a^b$$

```
\left. \int_a^b  dx \frac{df}{dx} =
f(x) \right |_a^b
```

$$f(x) = \left\{ \begin{array}{cl} 0 & x < 0 \\ 1 & x > 0 \end{array} \right.$$

```
f(x) = \left\{ \begin{array}{cl}
0 & x<0 \\
1 & x>0
\end{array} \right.
```

Fórmulas de más de una línea.

`eqnarray` ofrece una manera de ingresar a modo matemático (en reemplazo de `$`, `$$` o `equation`) equivalente a un `array` con argumentos `{rcl}`:

$$\begin{aligned} x &= a + b + c + \\ &\quad d + e \end{aligned}$$

```
\begin{eqnarray*}
x& = & a + b + c +\\
&& d + e
\end{eqnarray*}
```

El asterisco impide que aparezcan números en las ecuaciones. Si deseamos que numere cada línea como una ecuación independiente, basta omitir el asterisco:

$$\begin{aligned} x &= 5 \quad (3.2) \\ a + b &= 60 \quad (3.3) \end{aligned}$$

```
\begin{eqnarray}
x& = & 5 \\
a + b&= & 60
\end{eqnarray}
```

$\hat{a}$	\hat a	$\acute{a}$	\acute a	$\bar{a}$	\bar a	$\dot{a}$	\dot a
$\check{a}$	\check a	$\grave{a}$	\grave a	$\vec{a}$	\vec a	$\ddot{a}$	\ddot a
$\breve{a}$	\breve a	$\tilde{a}$	\tilde a				

Cuadro 3.11: Acentos matemáticos

Si queremos que solamente algunas líneas aparezcan numeradas, usamos `\nonumber`:

$$\begin{aligned} x &= a+b+c+ \\ &\quad d+e \end{aligned} \tag{3.4}$$

```
\begin{eqnarray}
x& = & a + b + c + \nonumber\\
&& d + e
\end{eqnarray}
```

El comando `\eqnarray` es suficiente para necesidades sencillas, pero cuando se requiere escribir matemática de modo intensivo sus limitaciones comienzan a ser evidentes. Al agregar al preámbulo de nuestro documento la línea `\usepackage{amsmath}` quedan disponibles muchos comandos mucho más útiles para textos matemáticos más serios, como el ambiente `equation*`, `\split`, `\multline` o `\intertext`. En la sección 3.8.2 se encuentra una descripción de estos y otros comandos.

3.4.10. Acentos.

Dentro de una fórmula pueden aparecer una serie de "acentos", análogos a los de texto usual (Tabla 3.11).

Las letras i y j deben perder el punto cuando son acentuadas: $\vec{i}$ es incorrecto. Debe ser $\vec{\imath}$. `\imath` y `\jmath` generan las versiones sin punto de estas letras:

$\vec{\imath} + \hat{\jmath}$

```
\vec \imath + \hat \jmath
```

3.4.11. Texto en modo matemático.

Para insertar texto dentro de modo matemático empleamos `\mbox`:

$V_{\mbox{\scriptsize cr\'{\i}tico}}$

```
V_{\mbox{\scriptsize cr\'{\i}tico}}
```

Bastante más óptimo es utilizar el comando `\text`, disponible a través de `amsmath` (sección 3.8.2).

3.4.12. Espaciado en modo matemático.

TEX ignora los espacios que uno escribe en las fórmulas y los determina de acuerdo a sus propios criterios. A veces es necesario ayudarlo para hacer ajustes finos. Hay cuatro comandos que agregan

pequeños espacios dentro de modo matemático:

`\,`	espacio pequeño	`\:`	espacio medio
`\!`	espacio pequeño (negativo)	`\;`	espacio grueso

Algunos ejemplos de su uso:

$\sqrt{2}\,x$	`\sqrt 2 \, x`	en vez de	$\sqrt{2}x$
$n/\log n$	`n / \!\log n`	en vez de	$n/\log n$
$\int f\,dx$	`\int f \, dx`	en vez de	$\int f dx$

El último caso es quizás el más frecuente, por cuanto la no inserción del pequeño espacio adicional entre f y dx hace aparecer el integrando como el producto de tres variables, f, d y x, que no es la idea.

3.4.13. Fonts.

Análogamente a los comandos para texto usual (Sec. 3.3.13), es posible cambiar los fonts dentro del modo matemático:

(A, x)	`\mathrm{(A,x)}`
(A, x)	`\mathnormal{(A,x)}`
$(\mathcal{A}, \mathcal{B})$	`\mathcal{(A,B)}`
$(\mathbf{A}, \mathbf{x})$	`\mathbf{(A,x)}`
(A, x)	`\mathsf{(A,x)}`
$(\mathtt{A}, \mathtt{x})$	`\mathtt{(A,x)}`
(A, x)	`\mathit{(A,x)}`

(Recordemos que la letras tipo `\cal` sólo existen en mayúsculas.)

Las declaraciones anteriores permiten cambiar los fonts de letras, dígitos y acentos, pero no de los otros símbolos matemáticos:

$\mathbf{\tilde A \times 1}$	`\mathbf{\tilde A \times 1}`

Como en todo ambiente matemático, los espacios entre caracteres son ignorados:

Hola	`\mathrm{H o l a}`

Finalmente, observemos que `\mathit` corresponde al font itálico, en tanto que `\mathnormal` al font matemático usual, que es también itálico... o casi:

$different$	`$different$`
$\mathnormal{different}$	`$\mathnormal{different}$`
$\mathit{different}$	`$\mathit{different}$`
different	`\textit{different}`

3.5. Tablas.

`array` nos permitió construir matrices en modo matemático. Para tablas de texto existe `tabular`, que funciona de la misma manera. Puede ser usado tanto en modo matemático como fuera de él.

Nombre	:	Juan Pérez
Edad	:	26
Profesión	:	Estudiante

```
\begin{tabular}{lcl}
Nombre&:&Juan P\'erez\\
Edad&:&26\\
Profesi\'on&:&Estudiante
\end{tabular}
```

Si deseamos agregar líneas verticales y horizontales para ayudar a la lectura, lo hacemos insertando `|` en los puntos apropiados del argumento de `tabular`, y `\hline` al final de cada línea de la tabla:

Item	Gastos
Vasos	$ 500
Botellas	$ 1300
Platos	$ 500
Total	$ 2300

```
\begin{tabular}{|l|r|}\hline
Item&Gastos\\ \hline
Vasos& \$ 500 \\
Botellas & \$ 1300 \\
Platos & \$ 500 \\ \hline
Total& \$ 2300 \\ \hline
\end{tabular}
```

3.6. Referencias cruzadas.

Ecuaciones, secciones, capítulos y páginas son entidades que van numeradas y a las cuales podemos querer referirnos en el texto. Evidentemente no es óptimo escribir explícitamente el número correspondiente, pues la inserción de una nueva ecuación, capítulo, etc., su eliminación o cambio de orden del texto podría alterar la numeración, obligándonos a modificar estos números dispersos en el texto. Mucho mejor es referirse a ellos de modo simbólico y dejar que TEX inserte por nosotros los números. Lo hacemos con `\label` y `\ref`.

La ecuación de Euler

$$e^{i\pi} + 1 = 0 \qquad (3.5)$$

reúne los números más importantes. La ecuación (3.5) es famosa.

```
La ecuaci\'on de Euler
\begin{equation}
\label{euler}
 e^{i\pi} + 1 = 0
\end{equation}
re\'une los n\'umeros
m\'as importantes.
La ecuaci\'on (\ref{euler})
es famosa.
```

El argumento de `\label` (reiterado luego en `\ref`) es una etiqueta simbólica. Ella puede ser cualquier secuencia de letras, dígitos o signos de puntuación. Letras mayúsculas y minúsculas son diferentes. Así, `euler`, `eq:euler`, `euler_1`, `euler1`, `Euler`, etc., son etiquetas válidas y distintas. Podemos usar `\label` dentro de `equation`, `eqnarray` y `enumerate`.

También podemos referenciar páginas con `\pageref`:

Ver página 82 para más detalles.

[Texto en pág. 82]

El significado de la vida...

```
Ver p\'agina
\pageref{significado}
para m\'as detalles.
...
El significado
\label{significado}
de la vida...
```

LaTeX puede dar cuenta de las referencias cruzadas gracias al archivo `aux` (auxiliar) generado durante la compilación.

Al compilar por primera vez el archivo, en el archivo `aux` es escrita la información de los `\label` encontrados. Al compilar por segunda vez, LaTeX lee el archivo `aux` e incorpora esa información al `dvi`. (En realidad, también lo hizo la primera vez que se compiló el archivo, pero el `aux` no existía entonces o no tenía información útil.)

Por tanto, para obtener las referencias correctas hay que compilar dos veces, una para generar el `aux` correcto, otra para poner la información en el `dvi`. Toda modificación en la numeración tendrá efecto sólo después de compilar dos veces más. Por cierto, no es necesario preocuparse de estos detalles a cada momento. Seguramente compilaremos muchas veces el archivo antes de tener la versión final. En todo caso, LaTeX avisa, tras cada compilación, si hay referencias inexistentes u otras que pudieron haber cambiado, y sugiere compilar de nuevo para obtener las referencias correctas. (Ver Sec. 3.14.2.)

3.7. Texto centrado o alineado a un costado.

Los ambientes `center`, `flushleft` y `flushright` permiten forzar la ubicación del texto respecto a los márgenes. Líneas consecutivas se separan con `\\`:

Una línea centrada,
otra
y otra más.

Ahora el texto continúa

alineado a la izquierda

y finalmente

dos líneas
alineadas a la derecha.

```
\begin{center}
Una l\'{\i}nea centrada,\\
otra\\
y otra m\'as.
\end{center}
Ahora el texto contin\'ua
\begin{flushleft}
alineado a la izquierda
\end{flushleft}
y finalmente
\begin{flushright}
dos l\'{\i}neas\\
alineadas a la derecha.
\end{flushright}
```

3.8. Algunas herramientas importantes

Hasta ahora hemos mencionado escencialmente comandos disponibles en LaTeX standard. Sin embargo, éstos, junto con el resto de los comandos básicos de LaTeX, se vuelven insuficientes cuando se trata de ciertas aplicaciones demasiado específicas, pero no inimaginables: si queremos escribir un texto de alta matemática, o usar LaTeX para escribir partituras, o para escribir un archivo `.tex` en un teclado croata.... Es posible que con los comandos usuales LaTeX responda a las necesidades, pero seguramente ello será a un costo grande de esfuerzo por parte del autor del texto. Por esta razón, las distribuciones modernas de LaTeX incorporan una serie de extensiones que hacen la vida un poco más fácil a los eventuales autores. En esta sección mencionaremos algunas extensiones muy útiles. Muchas otras no están cubiertas, y se sugiere al lector consultar la documentación de su distribución para saber qué otros paquetes se encuentran disponibles.

En general, las extensiones a LaTeX vienen contenidas en *paquetes* (*"packages"*, en inglés), en archivos `.sty`. Así, cuando mencionemos el paquete `amsmath`, nos referimos a características disponibles en el archivo `amsmath.sty`. Para que los comandos de un paquete `<package>.sty` estén disponibles, deben ser cargados durante la compilación, incluyendo en el preámbulo del documento la línea:

```
\usepackage{<package>}
```

Si se requiere cargar más de un paquete adicional, se puede hacer de dos formas:

```
\usepackage{<package1>,<package2>}
```

o

```
\usepackage{<package1>}
\usepackage{<package2>}
```

Algunos paquetes aceptan opciones adicionales (del mismo modo que la clase `article` acepta la opción `12pt`):

```
\usepackage[option1,option2]{<package1>}
```

Revisemos ahora algunos paquetes útiles.

3.8.1. `babel`

Permite el procesamiento de textos en idiomas distintos del inglés. Esto significa, entre otras cosas, que se incorporan los patrones de silabación correctos para dicho idioma, para cortar adecuadamente las palabras al final de cada línea. Además, palabras claves como "Chapter", "Index", "List of Figures", etc., y la fecha dada por `\date`, son cambiadas a sus equivalentes en el idioma escogido. La variedad de idiomas disponibles es enorme, pero cada instalación de LaTeX tiene sólo algunos de ellos incorporados. (Ésta es una decisión que toma el administrador del sistema, de acuerdo a las necesidades de los usuarios. Una configuración usual puede ser habilitar la compilación en inglés, castellano, alemán y francés.)

Ya sabemos como usar `babel` para escribir en castellano: basta incluir en el preámbulo la línea

```
\usepackage[spanish]{babel}
```

3.8.2. $\mathcal{AMS}$-LaTeX

El paquete `amsmath` permite agregar comandos para escritura de textos matemáticos profesionales, desarrollados originalmente por la American Mathematical Society. Si un texto contiene abundante matemática, entonces seguramente incluir la línea correspondiente en el preámbulo:

```
\usepackage{amsmath}
```

aliviará mucho la tarea. He aquí algunas de las características adicionales disponibles con $\mathcal{AMS}$-LaTeX.

Ambientes para ecuaciones

Con `equation*` generamos una ecuación separada del texto, no numerada:

$$x = 2y - 3$$

```
\begin{equation*}
x = 2y - 3
\end{equation*}
```

`multline` permite dividir una ecuación muy larga en varias líneas, de modo que la primera línea quede alineada con el margen izquierdo, y la última con el margen derecho:

$$\sum_{i=1}^{15} = 1 + 2 + 3 + 4 + 5+ \\ 6 + 7 + 8 + 9 + 10+ \\ 11 + 12 + 13 + 14 + 15 \quad (3.6)$$

```
\begin{multline}
\sum_{i=1}^{15} = 1 +2+3+4+5+\\
6+7+8+9+10+\\
11+12+13+14+15
\end{multline}
```

`align` permite reunir un grupo de ecuaciones consecutivas alineándolas (usando `&`, igual que la alineación vertical de `tabular` y `array`). `gather` hace lo mismo, pero centrando cada ecuación en la página independientemente.

$$a_1 = b_1 + c_1 \quad (3.7)$$
$$a_2 = b_2 + c_2 - d_2 + e_2 \quad (3.8)$$

```
\begin{align}
a_1 &= b_1 + c_1 \\
a_2 &= b_2 + c_2 - d_2 + e_2
\end{align}
```

$$a_1 = b_1 + c_1 \quad (3.9)$$
$$a_2 = b_2 + c_2 - d_2 + e_2 \quad (3.10)$$

```
\begin{gather}
a_1 = b_1 + c_1 \\
a_2 = b_2 + c_2 - d_2 + e_2
\end{gather}
```

Con `multline*`, `align*` y `gather*` se obtienen los mismos resultados, pero con ecuaciones no numeradas.

`split` permite escribir una sola ecuación separada en líneas (como `multline`), pero permite alinear las líneas con `&` (como `align`). `split` debe ser usado dentro de un ambiente como `equation`, `align` o `gather` (o sus equivalentes con asterisco):

$$\begin{aligned} a_1 &= b_1 + c_1 \\ &= b_2 + c_2 - d_2 + e_2 \end{aligned} \tag{3.11}$$

```
\begin{equation}
\begin{split}
a_1& = b_1 + c_1 \\
   & =  b_2 + c_2 - d_2 + e_2
\end{split}
\end{equation}
```

Espacio horizontal

`\quad` y `\qquad` insertan espacio horizontal en ecuaciones:

$$x > y \ , \quad \forall x \in A$$
$$x \leq z \ , \qquad \forall z \in B$$

```
\begin{gather*}
x > y \ , \quad \forall\, x \in A \\
x \leq z \ , \qquad \forall\, z \in B
\end{gather*}
```

Texto en ecuaciones

Para agregar texto a una ecuación, usamos `\text`:

$$x = 2^n - 1 \ , \quad \text{con } n \text{ entero}$$

```
\begin{equation*}
x = 2^n - 1 \ , \quad \text{con $n$ entero}
\end{equation*}
```

`\text` se comporta como un buen objeto matemático, y por tanto se pueden agregar subíndices textuales más fácilmente que con `\mbox` (ver sección 3.4.11):

$V_{\text{crítico}}$

```
$V_{\text{cr\'{\i}tico}}$
```

Referencia a ecuaciones

`\eqref` es equivalente a `\ref`, salvo que agrega los paréntesis automáticamente:

La ecuación (3.5) era la de Euler.

```
La ecuaci\'on \eqref{euler} era la de Euler.
```

Ecuaciones con casos

Ésta es una construcción usual en matemáticas:

$$f(x) = \begin{cases} 1 & \text{si } x < 0 \\ 0 & \text{si } x > 0 \end{cases}$$

```
f(x)=
\begin{cases}
1&\text{si $x<0$} \\
0&\text{si $x>0$}
\end{cases}
```

Notar cómo es más simple que el ejemplo con los comandos convencionales en la sección 3.4.9.

Texto insertado entre ecuaciones alineadas

Otra situación usual es insertar texto entre ecuaciones alineadas, preservando la alineación:

$$\begin{aligned} x_1 &= a + b + c \ , \\ x_2 &= d + e \ , \end{aligned}$$

y por otra parte

$$x_3 = f + g + h \ .$$

```
\begin{align*}
x_1 &=  a + b + c  \ , \\
x_2 &=  d + e \ , \\
\intertext{y por otra parte}
x_3 &=  f + g + h \ .
\end{align*}
```

Matrices y coeficientes binomiales

La complicada construcción de matrices usando `array` (sección 3.4.9), se puede reemplazar con ambientes como `pmatrix` y `vmatrix`, y comandos como `\binom`.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

```
\begin{pmatrix}
a&b\\
c&d
\end{pmatrix}
```

$$\Delta = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

```
\Delta = \begin{vmatrix}
a_{11} & a_{12}\\
a_{21} & a_{22}
\end{vmatrix}
```

$$v = \binom{k}{2}$$

```
v = \binom{k}{2}
```

Podemos observar que el espaciado entre los paréntesis y el resto de la fórmula es más adecuado que el de los ejemplos en la sección 3.4.9.

Flechas extensibles

Las flechas en la tabla 3.6 vienen en ciertos tamaños predefinidos. `amsmath` proporciona flechas extensibles `\xleftarrow` y `\xrightarrow`, para ajustar sub o superíndices demasiado anchos. Además, tienen un argumento opcional y uno obligatorio, para colocar material sobre o bajo ellas:

$$A \xleftarrow{n+\mu-1} B \xrightarrow[T]{n\pm i-1} C \underset{U}{\rightarrow} D$$

```
A \xleftarrow{n+\mu-1} B \xrightarrow[T]{n\pm i-1} C \underset{U}{\rightarrow} D
```

Uso del paquete `amssymb`

Este paquete contiene símbolos matemáticos muy estéticos a la hora de referirse a los conjuntos de números:

$\mathbb{C}$	`\mathbb{C}`	para los números complejos	$\mathbb{I}$	`\mathbb{I}`	para los números imaginarios
$\mathbb{R}$	`\mathbb{R}`	para los números reales	$\mathbb{Q}$	`\mathbb{Q}`	para los números racionales
$\mathbb{Z}$	`\mathbb{Z}`	para los números enteros	$\mathbb{N}$	`\mathbb{N}`	para los números naturales

Otro símbolo interesante es: $\therefore$ (`\therefore`) para "por lo tanto".

3.8.3. `fontenc`

Ocasionalmente, LaTeX tiene problemas al separar una palabra en sílabas. Típicamente, eso ocurre con palabras acentuadas, pues, debido a la estructura interna del programa, un carácter como la "á" en "matemáticas" no es tratado igual que los otros. Para solucionar el problema, y poder cortar en sílabas palabras que contengan letras acentuadas (además de acceder a algunos caracteres adicionales), basta incluir el paquete `fontenc`:

```
\usepackage[T1]{fontenc}
```

Técnicamente, lo que ocurre es que la codificación antigua para fonts es la `OT1`, que no contiene fonts acentuados, y que por lo tanto es útil sólo para textos en inglés. La codificación `T1` aumenta los fonts disponibles, permitiendo que los caracteres acentuados sean tratados en igual pie que cualquier otro.

3.8.4. `enumerate`

`enumerate.sty` define una muy conveniente extensión al ambiente `enumerate` de LaTeX. El comando se usa igual que siempre (ver sección 3.3.12), con un argumento opcional que determina el tipo de etiqueta que se usará para la lista. Por ejemplo, si queremos que en vez de números se usen letras mayúsculas, basta usar `\begin{enumerate}[A]`:

A Primer ítem.

B Segundo ítem.

Si queremos etiquetas de la forma "1.-", `\begin{enumerate}[1.-]`:

1.- Primer ítem.

2.- Segundo ítem.

Si deseamos insertar un texto que no cambie de una etiqueta a otra, hay que encerrarlo entre paréntesis cursivos (`\begin{enumerate}[{Caso} A:]`):

Caso A: Primer ítem.

Caso B: Segundo ítem.

3.8.5. Color.

A través de PostScript es posible introducir color en documentos LaTeX. Para ello, incluimos en el preámbulo el paquete `color.sty`:

```
\usepackage{color}
```

De este modo, está disponible el comando `\color`, que permite especificar un color, ya sea por nombre (en el caso de algunos colores predefinidos), por su código `rgb` (red-green-blue) o código `cmyk` (cian-magenta-yellow-black). Por ejemplo:

Un texto en azul
Un texto en un segundo color
Un texto en un tercer color

```
Un texto en {\color{blue} azul}

Un texto en un
{\color[rgb]{1,0,1} segundo color}

Un texto en un
{\color[cmyk]{.3,.5,.75,0} tercer color}
```

Los colores más frecuentes (azul, amarillo, rojo, etc.) se pueden dar por nombre, como en este ejemplo. Si se da el código `rgb`, se deben especificar tres números entre 0 y 1, que indican la cantidad de rojo, verde y azul que constituyen el color deseado. En el ejemplo, le dimos máxima cantidad de rojo y azul, y nada de verde, con lo cual conseguimos un color violeta. Si se trata del código `cmyk` los números a especificar son cuatro, indicando la cantidad de cian, magenta, amarillo y negro. En el ejemplo anterior pusimos una cantidad arbitraria de cada color, y resultó un color café. Es evidente que el uso de los códigos `rgb` y `cmyk` permite explorar infinidad de colores.

Observar que `\color` funciona de modo análogo a los comandos de cambio de font de la sección 3.3.13, de modo que si se desea restringir el efecto a una porción del texto, hay que encerrar dicho texto entre paréntesis cursivos. Análogamente al caso de los fonts, existe el comando `\textcolor`, que permite dar el texto a colorear como argumento:

Un texto en azul
Un texto en un segundo color
Un texto en un tercer color

```
Un texto en \textcolor{blue}{azul}

Un texto en un
\textcolor[rgb]{1,0,1}{segundo color}

Un texto en un
\textcolor[cmyk]{.3,.5,.75,0}{tercer color}
```

3.9. Modificando el estilo de la página.

TEX toma una serie de decisiones por nosotros. Ocasionalmente nos puede interesar alterar el comportamiento normal. Disponemos de una serie de comandos para ello, los cuales revisaremos a continuación. Todos deben aparecer en el preámbulo, salvo en los casos que se indique.

3.9.1. Estilos de página.

a) Números de página.

Si se desea que los números de página sean arábicos (1, 2, 3...):

```
\pagenumbering{arabic}
```

Para números romanos (i, ii, iii,...):

```
\pagenumbering{roman}
```

`arabic` es el default.

b) Estilo de página.

El comando `\pagestyle` determina dónde queremos que vayan los números de página:

`\pagestyle{plain}`	Números de página en el extremo inferior, al centro de la página. (Default para estilos `article`, `report`.)
`\pagestyle{headings}`	Números de página y otra información (título de sección, etc.) en la parte superior de la página. (Default para estilo `book`.)
`\pagestyle{empty}`	Sin números de página.

3.9.2. Corte de páginas y líneas.

TEX tiene modos internos de decidir cuándo cortar una página o una línea. Al preparar la versión final de nuestro documento, podemos desear coartar sus decisiones. En todo caso, no hay que hacer esto antes de preparar la versión verdaderamente final, porque agregar, modificar o quitar texto puede alterar los puntos de corte de líneas y páginas, y los cortes inconvenientes pueden resolverse solos.

Los comandos de esta sección no van en el preámbulo, sino en el interior del texto.

Corte de líneas.

En la página 70 ya vimos un ejemplo de inducción de un corte de línea en un punto deseado del texto, al dividir una palabra en sílabas.

Cuando el problema no tiene relación con sílabas disponemos de dos comandos:

`\newline`	Corta la línea y pasa a la siguiente en el punto indicado.
`\linebreak`	Lo mismo, pero justificando la línea para adecuarla a los márgenes.

Un corte de línea
no justificado a los márgenes
en curso.

```
Un corte de l\'{\i}nea\newline
no justificado a los m\'argenes
en curso.
```

Un corte de línea
justificado a los márgenes
en curso.

```
Un corte de l\'{\i}nea\linebreak
justificado a los m\'argenes
en curso.
```

Observemos cómo en el segundo caso, en que se usa `\linebreak`, la separación entre palabras es alterada para permitir que el texto respete los márgenes establecidos.

Corte de páginas.

Como para cortar líneas, existe un modo violento y uno sutil:

`\newpage`	Cambia de página en el punto indicado. Análogo a `\newline`.
`\clearpage`	Lo mismo, pero ajustando los espacios verticales en el texto para llenar del mejor modo posible la página.

`\clearpage`, sin embargo, no siempre tiene efectos visibles. Dependiendo de la cantidad y tipo de texto que quede en la página, los espacios verticales pueden o no ser ajustados, y si no lo son, el resultado termina siendo equivalente a un `\newpage`. TEX decide en última instancia qué es lo óptimo.

Adicionalmente, tenemos el comando:

`\enlargethispage{<longitud>}`	Cambia el tamaño de la página actual en la cantidad `<longitud>`.

(Las unidades de longitud que maneja TEX se revisan a continuación.)

Unidades de longitud y espacios.

a) Unidades.

TEX reconoce las siguientes unidades de longitud:

`cm`	centímetro
`mm`	milímetro
`in`	pulgada
`pt`	punto (1/72 pulgadas)
`em`	ancho de una "M" en el font actual
`ex`	altura de una "x" en el font actual

Las cuatro primeras unidades son absolutas; las últimas dos, relativas, dependiendo del tamaño del font actualmente en uso.

Las longitudes pueden ser números enteros o decimales, positivos o negativos:

```
1cm     1.6in     .58pt     -3ex
```

b) Cambio de longitudes.

TEX almacena los valores de las longitudes relevantes al texto en comandos especiales:

`\parindent`	Sangría.
`\textwidth`	Ancho del texto.
`\textheight`	Altura del texto.
`\oddsidemargin`	Margen izquierdo menos 1 pulgada.
`\topmargin`	Margen superior menos 1 pulgada.
`\baselineskip`	Distancia entre la base de dos líneas de texto consecutivas.
`\parskip`	Distancia entre párrafos.

Todas estas variables son modificables con los comandos `\setlength`, que le da a una variable un valor dado, y `\addtolength`, que le suma a una variable la longitud especificada. Por ejemplo:

`\setlength{\parindent}{0.3em}` (`\parindent` = 0.3 em.)

`\addtolength{\parskip}{1.5cm}` (`\parskip` = `\parskip` + 1.5 cm.)

Por default, el ancho y altura del texto, y los márgenes izquierdo y superior, están definidos de modo que quede un espacio de una pulgada ($\simeq 2.56$ cm) entre el borde del texto y el borde de la página.

Un problema típico es querer que el texto llene un mayor porcentaje de la página. Por ejemplo, para que el margen del texto en los cuatro costados sea la mitad del default, debemos introducir los comandos:

```
\addtolength{\textwidth}{1in}
\addtolength{\textheight}{1in}
\addtolength{\oddsidemargin}{-.5in}
\addtolength{\topmargin}{-.5in}
```

Las dos primeras líneas aumentan el tamaño horizontal y vertical del texto en 1 pulgada. Si luego restamos media pulgada del margen izquierdo y el margen superior, es claro que la distancia entre el texto y los bordes de la página sera de media pulgada, como deseábamos.

c) Espacios verticales y horizontales.

Se insertan con `\vspace` y `\hspace`:

`\vspace{3cm}`	Espacio vertical de 3 cm.
`\hspace{3cm}`	Espacio horizontal de 3 cm.

Algunos ejemplos:

Un primer párrafo de un pequeño texto.

Y un segundo párrafo separado del otro.

Tres palabras separadas del resto.

```
Un primer p\'arrafo de un
peque\~no texto.

\vspace{1cm}
Y un segundo p\'arrafo
separado del otro.

Tres\hspace{.5cm}palabras
\hspace{.5cm}separadas
del resto.
```

Si por casualidad el espacio vertical impuesto por `\vspace` debiese ser colocado al comienzo de una página, TEX lo ignora. Sería molesto visualmente que en algunas páginas el texto comenzara algunos centímetros más abajo que en el resto. Lo mismo puede ocurrir si el espacio horizontal de un `\hspace` queda al comienzo de una línea.

Los comandos `\vspace*{<longitud>}` y `\hspace*{<longitud>}` permiten que el espacio en blanco de la `<longitud>` especificada no sea ignorado. Ello es útil cuando invariablemente queremos ese espacio vertical u horizontal, aunque sea al comienzo de una página o una línea —por ejemplo, para insertar una figura.

3.10. Figuras.

Lo primero que hay que decir en esta sección es que LATEX es un excelente procesador de texto, tanto convencional como matemático. Las figuras, sin embargo, son un problema aparte.

LATEX provee un ambiente `picture` que permite realizar dibujos simples. Dentro de la estructura `\begin{picture}` y `\end{picture}` se pueden colocar una serie de comandos para dibujar líneas, círculos, óvalos y flechas, así como para posicionar texto. Infortunadamente, el proceso de ejecutar

dibujos sobre un cierto umbral de complejidad puede ser muy tedioso para generarlo directamente. Existe software (por ejemplo, `xfig`) que permite superar este problema, pudiéndose dibujar con el mouse, exportando el resultado al formato `picture` de LaTeX. Sin embargo, `picture` tiene limitaciones (no se pueden hacer líneas de pendiente arbitraria), y por tanto no es una solución óptima.

Para obtener figuras de buena calidad es imprescindible recurrir a lenguajes gráficos externos, y LaTeX da la posibilidad de incluir esos formatos gráficos en un documento. De este modo, tanto el texto como las figuras serán de la más alta calidad. Las dos mejores soluciones son utilizar Metafont o PostScript. Metafont es un programa con un lenguaje de programación gráfico propio. De hecho, los propios fonts de LaTeX fueron creados usando Metafont, y sus capacidades permiten hacer dibujos de complejidad arbitraria. Sin embargo, los dibujos resultantes no son trivialmente reescalables, y exige aprender un lenguaje de programación específico.

Una solución mucho más versátil, y adoptada como el estándar en la comunidad de usuarios de LaTeX, es el uso de PostScript. Como se mencionó brevemente en la sección 1.17, al imprimir, una máquina UNIX convierte el archivo a formato PostScript, y luego lo envía a la impresora. Pero PostScript sirve más que para imprimir, siendo un lenguaje de programación gráfico completo, con el cual podemos generar imágenes de gran calidad, y reescalables sin pérdida de resolución. Además, muchos programas gráficos permiten exportar sus resultados en formato PostScript. Por lo tanto, podemos generar nuestras figuras en alguno de estos programas (`xfig` es un excelente software, que satisface la mayor parte de nuestras necesidades de dibujos simples; `octave` o `gnuplot` pueden ser usados para generar figuras provenientes de cálculos científicos, etc.), lo cual creará un archivo con extensión `.ps` (PostScript) o `.eps` (PostScript encapsulado).[5] Luego introducimos la figura en el documento LaTeX, a través del paquete `graphicx`.

3.10.1. `graphicx.sty`

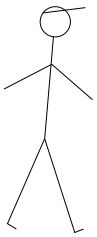
Si nuestra figura está en un archivo `figura.eps`, la instrucción a utilizar es:

```
\documentclass[12pt]{article}
\usepackage{graphicx}
\begin{document}
... Texto ...
\includegraphics[width=w, height=h]{figura.eps}
...
\end{document}
```

Los parámetros `width` y `height` son opcionales y puede omitirse uno para que el sistema escale de acuerdo al parámetro dado. Es posible variar la escala completa de la figura o rotarla usando comandos disponibles en `graphicx`.

[5] `eps` es el formato preferido, pues contiene información sobre las dimensiones de la figura, información que es utilizada por LaTeX para insertar ésta adecuadamente en el texto.

Una figura aquí:



puede hacer más agradable el
texto.

```
Una figura aqu\'{\i}:

\begin{center}
\includegraphics[height=3cm]{figura.eps}
\end{center}

puede hacer m\'as agradable
el texto.
```

En este ejemplo, indicamos sólo la altura de la figura (3cm). El ancho fue determinado de modo que las proporciones de la figura no fueran alteradas. Si no se especifica ni la altura ni el ancho, la figura es insertada con su tamaño natural.

Observemos también que pusimos la figura en un ambiente `center`. Esto no es necesario, pero normalmente uno desea que las figuras estén centradas en el texto.

3.10.2. Ambiente `figure`.

Insertar una figura es una cosa. Integrarla dentro del texto es otra. Para ello está el ambiente `figure`, que permite: (a) posicionar la figura automáticamente en un lugar predeterminado o especificado por el usuario; (b) numerar las figuras; y (c) agregar un breve texto explicativo junto a la figura.

Coloquemos la misma figura de la sección anterior dentro de un ambiente `figure`. El input:

```
\begin{figure}[h]
\begin{center}
\includegraphics[height=3cm]{figura.eps}
\end{center}
\caption{Un sujeto caminando.}
\label{caminando}
\end{figure}
```

da como resultado:

`figure` delimita lo que en TEX se denomina un *objeto flotante*, es decir, un objeto cuya posición no está determinada *a priori*, y se ajusta para obtener los mejores resultados posibles. TEX considera (de acuerdo con la tradición), que la mejor posición para colocar una figura es al principio o al final de la página. Además, lo ideal es que cada página tenga un cierto número máximo de figuras, que ninguna figura aparezca en el texto antes de que sea mencionada por primera vez, y que, por supuesto, las figuras aparezcan en el orden en que son mencionadas. Éstas y otras condiciones determinan la posición que un objeto flotante tenga al final de la compilación. Uno puede forzar la decisión de LATEX con el argumento opcional de `figure`:

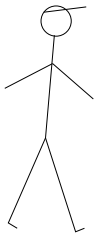


Figura 3.1: Un sujeto caminando.

`t`	(*top*)	extremo superior de la página
`b`	(*bottom*)	extremo inferior de la página
`h`	(*here*)	aquí, en el punto donde está el comando
`p`	(*page of floats*)	en una página separada al final del texto

El argumento adicional `!` suprime, para ese objeto flotante específico, cualquier restricción que exista sobre el número máximo de objetos flotantes en una página y el porcentaje de texto mínimo que debe haber en una página.

Varios de estos argumentos se pueden colocar simultánemente, su orden dictando la prioridad. Por ejemplo,

```
\begin{figure}[htbp]
...
\end{figure}
```

indica que la figura se debe colocar como primera prioridad aquí mismo; si ello no es posible, al comienzo de página (ésta o la siguiente, dependiendo de los detalles de la compilación), y así sucesivamente.

Además, `figure` numera automáticamente la figura, colocando el texto "Figura N:", y `\caption` permite colocar una leyenda, centrada en el texto, a la figura. Puesto que la numeración es automática, las figuras pueden ser referidas simbólicamente con `\label` y `\ref` (sección 3.6). Para que la referencia sea correcta, `\label` debe estar dentro del argumento de `\caption`, o después, como aparece en el ejemplo de la Figura 3.1 (`\ref{caminando}`!).

Finalmente, notemos que la figura debió ser centrada explícitamente con `center`. `figure` no hace nada más que tratar la figura como un objeto flotante, proporcionar numeración y leyenda. El resto es responsabilidad del autor.

3.11. Cartas.

Para escribir cartas debemos emplear el estilo `letter` en vez del que hemos utilizado hasta ahora, `article`. Comandos especiales permiten escribir una carta, poniendo en lugares adecuados la dirección del remitente, la fecha, la firma, etc.

A modo de ejemplo, consideremos el siguiente input:

```
\documentclass[12pt]{letter}

\usepackage[spanish]{babel}

\begin{document}

\address{Las Palmeras 3425\\
\~Nu\~noa, Santiago}
\date{9 de Julio de 1998}

\signature{Pedro P\'erez \\ Secretario}

\begin{letter}{Dr.\ Juan P\'erez \\ Las Palmeras 3425 \\
\~Nu\~noa, Santiago}
\opening{Estimado Juan}

A\'un no tenemos novedades.

Parece incre\'{\i}ble, pero los recientes acontecimientos nos han superado,
a pesar de nuestros esfuerzos.  Esperamos que mejores tiempos nos
aguarden.

\closing{Saludos,}
\cc{Arturo Prat \\ Luis Barrios}

\end{letter}
\end{document}
```

El resultado se encuentra en la próxima página.

Las Palmeras 3425
Ñuñoa, Santiago

9 de Julio de 1998

Dr. Juan Pérez
Las Palmeras 3425
Ñuñoa, Santiago

Estimado Juan

Aún no tenemos novedades.

Parece increíble, pero los recientes acontecimientos nos han superado, a pesar de nuestros esfuerzos. Esperamos que mejores tiempos nos aguarden.

Saludos,

Pedro Pérez
Secretario

Copia a: Arturo Prat
Luis Barrios

Observemos que el texto de la carta está dentro de un ambiente `letter`, el cual tiene un argumento obligatorio, donde aparece el destinatario de la carta (con su dirección opcionalmente).

Los comandos disponibles son:

`\address{<direccion>}`	`<direccion>` del remitente.
`\signature{<firma>}`	`<firma>` del remitente.
`\opening{<apertura>}`	Fórmula de `<apertura>`.
`\closing{<despedida>}`	Fórmula de `<despedida>`.
`\cc{<copias>}`	Receptores de `<copias>` (si los hubiera).

Uno puede hacer más de una carta con distintos ambientes `letter` en un mismo archivo. Cada una tomará el mismo remitente y firma dados por `\address` y `\signature`. Si deseamos que `\address` o `\signature` valgan sólo para una carta particular, basta poner dichos comandos entre el `\begin{letter}` y el `\opening` correspondiente.

Por ejemplo, la siguiente estructura:

```
\documentclass[12pt]{letter}
\begin{document}
\address{<direccion remitente>}
\date{<fecha>}
\signature{<firma>}

\begin{letter}{<destinatario 1>}
\opening<apertura 1>
...
\end{letter}

\begin{letter}{<destinatario 2>}
\address{<direccion remitente 2>}
\signature{<firma 2>}
\opening<apertura 2>
...
\end{letter}

\begin{letter}{<destinatario 3>}
\opening<apertura 3>
...
\end{letter}
\end{document}
```

dará origen a tres cartas con la misma dirección de remitente y firma, salvo la segunda.

En todos estos comandos, líneas sucesivas son indicadas con `\\`.

3.12. LaTeX y el formato pdf.

Junto con PostScript, otro formato ampliamente difundido para la transmisión de archivos, especialmente a través de Internet, es el formato pdf (Portable Document Format). Para generar un archivo pdf con LaTeX es necesario compilarlo con pdflatex. Así, pdflatex <archivo> generará un archivo <archivo>.pdf en vez del <archivo>.dvi generado por el compilador usual.

Si nuestro documento tiene figuras, sólo es posible incluirlas en el documento si están también en formato pdf. Por tanto, si tenemos un documento con figuras en PostScript, debemos introducir dos modificaciones antes de compilar con pdflatex:

a) Cambiar el argumento de \includegraphics (sección 3.10) de <archivo_figura>.eps a <archivo_figura>.pdf.

b) Convertir las figuras PostScript a pdf (con epstopdf, por ejemplo). Si tenemos una figura en el archivo <archivo_figura>.eps, entonces epstopdf <archivo_figura>.eps genera el archivo correspondiente <archivo_figura>.pdf.

Observar que el mismo paquete graphicx descrito en la sección 3.10 para incluir figuras PostScript permite, sin modificaciones, incluir figuras en pdf.

3.13. Modificando LaTeX.

Esta sección se puede considerar "avanzada". Normalmente uno se puede sentir satisfecho con el desempeño de LaTeX, y no es necesaria mayor intervención. A veces, dependiendo de la aplicación y del autor, nos gustaría modificar el comportamiento default. Una alternativa es definir nuevos comandos que sean útiles para nosotros. Si esos nuevos comandos son abundantes, o queremos reutilizarlos frecuentemente en otros documentos, lo conveniente es considerar crear un nuevo paquete o incluso una nueva clase. Examinaremos a continuación los elementos básicos de estas modificaciones.

3.13.1. Definición de nuevos comandos.

El comando \newcommand

Un nuevo comando se crea con:

```
\newcommand{<comando>}{<accion>}
```

El caso más sencillo es cuando una estructura se repite frecuentemente en nuestro documento. Por ejemplo, digamos que un sujeto llamado Cristóbal no quiere escribir su nombre cada vez que aparece en su documento:

Mi nombre es Cristóbal.
Sí, como oyes, Cristóbal.
Cristóbal Loyola.

```
\newcommand{\nombre}{Crist\'obal}
...
\begin{document}
...
Mi nombre es \nombre. S\'{\i}, como oyes,
\nombre. \nombre\ Loyola.
```

Un `\newcommand` puede aparecer en cualquier parte del documento, pero lo mejor es que esté en el preámbulo, de modo que sea evidente qué nuevos comandos están disponibles en el presente documento. Observemos además que la definición de un comando puede contener otros comandos (en este caso, `\'`). Finalmente, notamos que ha sido necesario agregar un espacio explícito con `\ `, al escribir "Cristóbal Loyola": recordemos que un comando comienza con un backslash y termina con el primer carácter que no es letra. Por tanto, `\nombre Loyola` ignora el espacio al final de `\nombre`, y el output sería "CristóbalLoyola".

También es posible definir comandos que funcionen en modo matemático:

Sea $\dot{x}$ la velocidad, de modo que $\dot{x}(t) > 0$ si $t < 0$.

```
\newcommand{\vel}{\dot x}

Sea $\vel$ la velocidad, de modo que
$ \vel(t)> 0$ si $t<0$.
```

Como `\vel` contiene un comando matemático (`\dot`), `\vel` sólo puede aparecer en modo matemático.

Podemos también incluir la apertura de modo matemático en la definición de `\vel`: `\newcommand{\vel}{$\dot x$}`. De este modo, `\vel` (no `$\vel$`) da como output directamente $\dot{x}$. Sin embargo, esta solución no es óptima, porque la siguiente ocurrencia de `\vel` da un error. En efecto, si `\vel` = `$\dot x$`, entonces `$ \vel(t)>0$` = `$ $\dot x$> 0$`. En tal caso, LaTeX ve que un modo matemático se ha abierto y cerrado inmediatamente, conteniendo sólo un espacio entremedio, y luego, *en modo texto*, viene el comando `\dot`, que es matemático: LaTeX acusa un error y la compilación se detiene.

La solución a este problema es utilizar el comando `\ensuremath`, que asegura que haya modo matemático, pero si ya hay uno abierto, no intenta volverlo a abrir:

Sea $\dot{x}$ la velocidad, de modo que $\dot{x}(t) > 0$ si $t < 0$.

```
\newcommand{\vel}{\ensuremath{\dot x}}

Sea \vel\ la velocidad, de modo que
$ \vel(t)> 0$ si $t<0$.
```

Un caso especial de comando matemático es el de operadores tipo logaritmo (ver Tabla 3.9). Si queremos definir una traducción al castellano de `\sin`, debemos usar el comando `\DeclareMathOperator` disponible via `amsmath`:

Ahora podemos escribir en castellano, $\operatorname{sen} x$.

```
\usepackage{amsmath}
\DeclareMathOperator{\sen}{sen}
...
Ahora podemos escribir en castellano, $\sen x$.
```

A diferencia de `\newcommand`, `\DeclareMathOperator` sólo puede aparecer en el preámbulo del documento.

Un nuevo comando puede también ser usado para ahorrar tiempo de escritura, reemplazando comandos largos de LaTeX:

1. El primer caso.
2. Ahora el segundo.
3. Y el tercero.

```
\newcommand{\be}{\begin{enumerate}}
\newcommand{\ee}{\end{enumerate}}

\be
\item El primer caso.
\item Ahora el segundo.
\item Y el tercero.
\ee
```

Nuevos comandos con argumentos

Podemos también definir comandos que acepten argumentos. Si el sujeto anterior, Cristóbal, desea escribir cualquier nombre precedido de "Nombre:" en itálica, entonces puede crear el siguiente comando:

Nombre: Cristóbal
Nombre: Violeta

```
\newcommand{\nombre}[1]{\textit{Nombre:} #1}

\nombre{Crist\'obal}

\nombre{Violeta}
```

Observemos que `\newcommand` tiene un argumento opcional, que indica el número de argumentos que el nuevo comando va a aceptar. Esos argumentos se indican, dentro de la definición del comando, con `#1`, `#2`, etc. Por ejemplo, consideremos un comando que acepta dos argumentos:

$f(x,y) + f(x_3, y*) = 0$.

```
\newcommand{\fn}[2]{f(#1,#2)}

$$ \fn{x}{y} + \fn{x_3}{y*} = 0 \ . $$
```

En los casos anteriores, todos los argumentos son obligatorios. LaTeX permite definir comandos con un (sólo un) argumento opcional. Si el comando acepta n argumentos, el argumento opcional es el `#1`, y se debe indicar, en un segundo paréntesis cuadrado, su valor default. Así, podemos modificar el comando `\fn` del ejemplo anterior para que el primer argumento sea opcional, con valor default x:

$f(x,y) + f(x_3, y*) = 0$.

```
\newcommand{\fn}[2][x]{f(#1,#2)}

$$ \fn{y} + \fn[x_3]{y*} = 0 \ . $$
```

Redefinición de comandos

Ocasionalmente no nos interesa definir un nuevo comando, sino redefinir la acción de un comando preexistente. Esto se hace con `\renewcommand`:

```
La antigua versión de ldots:    La antigua versi\'on de  {\tt ldots}: \ldots
...
La nueva versión de ldots:      \renewcommand{\ldots}{\textbullet \textbullet
•••                             \textbullet}

                                La nueva versi\'on de {\tt ldots}: \ldots
```

Párrafos y cambios de línea dentro de comandos

En el segundo argumento de `\newcommand` o `\renewcommand` puede aparecer cualquier comando de LaTeX, pero ocasionalmente la aparición de líneas en blanco (para forzar un cambio de párrafo) puede provocar problemas. Si ello ocurre, podemos usar `\par`, que hace exactamente lo mismo. Además, la definición del comando queda más compacta:

```
\newcommand{\comandolargo}{\par Un nuevo comando que incluye un cambio de
  p\'arrafo, porque deseamos incluir bastante texto.\par \'Este es el
  nuevo p\'arrafo.\par}

Observemos en acci\'on el comando: \comandolargo Listo.
```

da como resultado:

Observemos en acción el comando:
Un nuevo comando que incluye un cambio de párrafo, porque deseamos incluir bastante texto.
Éste es el nuevo párrafo.
Listo.

Un ejemplo más útil ocurre cuando queremos asegurar un cambio de párrafo, por ejemplo, para colocar un título de sección:

```
Observemos en acción el co-     \newcommand{\seccion}[1]{\par\vspace{.5cm}
mando:                          {\bf Secci\'on: #1}\par\vspace{.5cm}}

Sección: Ejemplo                Observemos en acci\'on el comando:
                                \seccion{Ejemplo} Listo.
```

Listo.

Además de las líneas en blanco, los cambios de línea pueden causar problemas dentro de la definición de un nuevo comando. El ejemplo anterior, con el comando `\seccion`, es un buen ejemplo: notemos que cuando se definió, pusimos un cambio de línea después de `\vspace{.5cm}`. Ese cambio de línea es interpretado (como todos los cambios de línea) como un espacio en blanco, y es posible que, bajo ciertas circunstancias, ese espacio en blanco produzca un output no deseado. Para ello basta utilizar sabiamente el carácter `%`, que permite ignorar todo el resto de la línea, *incluyendo el cambio de línea*. Ilustremos lo anterior con los siguientes tres comandos, que subrayan (comando `\underline`) una palabra, y difieren sólo en el uso de `%` para borrar cambios de línea:

Notar la diferencia entre:
Un texto de prueba ,
Un texto de prueba , y
Un texto de prueba.

```
\newcommand{\texto}{
Un texto de prueba
}
\newcommand{\textodos}{%
Un texto de prueba
}
\newcommand{\textotres}{%
Un texto de prueba%
}

Notar la diferencia entre:

\underline{\texto},
\underline{\textodos},
y
\underline{\textotres}.
```

`\texto` conserva espacios en blanco antes y después del texto, `\textodos` sólo el espacio en blanco después del texto, y `\textotres` no tiene espacios en blanco alrededor del texto.

Nuevos ambientes

Nuevos ambientes en LaTeX se definen con `\newenvironment`:

```
\newenvironment{<ambiente>}{<comienzo ambiente>}{<final ambiente>}
```

define un ambiente `<ambiente>`, tal que `\begin{ambiente}` ejecuta los comandos `<comienzo ambiente>`, y `\end{ambiente}` ejecuta los comandos `<final ambiente>`.

Definamos un ambiente que, al comenzar, cambia el font a itálica, pone una línea horizontal (`\hrule`) y deja un espacio vertical de .3cm, y que al terminar cambia de párrafo, coloca `XXX` en sans serif, deja un nuevo espacio vertical de .3cm, y vuelve al font roman:

```
\newenvironment{na}{\it \hrule \vspace{.3cm}}{\par\sf XXX \vspace{.3cm}\rm}
```

Entonces, con

```
\begin{na}
  Hola a todos. Es un placer saludarlos en este d\'{\i}a tan especial.

Nunca esper\'e una recepci\'on tan calurosa.
\end{na}
```

obtenemos:

Hola a todos. Es un placer saludarlos en este día tan especial.
Nunca esperé una recepción tan calurosa.
XXX

Los nuevos ambientes también pueden ser definidos de modo que acepten argumentos. Como con `\newcommand`, basta agregar como argumento opcional a `\newenvironment` un número que indique cuántos argumentos se van a aceptar:

```
\newenvironment{<ambiente>}[n]{<comienzo ambiente>}{<final ambiente>}
```

Dentro de `<comienzo ambiente>`, se alude a cada argumento como `#1`, `#2`, etc. Los argumentos no pueden ser usados en los comandos de cierre del ambiente (`<final ambiente>`). Por ejemplo, modifiquemos el ambiente `na` anterior, de modo que en vez de colocar una línea horizontal al comienzo, coloque lo que le indiquemos en el argumento:

```
\newenvironment{na}[1]{\it #1 \vspace{.3cm}}{\par\sf XXX\hrule\vspace{.3cm}\rm}
```

Ahora usémoslo dos veces, cada una con un argumento distinto:

El mismo ejemplo anterior, ahora es

Hola a todos...
XXX

Pero podemos ahora cambiar el comienzo:
XXX Hola a todos...

XXX

```
El mismo ejemplo anterior, ahora es

\begin{na}{\hrule}
  Hola a todos...
\end{na}

Pero podemos ahora cambiar el comienzo:

\begin{na}{\it XXX}
  Hola a todos...
\end{na}
```

3.13.2. Creación de nuevos paquetes y clases

Si la cantidad de nuevos comandos y/o ambientes que necesitamos en nuestro documento es suficientemente grande, debemos considerar crear un nuevo paquete o una nueva clase. Para ello hay que tener clara la diferencia entre uno y otro. En general, se puede decir que si nuestros comandos involucran alterar la apariencia general del documento, entonces corresponde crear una nueva clase (`.cls`). Si, por el contrario, deseamos que nuestros comandos funcionen en un amplio rango de circunstancias, para diversas apariencias del documento, entonces lo adecuado es un paquete (`.sty`).

Consideremos por ejemplo la experiencia de los autores de estos apuntes. Para crear estos apuntes necesitamos básicamente la clase `book`, con ciertas modificaciones: márgenes más pequeños, inclusión automática de los paquetes `amsmath`, `babel` y `graphicx`, entre otros, y definición de ciertos ambientes específicos. Todo ello afecta la apariencia de este documento, cambiándola de manera apreciable, pero a la vez de un modo que en general no deseamos en otro tipo de documento. Por ello lo hemos compilado usando una clase adecuada, llamada `mfm2.cls`.

Por otro lado, uno de los autores ha necesitado escribir muchas tareas, pruebas y controles de ayudantía en su vida, y se ha convencido de que su trabajo es más fácil creando una clase `tarea.cls`, que sirve para esos tres propósitos, definiendo comandos que le permiten especificar fácilmente la

fecha de entrega de la tarea, o el tiempo disponible para una prueba, los nombres del profesor y el ayudante, etc., una serie de comandos específicos para sus necesidades.

Sin embargo, tanto en este documento que usa `mfm2.cls`, como en las tareas y pruebas que usan `tarea.cls`, se utilizan algunos comandos matemáticos que no vienen con LaTeX, pero que son recurrentes, como `\sen` (la función seno en castellano), `\modulo` (el módulo de un vector), o `\TLaplace` (la transformada de Laplace). Para que estos comandos estén disponibles en cualquier tipo de documento, necesitamos reunirlos en un paquete, en este caso `addmath.sty`. De este modo, `mfm2.cls`, `tarea.cls` o cualquier otra clase pueden llamar a este paquete y utilizar sus comandos.

Estructura básica.

La estructura básica de un paquete o una clase es:

a) Identificación: Información general (nombre del paquete, fecha de creación, etc.). (Obligatoria.)

b) Declaraciones preliminares: Opcionales, dependiendo del paquete o clase en cuestión.

c) Opciones: Comandos relacionados con el manejo de las opciones con las cuales el paquete o clase pueden ser invocados. (Opcional.)

d) Más declaraciones: Aquí van los comandos que constituyen el cuerpo de la clase o paquete. (Obligatoria: si no hay ninguna declaración, el paquete o clase no hace nada, naturalmente.)

La identificación está consituida por las siguientes dos líneas, que deben ir al comienzo del archivo:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{<paquete>}[<fecha> <otra informacion>]
```

La primera línea indica a LaTeX que éste es un archivo para LaTeX 2_ε. La segunda línea especifica que se trata de un paquete, indicando el nombre del mismo (es decir, el nombre del archivo sin extensión) y, opcionalmente, la fecha (en formato `YYYY/MM/DD`) y otra información relevante. Por ejemplo, nuestro paquete `addmath.sty` comienza con las líneas:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{addmath}[1998/09/30 Macros matematicos adicionales (VM)]
```

Si lo que estamos definiendo es una clase, usamos el comando `\ProvidesClass`. Para nuestra clase `mfm2.cls`:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{mfm2}[2002/03/25 Estilo para apuntes MFM II (VM)]
```

A continuación de la identificación vienen los comandos que se desean incorporar a través de este paquete o clase.

Como hemos dicho, `addmath.sty` contiene muchos nuevos comandos matemáticos que consideramos necesario definir mientras escribíamos estos apuntes. Veamos los contenidos de una versión simplificada de dicho paquete:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{addmath}[1998/09/30 Macros matematicos adicionales (VM)]

\newcommand{\prodInt}[2]{\ensuremath \left(\, #1\, |\, #2\, \right ) }
\newcommand{\promedio}[1]{\langle #1 \rangle}
\newcommand{\intii}{\int_{-\infty}^{\infty}}
\newcommand{\grados}{\ensuremath{^\circ}}
\newcommand{\Hipergeometrica}[4]{{}_2F_1\left (#1, #2, #3\, ; #4\right )}
...
```

De este modo, incluyendo en nuestro documento el paquete con `\usepackage{addmath}`, varios nuevos comandos están disponibles:

$(x \mid y)$	`\prodInt{x}{y}`
$\langle x \rangle$	`\promedio{x}`
$\int_{-\infty}^{\infty} dz\, f(z)$	`\intii dz\, f(z)`
$\angle ABC = 90^\circ$	`\angle\, ABC = 90\grados`
${}_2F_1\left(a, b, c\,; d\right)$	`\Hipergeometrica{a}{b}{c}{d}`

Incluyendo otros paquetes y clases

Los comandos `\RequirePackage` y `\LoadClass` permiten cargar un paquete o una clase, respectivamente.[6]. Esto es de gran utilidad, pues permite construir un nuevo paquete o clase aprovechando la funcionalidad de otros ya existentes.

Así, nuestro paquete `addmath.sty` define bastantes comandos, pero nos gustaría definir varios más que sólo pueden ser creados con las herramientas de $\mathcal{AMS}$-LaTeX. Cargamos entonces en `addmath.sty` el paquete `amsmath` y otros relacionados, y estamos en condiciones de crear más comandos:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{addmath}[1998/09/30 Macros matematicos adicionales (VM)]

\RequirePackage{amsmath}
\RequirePackage{amssymb}
\RequirePackage{euscript}
...
\newcommand{\norma}[1]{\ensuremath \left\lVert\, #1 \,\right\rVert}
\newcommand{\intC}{{\sideset{^*}{}\int}}
\DeclareMathOperator{\senh}{senh}
...
```

[6]Estos comandos sólo se pueden usar en un archivo `.sty` o `.cls` Para documentos normales, la manera de cargar un paquete es `\usepackage`, y para cargar una clase es `\documentclass`.

Por ejemplo:

$\| x \|$	`\norma{x}`
$\overset{*}{\int} dz\, f(z)$	`\intC dz \, f(z)`
$\operatorname{senh}(2y)$	`\senh (2y)`

La posibilidad de basar un archivo `.sty` o `.cls` en otro es particularmente importante para una clase, ya que contiene una gran cantidad de comandos y definiciones necesarias para compilar el documento exitosamente. Sin embargo, un usuario normal, aun cuando desee definir una nueva clase, estará interesado en modificar sólo parte del comportamiento. Con `\LoadClass`, dicho usuario puede cargar la clase sobre la cual se desea basar, y luego introducir las modificaciones necesarias, facilitando enormemente la tarea.

Por ejemplo, al preparar este documento fue claro desde el comienzo que se necesitaba esencialmente la clase `book`, ya que sería un texto muy extenso, pero también era claro que se requerían ciertas modificaciones. Entonces, en nuestra clase `mfm2.cls` lo primero que hacemos es cargar la clase `book`, más algunos paquetes necesarios (incluyendo nuestro `addmath`), y luego procedemos a modificar o añadir comandos:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{mfm2}[2002/03/25 Estilo para apuntes MFM II (VM)]

\LoadClass[12pt]{book}

\RequirePackage[spanish]{babel}
\RequirePackage{enumerate}
\RequirePackage{addmath}
```

En un archivo `.sty` o un `.cls` se pueden cargar varios paquetes con `\RequirePackage`. `\LoadClass`, en cambio, sólo puede aparecer en un `.cls`, y sólo es posible usarlo una vez (ya que normalmente clases distintas son incompatibles entre sí).

Manejo de opciones

En el último ejemplo anterior, la clase `mfm2` carga la clase `book` con la opción `12pt`. Esto significa que si nuestro documento comienza con `\documentclass{mfm2}`, será compilado de acuerdo a la clase `book`, en 12 puntos. No es posible cambiar esto desde nuestro documento. Sería mejor que pudiéramos especificar el tamaño de letra fuera de la clase, de modo que `\documentclass{mfm2}` dé un documento en 10 puntos, y `\documentclass[12pt]{mfm2}` uno en 12 puntos. Para lograr esto hay que poder pasar opciones desde la clase `mfm2` a `book`.

El modo más simple de hacerlo es con `\LoadClassWithOptions`. Si `mfm2.cls` ha sido llamada con opciones `<opcion1>,<opcion2>, etc.`, entonces `book` será llamada con las mismas opciones. Por tanto, basta modificar en `mfm2.cls` la línea `\LoadClass[12pt]{book}` por:

```
\LoadClassWithOptions{book}
```

`\RequirePackageWithOptions` es el comando análogo para paquetes. Si una clase o un paquete llaman a un paquete `<paquete_base>` y desean pasarle todas las opciones con las cuales han sido invocados, basta indicarlo con:

```
\RequirePackageWithOptions{<paquete_base>}
```

El ejemplo anterior puede ser suficiente en muchas ocasiones, pero en general uno podría llamar a nuestra nueva clase, `mfm2`, con opciones que no tienen nada que ver con `book`. Por ejemplo, podríamos llamarla con opciones `spanish,12pt`. En tal caso, debería pasarle `spanish` a babel, y `12pt` a book. Más aún, podríamos necesitar definir una *nueva* opción, que no existe en ninguna de las clases o paquetes cargados por `book`, para modificar el comportamiento de `mfm2.cls` de cierta manera específica no prevista. Estas dos tareas, discriminar entre opciones antes de pasarla a algún paquete determinado, y crear nuevas opciones, constituyen un manejo más avanzado de opciones. A continuación revisaremos un ejemplo combinado de ambas tareas, extraido de la clase con la cual compilamos este texto, `mfm2.cls`.

La idea es poder llamar a `mfm2` con una opción adicional `keys`, que permita agregar al `dvi` información sobre las etiquetas (dadas con `\label`) de ecuaciones, figuras, etc., que aparezcan en el documento (veremos la utilidad y un ejemplo de esto más adelante). Lo primero es *declarar* una nueva opción, con:

```
\DeclareOption{<opcion>}{<comando>}
```

`<opcion>` es el nombre de la nueva opción a declarar, y `<comando>` es la serie de comandos que se ejecutan cuando dicha opción es especificada.

Así, nuestro archivo `mfm2.cls` debe ser modificado:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{mfm2}[2002/03/25 Estilo para apuntes MFM II (VM)]
...
\DeclareOption{keys}{...}
...
\ProcessOptions\relax
...
```

Observamos que después de declarar la o las opciones (en este caso `keys`), hay que *procesarlas*, con `\ProcessOptions`.[7]

Las líneas anteriores permiten que `\documentclass{mfm2}` y `\documentclass[keys]{mfm2}` sean ambas válidas, ejecutándose o no ciertos comandos dependiendo de la forma utilizada.

Si ahora queremos que `\documentclass[keys,12pt]{mfm2}` sea una línea válida, debemos procesar `keys` dentro de `mfm2.cls`, y pasarle a `book.cls` las opciones restantes. El siguiente es el código definitivo:

[7] `\relax` es un comando de TEX que, esencialmente, no hace nada, ni siquiera introduce un espacio en blanco, y es útil incluirlo en puntos críticos de un documento, como en este ejemplo.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{mfm2}[2002/03/25 Estilo para apuntes MFM II (VM)]
\newif\ifkeys\keysfalse
\DeclareOption{keys}{\keystrue}
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{book}}
\ProcessOptions\relax
\LoadClass{book}

\RequirePackage[spanish]{babel}
\RequirePackage{amsmath}
\RequirePackage{theorem}
\RequirePackage{epsfig}
\RequirePackage{ifthen}
\RequirePackage{enumerate}
\RequirePackage{addmath}
\ifkeys\RequirePackage[notref,notcite]{showkeys}\fi

<nuevos comandos de la clase mfm2.cls>
```

Sin entrar en demasiados detalles, digamos que la opción `keys` tiene el efecto de hacer que una cierta variable lógica `\ifkeys`, sea verdadera (cuarta línea del código). La siguiente línea (`\DeclareOption*...`) hace que todas las opciones que no han sido procesadas (`12pt`, por ejemplo) se pasen a la clase `book`. A continuación se procesan las opciones con `\ProcessOptions`, y finalmente se carga la clase `book`.

Las líneas siguientes cargan todos los paquetes necesarios, y finalmente se encuentran todos los nuevos comandos y definiciones que queremos incluir en `mfm2.cls`.

Observemos que la forma particular en que se carga el paquete `showkeys`. Ésa es precisamente la función de la opción `keys` que definimos: `showkeys.sty` se carga con ciertas opciones sólo si se da la opción `keys`.

¿Cuál es su efecto? Consideremos el siguiente texto de ejemplo, en que `mfm2` ha sido llamada sin la opción `keys`:

```
\documentclass[12pt]{mfm2}
\begin{document}
La opci\'on \verb+keys+ resulta muy \'util cuando tengo objetos numerados
autom\'aticamente, como una ecuaci\'on:
\begin{equation}
  \label{newton}
  \vec F = m \vec a \ .
\end{equation}
y luego quiero referirme a ella: Ec.\  \eqref{newton}.
```

En el primer caso, se ha compilado sin la opción `keys`, y en el segundo con ella. El efecto es que, si se usa un `\label` en cualquier parte del documento, aparece en el margen derecho una caja con el nombre de dicha etiqueta (en este caso, `newton`). Esto es útil para cualquier tipo de documentos, pero lo es especialmente en textos como estos apuntes, muy extensos y con abundantes referencias. En tal caso, tener un modo visual, rápido, de saber los nombres de las ecuaciones sin tener que revisar trabajosamente el archivo fuente es una gran ayuda. Así, versiones preliminares pueden ser compiladas con la opción `keys`, y la versión final sin ella, para no confesar al lector nuestra mala memoria o nuestra comodidad.

Caso 1: `\documentclass[12pt]{mfm2}`

La opción `keys` resulta muy útil cuando tengo objetos numerados automáticamente, como una ecuación:

$$\vec{F} = m\vec{a} \ . \tag{1}$$

y luego quiero referirme a ella: Ec. (1).

Caso 2: `\documentclass[keys,12pt]{mfm2}`

La opción `keys` resulta muy útil cuando tengo objetos numerados automáticamente, como una ecuación:

$$\vec{F} = m\vec{a} \ . \tag{1}$$ `newton`

y luego quiero referirme a ella: Ec. (1).

3.14. Errores y advertencias.

3.14.1. Errores.

Un mensaje de error típico tiene la forma:

```
LaTeX error. See LaTeX manual for explanation.
             Type H <return> for immediate help.
! Environment itemie undefined.
\@latexerr ...or immediate help.}\errmessage {#1}
                                                \endgroup
l.140 \begin{itemie}

?
```

La primera línea nos comunica que LaTeX ha encontrado un error. A veces los errores tienen que ver con procesos más internos, y son encontrados por TeX. Esta línea nos informa quién encontró el error.

La tercera línea comienza con un signo de exclamación. Éste es el indicador del error. Nos dice de qué error se trata.

Las dos líneas siguientes describen el error en términos de comandos de bajo nivel.

La línea 6 nos dice dónde ocurrió el error: la línea 140 en este caso. Además nos informa del texto conflictivo: `\begin{itemie}`.

En realidad, el mensaje nos indica dónde LaTeX advirtió el error por primera vez, que no es necesariamente el punto donde el error se cometió. Pero la gran mayoría de las veces la indicación es precisa. De hecho, es fácil darse cuenta, con la tercera línea
(`Environment itemie undefined`)
y la sexta (`\begin{itemie}`) que el error consistió en escribir `itemie` en vez de `itemize`. La información de LaTeX es clara en este caso y nos dice correctamente qué ocurrió y dónde.

Luego viene un `?`. LaTeX está esperando una respuesta de nosotros. Tenemos varias alternativas. Comentaremos sólo cuatro, típicamente usadas:

(a) `h <Enter>`

Solicitamos ayuda. TeX nos explica brevemente en qué cree él que consiste el error y/o nos da alguna recomendación.

(b) `x <Enter>`

Abortamos la compilación. Deberemos volver al editor y corregir el texto. Es la opción más típica cuando uno tiene ya cierta experiencia, pues el mensaje basta para reconocer el error.

(c) `<Enter>`

Ignoramos el error y continuamos la compilación. TEX hace lo que puede. En algunos casos esto no tiene consecuencias graves y podremos llegar hasta el final del archivo sin mayores problemas. En otros casos, ignorar el error puede provocar que ulteriores comandos —perfectamente válidos en principio— no sean reconocidos y, así, acumular muchos errores más. Podemos continuar con `<Enter>` sucesivos hasta llegar al final de la compilación.

(d) `q <Enter>`

La acción descrita en el punto anterior puede llegar a ser tediosa o infinita. `q` hace ingresar a TEX en `batchmode`, modo en el cual la compilación prosigue ignorando todos los errores hasta el final del archivo, sin enviar mensajes a pantalla y por ende sin que debamos darle infinitos `<Enter>`.

Las opciones (c) y (d) son útiles cuando no entendemos los mensajes de error. Como TEX seguirá compilando haciendo lo mejor posible, al mirar el `dvi` puede que veamos más claramente dónde comenzaron a ir mal las cosas y, por tanto, por qué.

Como dijimos, LATEX indica exactamente dónde encontró el error, de modo que hemos de ponerle atención. Por ejemplo, si tenemos en nuestro documento la línea:

```
... un error inesperado\fotnote{En cualquier punto.}
puede decidir...
```

generaría el mensaje de error:

```
! Undefined control sequence.

l.249 ...un error inesperado\fotnote
                                    {En cualquier punto.}
?
```

En la línea de localización, LATEX ha cortado el texto justo después del comando inexistente. LATEX no sólo indica la línea en la cual detectó el error, sino el punto de ella donde ello ocurrió. (En realidad, hizo lo mismo —cortar la línea para hacer resaltar el problema— en el caso expuesto en la pág. 112, pero ello ocurrió en medio de comandos de bajo nivel, así que no era muy informativo de todos modos.)

Errores más comunes.

Los errores más comunes son:

a) Comando mal escrito.

b) Paréntesis cursivos no apareados.

c) Uso de uno de los caracteres especiales `#`, `$`, `%`, `&`, `_`, `{`, `}`, `~`, `^`, `\` como texto ordinario.

d) Modo matemático abierto de una manera y cerrado de otra, o no cerrado.

e) Ambiente abierto con `\begin...` y cerrado con un `\end...` distinto.

f) Uso de un comando matemático fuera de modo matemático.

g) Ausencia de argumento en un comando que lo espera.

h) Línea en blanco en ambiente matemático.

Algunos mensajes de error.

A continuación, una pequeña lista de errores (de LaTeX y TeX) en orden alfabético, y sus posibles causas.

`*`

Falta `\end{document}`. (Dar `Ctrl-C` o escribir `\end{document}` para salir de la compilación.)

`! \begin{...} ended by \end{...}`

Error e) de la Sec. 3.14.1. El nombre del ambiente en `\end{...}` puede estar mal escrito, sobra un `\begin` o falta un `\end`.

`! Double superscript` (o `subscript`).

Una expresión como `x^2^3` o `x_2_3`. Si se desea obtener x^{2^3} (x_{23}), escribir `{x^2}^3` (`{x_2}_3`).

`! Environment ... undefined.`

`\begin{...}` con un argumento que corresponde a un ambiente no definido.

`! Extra alignment tab has been changed.`

En un `tabular` o `array` sobra un `&`, falta un `\\`, o falta una `c`, `l` ó `r` en el argumento obligatorio.

`! Misplaced alignment tab character &.`

Un `&` aparece fuera de un `tabular` o `array`.

`! Missing $ inserted.`

Errores c), d), f), h) de la Sec. 3.14.1.

`! Missing {` (o `}`) `inserted.`

Paréntesis cursivos no apareados.

`! Missing \begin{document}.`

Falta `\begin{document}` o hay algo incorrecto en el preámbulo.

`! Missing number, treated as zero.`

Falta un número donde LaTeX lo espera: `\hspace{}`, `\vspace cm`, `\setlength{\textwidth}{a}`, etc.

`! Something's wrong -- perhaps a missing \item.`

Posiblemente la primera palabra después de un `\begin{enumerate}` o `\begin{itemize}` no es `\item`.

`! Undefined control sequence.`

Aparece una secuencia `\<palabra>`, donde `<palabra>` no es un comando.

3.14.2. Advertencias.

La estructura de una advertencia de LaTeX es:

`LaTeX warning. <mensaje>.`

Algunos ejemplos:

`Label `...' multiply defined.`

Dos `\label` tienen el mismo argumento.

`Label(s) may have changed. Rerun to get cross-references right.`

Los números impresos por `\ref` y `\pageref` pueden ser incorrectos, pues los valores correspondientes cambiaron respecto al contenido del `aux` generado en la compilación anterior.

`Reference `...' on page ... undefined.`

El argumento de un `\ref` o un `\pageref` no fue definido por un `\label`.

TeX también envía advertencias. Se reconocen porque no comienzan con `TeX warning`. Algunos ejemplos.

`Overfull \hbox ...`

TeX no encontró un buen lugar para cortar una línea, y puso más texto en ella que lo conveniente.

`Overfull \vbox ...`

TeX no encontró un buen lugar para cortar una página, y puso más texto en ella que lo conveniente.

`Underfull \hbox ...`

TEX construyó una línea con muy poco material, de modo que el espacio entre palabras puede ser excesivo.

`Underfull \vbox ...`

TEX construyó una página con muy poco material, de modo que los espacios verticales (entre párrafos) pueden ser excesivos.

Las advertencias de LATEX siempre deben ser atendidas. Una referencia doblemente definida, o no compilar por segunda vez cuando LATEX lo sugiere, generará un resultado incorrecto en el `dvi`. Una referencia no definida, por su parte, hace aparecer un signo **??** en el texto final. Todos resultados no deseados, por cierto.

Las advertencias de TEX son menos decisivas. Un overfull o underfull puede redundar en que alguna palabra se salga del margen derecho del texto, que el espaciado entre palabras en una línea sea excesivo, o que el espacio vertical entre párrafos sea demasiado. Los estándares de calidad de TEX son altos, y por eso envía advertencias frecuentemente. Pero generalmente los defectos en el resultado final son imperceptibles a simple vista, o por lo menos no son suficientes para molestarnos realmente. A veces sí, por supuesto, y hay que estar atentos. Siempre conviene revisar el texto y prestar atención a estos detalles, aunque ello sólo tiene sentido al preparar la versión definitiva del documento.

Capítulo 4

Introducción a programación.

versión 1.0, 30 de Agosto del 2007

En este capítulo se intentará dar los elementos básicos de lo que es un lenguaje de programación y lo que es programar.

4.1. ¿Qué es programar?

A continuación, presentamos algunas alternativas de respuesta a esta pregunta:

- Hacer un programa.
- Escribir una secuencia de instrucciones para que un computador haga algo que uno le pide.
- Darle, de alguna forma, una secuencia de pasos lógicos para que un computador los ejecute con la intención de alcanzar algún objetivo.
- Escribir una precisa secuencia de comandos o instrucciones, en algún lenguaje que el computador entienda (a este tipo de lenguaje lo llamaremos *lenguaje de programación*) para que luego el computador las realice exactamente, paso a paso.

Un programa es un archivo que puede ser tan corto como una sola línea de código, o tan largo como varios millones de líneas de código.

4.2. Lenguajes de programación.

Existen diferentes tipos de lenguajes de programación, algunos más cercanos a la máquina y menos al programador; otros más cercanos al programador y distantes de la máquina. Realmente existe toda una jerarquía entre los lenguajes de programación. Veamos algunos ejemplos:

4.2.1. Código de Máquina binario.

Es el lenguaje de la CPU, y el lenguaje de más bajo nivel. Compuesto de 0 y 1 `binario`, lo que está muy cerca de la máquina pero muy lejos del programador. Una de sus grandes desventajas es que no es fácil de escribir o de leer para el programador.

Un programa simple, como *Hola mundo*, se vería en código binario algo así como:

```
10001010101010011110010
10001011010101000111010
11000101000101010000111
00101010101010010110000
11110010101010101000011
10001010010101010101001
00101010101101010101001
```

4.2.2. Lenguaje de Ensamblador (Assembler).

El paso siguiente es reemplazar los 1 y 0 por una secuencia de abreviaturas de lenguaje de máquina, este tipo de lenguaje se conoce como lenguaje de Ensamblador o *Assembler*. Está cerca de la máquina pero no tanto como el anterior y esta un poco más cerca del programador. Veamos el programa *Hola mundo* en lenguaje de Ensamblador para la familia de procesadores X86.

```
title Programa Hola Mundo (hola.asm)
; Este programa muestra "Hola, Mundo!" dosseg
.model small
.stack 100h .data
hello_message db 'Hola, Mundo!', 0dh, 0ah, '$' .code
main proc
mov ax, @ .data
mov ds,ax mov ah,9
mov dx, offset hello_message
int 21h mov ax,4C00h
int 21h
main endp
end main
```

4.2.3. Lenguaje de alto nivel.

Utilizan declaraciones y sentencias con palabras y expresiones algebraicas. Estos lenguajes fueron inicialmente desarrollados en las décadas del 50 y 60. Son lenguajes que están más cerca del programador que de la máquina, por lo tanto, necesitan una etapa de tradución

para que los entienda la máquina. Este proceso se puede hacer de dos maneras: traduciendo el programa fuente todo de una vez, lo que llamaremos lenguaje compilando; o traduciendo el programa fuente línea por línea, lo que llamaremos lenguaje interpretando.

Lenguajes Compilados.

En este caso, otro programa (el compilador) lee el programa fuente, un archivo en ASCII donde se encuentran el listado de instruciones y lo reescribe en un archivo binario, en lenguaje de máquina para que la CPU pueda entenderlo. Esto se hace de una sola vez y el programa final se guarda en esta nueva forma (un ejecutable). El ejecutable de un programa que es compilado se estima que será considerablemente más largo que el original, programa fuente. Algunos de los lenguajes compilados más notables son Fortran, C y C++. Un ejemplo del programa *Hola mundo* escrito en C++ es dado a continuación:

```
//
// Programa Hola Mundo
//
#include <iostream>

using namespace std;

int main()
{
    cout << "Hola mundo" << endl;
    return 0;
}
```

4.2.4. Lenguajes interpretados.

En este caso otro programa (el intérprete) traduce las declaraciones del programa original a lenguaje de máquina, línea por línea, a medida que va ejecutando el programa original. Un programa interpretado suele ser más pequeño que uno compilado pero tardará más tiempo en ser ejecutado. Existe gran cantidad de este tipo de lenguajes, Python, Perl, Bash, por nombrar algunos. Un ejemplo del programa *Hola mundo* escrito en Python es dado a continuación:

```
# Programa Hola mundo
print "Hola Mundo"
```

4.2.5. Lenguajes especializados.

Desde el punto de vista de la funcionalidad de los lenguajes podemos separarlos en lenguajes de carácter general y lenguajes especializados. Los lenguajes de carácter general son aquellos que sirven para programar una gran número de problemas, por ejemplo C o C++, Python.

Los lenguajes especializados han sido diseñados para realizar tareas específicas. Ejemplos de ello son PHP y JavaScript, especializados en crear páginas web, o SQL, creado para manipular información en bases de datos.

Una lista de lenguajes.

A continuación, damos una lista, probablemente muy incompleta, de los lenguajes de programación más comunes en la actualidad:

ABC, Ada, ASP, Awk, BASIC, C, **C++**, C#, Caml, Cobol, código de máquina, Corba, Delphi, Eiffel, Erlang, Fortran, Haskell, Java, JavaScript, Lisp, Logo, Modula, Modula 2, Mozart, Mumps, Oberon, Objetive C, Oz, Pascal, Perl, PHP, **Python**, Realbasic, Rebol, Rexx, RPG, Ruby, Scheme, Smaltalk, SQL, Squeak, TCL, Visual Basic.

4.3. Lenguajes naturales y formales.

4.3.1. Lenguajes naturales.

Son lenguajes hablados por la gente (por ejemplo: Español, Inglés, Alemán o Japonés). Una de sus caracteristicas es que son ambiguos, por ejemplo: "Dame esa cosa" o "¡Oh, seguro, Grande!". En ambos ejemplos no es claro a que se están refiriendo y se necesita un contexto para entenderlos. Muchas veces estos lenguajes son redundantes y están llenos de expresiones idiomáticas las cuales no deben ser tomadas literalmente, por ejemplo: "Me podría comer una vaca", "Me mataste", o "Ándate a la punta del cerro".

4.3.2. Lenguajes formales.

Hecho por el hombre, como las matemáticas, la notación en química o los lenguajes de programación de computadores. Se caracterizan por ser inambiguos. Por ejemplo, una expresión matemática: $1 + 4 = 5$; o una expresión en química: $CH_4 + 2O_2 \rightarrow 2H_2O + CO_2$; o, finalmente, una expresión en lenguaje de programación `print "Hola mundo"`. Los lenguajes formales son además concisos y estrictamente literales.

Sintaxis.

Los lenguajes, tanto naturales como formales, tienen reglas de sintaxis. Por una parte, están los *tokens*, que corresponden a los elementos básicos (*i.e.* letras, palabras, símbolos) del lenguaje:

- Tokens correctos: {1,3,4,+,=};{negro,gato,el}; {CH_4,O_2,H_2O,CO_2}.
- Tokens incorrectos: {@,#,&,?}; {grneo,gt,l;}; {C^H,2_O,$_2HO$}

Por otro lado, tenemos las estructuras, esto es la manera en que los *tokens* son organizados:

- Estructuras correctas: $1 + 3 = 4$, el gato negro, $CH_4 + 2O_2 \rightarrow 2H_2O + CO_2$
- Estructuras incorrectas: $13 + = 4$, negro gato el, $\rightarrow CH_4\ 2O_2\ ++ 2H_2O\ CO_2$

4.4. Desarrollando programas.

Para desarrollar sus primeros programas parta escribiendo en sus propias palabras lo que el programa debería hacer. Convierta esta descripción en una serie de pasos en sus propias palabras. Para cada uno de los pasos propuestos traduzca sus palabras en un código (Python o C++). Dentro del código incluya instrucciones que impriman los valor de las variables para probar que el programa está haciendo lo que usted esperaba.

4.5. La interfaz con el usuario.

Siempre que escriba un programa debe tener presente que alguien, que puede no ser usted mismo, lo puede usar alguna vez. Lo anterior significa, en particular, que el programa debe tener documentación, ya que un programa sin documentación es muy difícil de usar. Pero además es importante cuidar la parte del programa con la que el usuario interactúa, es decir la *interfaz con el usuario*. Esta interfaz podrían ser tanto mensajes simples de texto como sofisticadas ventanas gráficas. Lo importante es que ayuden al usuario a ejecutar correctamente el programa.

Revisemos una mala interfaz con el usuario. Tenemos un programa que no sabemos lo que hace, pero al ejecutarse resulta lo siguiente:

```
Entre un numero 5
Entre otro numero 7
La respuesta es 12
```

Hay una evidente falta de instrucciones de parte del programador para el usuario, que primero no sabe para qué se le pide cada número, y luego no sabe qué hizo con ellos, sólo la respuesta, 12, sin mayor explicación de lo que significa.

Como contraparte, una buena interfaz con el usuario tiene documentación anexa, o bien, alguna ayuda en el mismo programa. Esta documentación debiera explicar que hace el programa, los datos que necesitará y el o los resultados que entregará cuando finalice.

Cada vez que se le pide algo al usuario deberían estar claras las siguientes preguntas: ¿qué es exactamente lo que se supone que yo tipee?; ¿los números que ingreso deben tener decimales?; ¿o deben ser sin decimales?; ¿los números que ingreso son positivos o negativos?; ¿los números que se piden son grandes o son números pequeños?; ¿en qué unidades de medidas debo ingresarlos? Si se trata de palabras, ¿debo ingresarlas en minúsculas o mayúsculas?

Algunos lineamientos básicos que debería observar para construir interfaces con el usuario que sea claras son los siguientes:

- Parta con un título e indicaciones dentro del programa.
- Cuando pregunte por un dato que quiere que el usuario ingrese, dele la ayuda necesaria, por ejemplo

```
Entre el largo en metros (0-100):
```

- Que las preguntas tengan sentido.
- Use espacios y caracteres especiales para mantener la pantalla despejada.
- Indíquele al usuario que el programa terminó.

Una versión mejorada del programa anterior podría ser la siguiente:

```
Calculo de la suma de dos numeros
Ingrese un numero entero: 5
Ingrese otro numero entero: 7
La suma es 12
```

4.6. Sacar los errores de un programa.

Los errores en un programa son llamados *bugs*. Al proceso de rastrear los errores y corregirlos se le conoce como *debugging*. Un programa especializado en hacer *debugging* es llamado *debugger*. **El debugging es una de las más importantes habilidades en programación.** Los tres principales tipos de errores o *bugs* y sus consecuencias para la ejecución del programa son:

1. Errores de sintaxis
   - Usar un *token* incorrecto o usar *token* correctos pero estructurarlos en forma incorrecta.
   - Caso compilado, no genera el ejecutable.
   - Caso interpretado, el programa termina abruptamente con un mensaje de error.
2. Errores de ejecución (*run-time error*)
   - Errores que ocurren durante la ejecución.
   - El programa deja de correr abruptamente.

3. Errores lógicos
    - Errores en cómo el programa está lógicamente construido.
    - El programa corre, pero hace las cosas mal.

Capítulo 5

Una breve introducción a Python.

versión 3.03, 27 de octubre de 2010

En este capítulo se intentará dar los elementos más básicos del lenguaje de programación Python.

Se debe consignar que no se consideran todas las posibilidades del lenguaje y las explicaciones están reducidas al mínimo.

5.1. Python.

El lenguaje de programación Python fue creado alrededor de 1990 por el científico en computación holandés Guido van Rossem y su nombre es un tributo a la grupo cómico *Monty Python* del cual Guido es admirador. Es un lenguaje de interpretado o de *script* multiplataforma con una sintaxis muy clara y orientado a objetos, que favorece crear código facilmente legible y reutilizable. El sitio oficial del lenguage en la *web* es `http://www.python.org`.

El programa Python (como programa, no como lenguaje) posee un ambiente interactivo que nos permite ejecutar instrucciones del lenguaje Python directamente. Para ello, basta dar el comando:

```
username@host:~$ python
Python 2.6.6 (r266:84292, Sep 15 2010, 16:00:36)
[GCC 4.4.5 20100909 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

El programa ofrece un *prompt* (`>>>`), esperando instrucciones del usuario. Las instrucciones son interpretadas y ejecutadas de inmediato. Esta forma de usar Python tiene la ventaja de la retroalimentación inmediata; de inmediato el programador sabe si la instrucción está correcta o incorrecta. Sin embargo, tiene la desventaja de que el código no es guardado, y no puede por tanto ser reutilizado.

Por otra parte, cuando escribimos un archivo de instrucciones en Python (*script*), tenemos la ventaja de que el código sí es almacenado, pudiendo ser reutilizado. En este caso las

desventajas son que la retroalimentación no es inmediata y que habitualmente requiere más *debugging* para que el código funcione correctamente.

5.2. El Zen de Python.

Los usuarios de Python se refieren a menudo a la Filosofía de Python. Estos principios o filosofía fueron descritos por el desarrollador de Python, Tim Peters, en El Zen de Python:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Ralo es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales para romper las reglas.
- Aunque lo práctico gana a la pureza.
- Los errores nunca deben pasar silenciosamente.
- A menos que esté silenciado explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una –y preferiblemente sólo una– manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia en principio, a menos que usted sea holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ahora mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (*namespaces*) son una gran idea ¡Hagamos más de esas cosas!

Los códigos que que están escrito bajo estos principios, de legibilidad y transparencia, se dice que son "pythonicos". Contrariamente, códigos opacos u ofuscados son bautizados como "no pythonicos" ("unpythonics" en inglés).

5.3. El primer programa.

El primer programa que escribiremos en Python es, el clásico, `hola mundo`. Para crear este *script* en Python requerimos de un editor (`vi`, `jed`, `xemacs`, `gedit`... elija su favorito). Escribimos en un archivo:

```
print "Hola mundo"
```

Para poder identificarlo rápidamente, más adelante, grabemoslo con extensión `.py`. Supongamos que lo grabamos con en el nombre `hola.py`. Ahora, ejecutamos el programa dando el comando

```
username@host:~$ python hola.py
```

Agreguemosle una segunda línea, para que el programa quede esperando un `enter` antes de terminar.

```
print "Hola mundo"
raw_input()
```

Nuevamente ejecutamos el programa dando el comando

```
username@host:~$ python hola.py
```

Para que el programa se ejecute con el intérprete adecuado, Python en este caso, es necesario añadir una nueva línea al principio del archivo:

```
#!/usr/bin/python
print "Hola Mundo"
raw_input()
```

A esta línea se le conoce como *hashbang*, *sharpbang* o *shebang*. El par de caracteres `#!` indica al sistema operativo que el *script*, contenido en las siguientes líneas de ese archivo, se debe ejecutar utilizando el intérprete especificado a continuación.

Otra opción es utilizar el programa `env` para preguntar al sistema por el *path* específico del intérprete que nos interesa, en este caso Python,

```
#!/usr/bin/env python
print "Hola Mundo"
raw_input()
```

Recuerde que además de añadir el *shebang*, debe darle los permisos de ejecución adecuados (`chmod u+x archivo.py`). Finalmente, para ejecutarlo basta ubicarse en el directorio donde está el archivo y dar el comando

```
jrogan@manque:~/InProgress/python$ ./hola.py
```

5.4. Tipos básicos.

Los tipos básicos del lenguaje son

- **Números**,
  - **Enteros**: Los números que pertenecen al conjunto $\mathbb{Z}$, es decir, sin decimales. Son números entre $-2^{31} = -2147483648$ y $2^{31} - 1 = 2147483647$ en una máquina de 32 bits y entre $-2^{63} = -9223372036854775808$ y $2^{63} - 1 = 9223372036854775807$ en una máquina de 64 bits. Para conocer estos límites se carga el modulo `sys` en Python interactivo, con el comando `import sys`, y luego se dan los comandos `-sys.maxint-1` y `sys.maxint`.
  - **Enteros largos**: Números enteros de precisión arbitraria, estando sólo limitados por la cantidad de memoria disponible en la máquina. Los distinguimos por una `L` al final del número, por ejemplo: `23434235234L`.
  - **Con punto flotante**: Los números que pertenecen al conjunto $\mathbb{R}$, pero con un número finito de decimales. Los valores van desde $\pm 2.2250738585072020 \times 10^{-308}$ hasta $\pm 1.7976931348623157 \times 10^{308}$. Para conocer estos límites se carga el modulo `sys` en Python interactivo, con el comando `import sys`, y luego se da el comando `sys.float_info`.
  - **Complejos**: Son números que pertenecen al conjunto $\mathbb{C}$, donde la parte real y la parte imaginaria son números con punto flotante. Se pueden escribir como *real*+*imag*`j` o bien como *real*+*imag*`J`.
- **Cadenas de caracteres** (*strings*): Usualmente un conjunto de caracteres, *i.e.* un texto: "Hola mundo". Están delimitados por comillas simples o dobles.
- **Booleanos**: Un tipo que sólo puede tomar dos valores True(verdadero) o False(falso).

Tipo	Descripción	Ejemplo
int	entero	117
long	entero largos	23434235234L
float	número con punto flotante	1.78
complex	número complejo	0.5 +2.0j
str	*string*	"abc"
bool	booleano	True o False

Cuadro 5.1: Los tipos básicos del lenguaje Python.

5.4.1. Las variables.

Las variable son un nombre, usado dentro del programa, para referirse a un objeto o valor. Las limitaciones y consideraciones que hay que tener en cuenta para darle nombre a una variable son:

- No puede ser una palabra reservada del lenguaje, *i.e.*

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

- No puede comenzar por un número.
- Las letras mayusculas y las minúsculas en los nombres se consideran diferentes.
- No puede incluir caracteres ilegales tales como `$,%,+,=`, etc.

5.4.2. Asignación de variables.

Para asignarle o almacenar un valor a una variable, por ejemplo en una variable llamada `num`, basta poner el nombre de la variable a la izquierda un signo igual y al lado derecho el valor o expresión que queremos asignarle

```
num=8                # Entero
k=23434235234L       # Entero largo
pi=3.14              # Punto flotante
z=1.5+0.5j           # Complejo
saludo=¨Hola mundo¨  # String
afirmacion=True      # Booleano
# Asignación de una operación
num=pi*3.0**2
```

Un mismo valor puede ser asignado a varias variables simultaneamente

```
>>> x=y=z=0      # Todas las variables valen cero
>>> print x,y,z  # Muestra en pantalla el valor de las variables
0 0 0
```

O bien, podemos hacer asignasiones de diferentes valores a diferentes variables en una misma asignación

```
>>> a,b,c=0,1,2
>>> print a,b,c
0 1 2
>>> a,b=b,a+b
>>> print a,b
1 1
```

Consejo: cree variables cuyos nombres signifiquen algo:

```
ddm=31 # MAL
dias_del_mes=31 # BIEN
```

5.4.3. Reciclando variables.

Una vez que una variable es creada su valor puede ser reasignado. Veamos un ejemplo donde la variable `card_value` es reutilizada

```
card1, card2, card3= 1,2,3
card_value=card1+card2
print card_value
card_value=card1+card2+card3
print card_value
```

5.4.4. Comentarios.

Los comentarios son anotaciones que usted escribe para ayudar a explicar lo que está haciendo el programa. Los comentarios comienzan con el caracter `#`. Lo escrito después de `#`, hasta el final de la línea, es ignorado por el intérprete. Por ejemplo:

```
dias = 60 #disponibles para el proyecto
```

Naturalmente, los comentarios no son muy útiles cuando se trabaja interactivamente con Python, pero sí lo son cuando se escribe un *script*. De este modo se pueden insertar explicaciones en el código que ayuden a recordar qué hace un programa en cada una de sus secciones, o explicarlo a terceros.

Consejo: Es buena costumbre de programación que las primeras líneas de un código sean comentarios que incluyan el nombre del programador y una breve descripción del programa.

5.4.5. Operaciones matemáticas.

Con Python podemos realizar las operaciones básicas: suma ($+$), resta ($-$), multiplicación ($*$) y división ($/$). Operaciones menos básicas también están disponibles: el exponente ($**$), la división entera ($//$) o el módulo ($\%$).

Entre las operaciones hay un orden de precedencia, unas se realizarán primero que otras. A continuación, damos el orden de precedencia, partiendo por lo que se hace primero:

- Paréntesis, exponentes, división y multiplicación.
- Suma y resta.
- De izquierda a derecha.

Como ejemplo de la importancia de saber el orden de precedencia veamos los siguiente ejemplos:

$$2 * (3 - 1) = 4 \qquad \text{y} \qquad 2 * 3 - 1 = 5$$

5.4.6. Cadenas de caracteres (*strings*).

Una cadena de caracteres debe estar entre apóstrofes o comillas simples o dobles. Por ejemplo:

- `nombre = "Este es tu nombre"`
- `nombre2= ‘Este es tambien su nombre’`

Si una cadena de caracteres necesita un apóstrofe dentro de ella, anteponga un \ al apóstrofe extra. Ejemplos:

- `titulo = "Ella dijo: \"Te amo\""`
- `titulo2 = ‘I\’m a boy’`

Algunas cadenas de caracteres con significado especial empiezan con el caracter \ (*String Backslash Characters*).

- `\\` = Incluya \.
- `\’` = Apóstrofe simple.
- `\"` = Apóstrofe doble.
- `\n` = Cambio de línea.

En una cadena o *strings* con triples comillas (simples o dobles) podemos escribir un texto en varias líneas:

```
triple = """Esta es la primera linea
            y esta.. es la segunda"""
```

al imprimir el anterior *strings* se respetará el salto de línea.

Un string puede estar precedido por caracteres, si el carácter es `u` indican que se trata de un *string* codificado en `Unicode`,[1] por ejemplo

```
unicode = u"äóè"
```

si el carácter es `r` indica que se trata de un *string raw* (del inglés, crudo). Los *string raw* son aquellos en que los caracteres que comienzan con (\) no se sustituyen, por ejemplo

```
raw = r"\n"
```

5.4.7. Operaciones simples con *strings*.

Dos de las operaciones más comunes con *strings*:

- Concatenación: se pueden concatenar dos *strings* al sumarlos, veamos un ejemplo:

```
>>> x = "Hola"
>>> y = "Mundo"
>>> print x+y
>>> HolaMundo
```

- Repetición:

```
>>> z = "Ja"
>>> print z*3
>>> JaJaJa
```

5.4.8. Los caracteres dentro de un *strings*.

Los *strings* son hechos de pequeñas unidades, cada caracter individual. Cada uno de los caracteres tiene una dirección numérica dentro del *string*, donde el primer caracter tiene la dirección cero (0). Cada caracter individual, o conjunto de caracteres, en un *string* puede ser acccesado usando sus direcciones numéricas. Use `[ ]` para accesar caracteres dentro de un *string*. Veamos un ejemplo

```
palabra = "computador"
letra = palabra[0]
```

Para acceder un conjunto de caracteres dentro de un *string* lo podemos hacer como sigue:

- Use `[#:#]` para obtener un conjunto de letras.
  `parte = palabra[1:3]`

[1] En Python 3.x los `string` pasa a ser `Unicode` luego `cadena="abc"` es lo mismo que `cadena=u"abc"`

- Para tomar desde el comienzo a un punto dado en el *string*.
  `parte = palabra[:4]`
- Para tomar desde un punto dado al final del *string*.
  `parte = palabra[3:]`

5.4.9. Índices negativos.

Veamos que pasa cuando usamos índices negativos

```
>>> a="hola"
>>> a[0]
'h'
>>> a[-1]
'a'
>>> a[-2]
'l'
>>> a[-3]
'o'
>>> a[-4]
'h'
>>> a[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

5.4.10. Booleanos.

Una variable booleana sólo puede tomar dos valores: True (cierto) o False (falso). Ellas son el resultado de comparaciones entre valores. Supondremos, para los ejemplos siguientes, que la variable `a=10` y la variable `b=20`.

El operador `==`, evalúa si los valores de los dos operandos son o no iguales, si son iguales la condición es `True` si no lo son es `False`. En nuestro caso, `(a == b)` es `False`.

El operador `!=`, evalúa si los valores de los dos operandos son distintos o son iguales, si son distintos la condición es `True` si no lo son, es decir son iguales, es `False`. En nuestro caso, `(a != b)` es `True`.

El operador `<>`, evalúa si los valores de los dos operandos son distintos o son iguales, si son distintos la condición es `True` si no lo son, es decir son iguales, es `False`. En nuestro caso, `(a <> b)` es `True`.

El operador `>`, evalúa si el valor del operando izquierdo es mayor que el valor del operando derecho. Si la respuesta es si entonces la condición es `True` si la respuesta es no entonces la condición es `False`. En nuestro caso, `(a > b)` es `False`.

El operador <, evalúa si el valor del operando izquierdo es menor que el valor del operando derecho. Si la respuesta es si entonces la condición es `True` si la respuesta es no entonces la condición es `False`. En nuestro caso, `(a < b)` es `True`.

El operador >=, evalúa si el valor del operando izquierdo es mayor o igual que el valor del operando derecho. Si la respuesta es si entonces la condición es `True` si la respuesta es no entonces la condición es `False`. En nuestro caso, `(a >= b)` es `False`.

El operador <=, evalúa si el valor del operando izquierdo es menor o igual que el valor del operando derecho. Si la respuesta es si entonces la condición es `True` si la respuesta es no entonces la condición es `False`. En nuestro caso, `(a <= b)` es `True`.

Los condicionales pueden ser unidos usando las palabras reservadas `and`, `or` o `not`. Si ocupamos un `and` para unir dos condiciones lógicas tenemos que ambas condiciones deben satisfacerse para que el condicional sea cierto. En el caso de ocupar `or` para unir dos condiciones lógicas una de ellas debe ser satisfecha para que el condicional sea cierto. Finalmente el `not` se antepone a una condición y la niega, es decir, será cierto si la condición no es satisfecha. En todos los caso se aplica que `False`$== 0$ y `True`$== 1$ (en realidad `True`$! = 0$).

5.5. Imprimiendo e ingresando.

La intrucción `print`[2] evalúa la expresión a la derecha y escribe el objeto resultante al *standard output*. Si el objeto no es un *string*, primero lo convierte en *string* usando las reglas de conversión y luego lo escribe en el *standard output*. Algunos ejemplos simples,

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

>>> print "Hola", "mundo"
Hola mundo
>>> print (1, 2)
(1, 2)
```

5.5.1. Imprimiendo en la misma línea.

Agregando una coma (`,`) al final de una instrucción `print` hará que el próximo comando `print` aparezca en la misma línea. Ejemplo

```
print num1,"+", num2, "=",
print respuesta
```

[2]En Python 3.x `print` deja de ser una instrucción y pasa a ser una función, `print("Hello world!")`

5.5.2. Imprimiendo un texto de varias líneas.

Si queremos imprimir un texto que tenga varias líneas podemos usar dos formas distintas de la función `print` usando el caracter `\n` o bien usando un texto entre triple comilla

```
>>> print "primera linea\nsegunda linea"
primera linea
segunda linea
>>> print """primera linea
... segunda linea"""
primera linea
segunda linea
```

5.5.3. Composición.

Se pueden combinar sentencias simples en una compuesta, a través del operador ",":

```
>>> x = "Elizabeth"
>>> print "Tu nombre es : ", x
>>> Tu nombre es : Elizabeth
```

En el ejemplo, `x` fue asignado explícitamente a una variable, pero naturalmente cualquier tipo de asignación es posible, por ejemplo:

```
>>> promedio=(nota+extra_creditos)/posibles
>>> print "Tu promedio es : ", promedio
```

5.5.4. Imprimiendo con formato

El siguiente comando ilustra como se puede imprimir con formato

```
>>> entero = 15       # Int
>>> real = 3.14159    # Float
>>> cadena = "Hola"   # String
>>> print "|%4d, %6.4f,%5s" % (entero,real,cadena)  # string % (tupla)
|  15, 3.1416, Hola
```

5.5.5. Entrada (input).

Para leer *strings* del *stdin* use la instrucción `raw_input()`, por ejemplo

```
nombre = raw_input("Cual es tu nombre?")
```

Si necesita leer números del *stdin* use la instrucción `input()`:

```
numero=input("Cuantos?")
```

En ambos casos, el mensaje entre comillas dentro de los paréntesis es opcional, sin embargo, aclara al usuario lo que el programa le está solicitando. En el siguiente par de ejemplos, el programa solicita información al usuario, salvo que en el primero, el programa queda esperando una respuesta del usuario, quien, a menos que sepa de antemano qué quiere el programa, no tiene modo de saber por qué el programa no continúa ejecutándose.

Ejemplo sin mensaje (queda esperando para siempre una respuesta):

```
>>> nombre = raw_input()
```

Ejemplo con mensaje:

```
>>> nombre = raw_input("Cual es tu nombre?")
Cual es tu nombre? Pedro
>>>
```

5.6. Tipos avanzados, contenedores.

Los contenedores del lenguaje son

- **Listas**: las listas son colecciones ordenadas de elementos o ítems (*strings*, números o incluso otras listas). Las listas están encerradas entre paréntesis `[ ]`. Cada ítem en una lista está separado por una coma.
- **Tuplas**: las tuplas son colecciones ordenadas e inmutable de elementos o ítems (*strings*, números o incluso otras tuplas). Las tuplas están encerradas entre paréntesis `( )`. Cada ítem en una tupla está separado por una coma. Una tupla es una lista inmutable.
- **Conjuntos**: los conjuntos se construyen mediante set(items) donde ítems es cualquier objeto iterable, como listas o tuplas.
- **Diccionarios**: un diccionario es una colección de ítems que tiene una llave y un valor. Los diccionarios están encerrados entre paréntesis de llave `{ }`. Cada elemento está separado por una coma y cada elemento está compuesto por un par `llave:valor`

5.6.1. Listas.

Veamos ejemplos de listas

```
mascotas = ["perros", "gatos", "canarios", "elefantes"]
numeros = [1,2,3,4,5,6]
cosas = [ 1, 15, "gorila", 23.9, "alfabeto"]
```

Tipo	Descripción	Ejemplo
list	listas	[1, 'hum', 2.0]
tuple	tuplas	(1, 'hum', 2.0)
set	conjuntos	set([1, 'hum', 2.0])
dict	diccionario	{'a':7.0, 23: True}

Cuadro 5.2: Los tipos contenedores del lenguaje Python.

Un elemento de una lista puede ser otra lista. Una lista dentro de otra lista es llamada *lista anidada*. A continuación un ejemplo de listas anidadas

```
para_hacer = ["limpiar", ["comida perro", "comida gato","comida pez"], "cena"]
```

5.6.2. Rebanando listas.

Una lista puede ser accesada al igual que un *string* usando el operador `[ ]`. Para acceder a un valor de la lista uno debe saber su índice de posición. Ejemplo

```
>>> lista=["Pedro", "Andres", "Jaime", "Juan"]
>>> print lista[0]
Pedro
>>> print lista[1:]
['Andres', 'Jaime', 'Juan']
```

Si uno remueve un item desde la lista, el índice puede cambiar por el de otro ítem en la lista.

Para accesar un item en una lista anidada hay que proveer dos índices. Ejemplo

```
>>> lista_palabras = ["perro", ["fluffy", "mancha", "toto"], "gato"]
>>> print lista_palabras[1][2]
toto
```

5.6.3. Mutabilidad.

A diferencia de los *strings* las listas son mutables, lo que significa que se pueden cambiar. Ejemplo

```
>>> string = "perro"
>>> string [2] = "d" # Esta NO es un instruccion VALIDA
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

En cambio en una lista

```
>>> lista = ["p", "e", "r", "r", "o"]
>>> lista [2] = "d"
>>> print lista
['p', 'e', 'd', 'r', 'o']
```

5.6.4. Modificando listas.

Como se muestra en la comparación anterior una lista puede ser cambiada usando el operador `[ ]`. Ejemplo

```
>>> lista=["Pedro", "Andres", "Jaime", "Juan"]
>>> lista[0]="Matias"
>>> print lista
['Matias', 'Andres', 'Jaime', 'Juan']
>>> lista[1:2]=["perro","gato"]
>>> print lista
['Matias', 'perro', 'gato', 'Jaime', 'Juan']
```

5.6.5. Agregando a una lista.

Para agregar items al final de una lista use `list.append(item)`. Ejemplo

```
>>> lista=["Pedro", "Andres", "Jaime", "Juan"]
>>> lista.append("Matias")
>>> print lista
['Pedro', 'Andres', 'Jaime', 'Juan', 'Matias']
```

Notemos que las operaciones que modifican la lista la modificarán de manera tal que si multiples variables apuntan a la misma lista todas las variables cambiarán al mismo tiempo.

```
>>> L=[]
>>> M=L
>>> # modifica ambas listas
>>> L.append(obj)
```

Para crear una lista separada se puede usar el "rebanado" o la función `list` para crear una copia.

```
>>> L=[]
>>> M=L[:]          # creando una copia
>>> N=list(L)       # crea otra copia
>>> # modifica solo a L
>>> L.append(obj)
```

5.6.6. Borrando items de una lista.

Use el comando `del` para remover items basado en el índice de posición. Ejemplo en forma interactivo

```
>>> lista=["Pedro", "Andres", "Jaime", "Juan"]
>>> del lista[1]
>>> print lista
['Pedro', 'Jaime', 'Juan']
```

Para remover items desde una lista sin usar el índice de posición, use el siguiente comando `nombre_lista.remove("item")` que borra la primera aparición del item en la lista. Un ejemplo interactivo

```
>>> jovenes = ["Pancho", "Sole", "Jimmy", "Pancho"]
>>> jovenes.remove("Pancho")
>>> print jovenes
['Sole', 'Jimmy', 'Pancho']
```

5.6.7. Operaciones con listas.

Las listas se pueden sumar resultando una sola lista que incluya ambas listas iniciales. Además, podemos multiplicar una lista por un entero n obteniendo una nueva lista con n réplicas de la lista inicial. Veamos ejemplos de ambas operaciones en forma interactiva

```
>>> lista1=["Pedro", "Andres", "Jaime", "Juan"]
>>> lista2=["gato", 2]
>>> print lista1+lista2
['Pedro', 'Andres', 'Jaime', 'Juan', 'gato', 2]
>>> print lista2*2
['gato', 2, 'gato', 2]
>>> print 2*lista2
['gato', 2, 'gato', 2]
```

La sentencia `list_name.sort()` pone la lista en orden alfabético y/o numérico.

5.6.8. Tuplas.

Una tupla no puede modificarse de ningún modo después de su creación.

```
>>> t = ("a", "b", 8)
>>> t[0]
'a'
```

Una tupla se define del mismo modo que una lista, salvo que el conjunto se encierra entre paréntesis `( )`, en lugar de entre corchetes `[ ]`. Los elementos de la tupla tienen un orden definido, como los de la lista. Las tuplas tienen primer índice 0, como las listas, de modo que el primer elemento de una tupla `t`, no vacía es siempre `t[0]`. Los índices negativos cuentan desde el final de la tupla, como en las listas. Las porciones funcionan como en las listas. Advierta que al extraer una porción de una lista, se obtiene una nueva lista; al extraerla de una tupla, se obtiene una nueva tupla. No hay métodos asociados a tuplas (tal como `append()` en una lista).

No pueden añadirse elementos a una tupla, no pueden eliminarse elementos de una tupla, no pueden buscarse elementos en una tupla, se puede usar `in` para ver si un elemento existe en la tupla.

Las tuplas son más rápidas que las listas. Si está definiendo un conjunto constante de valores y todo lo que va ha hacer con él es recorrerlo, utilice una tupla en lugar de una lista. Una tupla puede utilizarse como clave en un diccionario, pero las listas no. Las tuplas pueden convertirse en listas y vice versa. La función incorporada `tuple(lista)` toma una lista y devuelve una tupla con los mismos elementos. La función `list(tupla)` toma una tupla y devuelve una lista.

5.6.9. Conjuntos

Los conjuntos no mantienen el orden ni contienen elementos duplicados. Se suelen utilizar para eliminar duplicados de una secuencia, o para operaciones matemáticas como intersección, unión diferencia y diferencia simétrica. Existe también una versión inmutable de los conjuntos. Veamos algunos ejemplos:

```
>>> conjuntoA = set(["primero", "segundo", "tercero", "primero"])
>>> conjuntoA
set(['tercero', 'segundo', 'primero'])
>>> conjuntoB = set(["segundo", "cuarto"])
>>> conjuntoB
set(['cuarto', 'segundo'])
>>> conjuntoA & conjuntoB                          # Intersección
set(['segundo'])
>>> conjuntoA | conjuntoB                          # Unión
set(['cuarto', 'primero', 'segundo', 'tercero'])
>>> conjuntoA - conjuntoB                          # Diferencia A-B
set(['primero', 'tercero'])
>>> conjuntoB - conjuntoA                          # Diferencia B-A
set(['cuarto'])
>>> conjuntoA ^ conjuntoB                          # Diferencia simétrica
set(['cuarto', 'primero', 'tercero'])
conjunto_inmutable = frozenset(["a", "b", "a"])     # No es mutable
```

5.6.10. Diccionarios.

Los diccionarios son parecidos a las listas, excepto que en vez de tener asignado un índice uno crea los índices.

```
lista = ["primero", "segundo", "tercero"]
diccionario = {0:"primero", 1:"segundo", 2:"tercero"}
```

Para crear un diccionario debemos encerrar los item entre paréntesis de llave `{}`. Debemos proveer una llave y un valor, un signo : se ubica entre la llave y el valor `(llave:valor)`. cada llave debe ser única. Cada par `llave:valor` está separado por una coma. Veamos un par de ejemplos con diccionarios

```
ingles = {'one':'uno', 'two':'dos'}
```

Uno en japonés

```
nihon_go = {}
nihon_go["ichi"] = "uno"
nihon_go["ni"] = "dos"
nihon_go["san"] = "tres"
print nihon_go
{ 'ichi':'uno', 'ni':'dos', 'san':'tres'}
```

Para acceder el valor de un ítem de un diccionario uno debe entrar la llave. Los diccionarios sólo trabajan en una dirección. Uno debe dar la llave y le devolverán el valor. Uno no puede dar el valor y que le devuelvan la llave. Ejemplo

```
nihon_go = { 'ichi':'uno', 'ni':'dos', 'san':'tres'}
print nihon_go['ichi']
uno
```

Notemos que este diccionario traduce del japonés al español pero no del español al japonés.

5.6.11. Editando un diccionario.

- Para cambiar un valor de un par, simplemente reasígnelo
  `nihon_go["ichi"]=1`
- Para agregar un par `llave:valor`, éntrelo
  `nihon_go["shi"]=cuatro`
- Para remover un par use `del`
  `del nihon_go["ichi"]`

- Para ver si una llave ya existe, use la función `has_key()`
  `nihon_go.has_key("ichi")`

- Para copiar el diccionario entero use la función o método `copy()`.
  `japones= nihon_go.copy()`

Los diccionarios son mutables. Uno no tiene que reasignar el diccionario para hacer cambios en él.

Los diccionarios son útiles cada vez que usted tiene items que desea ligar juntos. También son útiles haciendo sustituciones (reemplace todos los x por y). Almacenando resultados para una inspección rápida. Haciendo menús para programas. Creando mini bases de datos de información.

5.7. Control de flujo.

En esta sección veremos los condicionales y los ciclos o bucles.

5.7.1. Condicionales.

Los condicionales son expresiones que puede ser ciertas o falsas. Por ejemplo, ¿el usuario tipeó la palabra correcta? o ¿El número es mayor que 10? El resultado de la condición decide que sucederá, por ejemplo, a todos los números mayores que 100 réstele 20, cuando la palabra ingresada sea la correcta, imprima "¡Bien!"

5.7.2. Posibles condicionales.

- `x == y    # x es igual a y.`
- `x != y    # x no es igual a y.`
- `x <> y    # x no es igual a y.`
- `x >  y    # x es mayor que y.`
- `x <  y    # x es menor que y.`
- `x >= y    # x es mayor igual a y.`
- `x <= y    # x es menor igual a y.`

A continuación, algunos ejemplos de los anteriores condicionales:

- `x == 125`

- `passwd == "nix"`
- `num >= 0`
- `letter > "L"`
- `num/2 == (num1-num)`
- `num%5 != 0`

5.7.3. Comparando *strings*.

Los *strings* también pueden ser usados en comparaciones. De acuerdo a Python, todas las letras minúsculas son mayores que las letras mayúsculas Así `"a">"Z"`.

Una buena idea es convertir todos los *strings* a mayúscula o minúscula, según sea el caso, antes de hacer comparaciones. Recordemos que el módulo `string` contiene varias funciones útiles incluyendo: `lower(string)` y `upper(string)`. Revise la documentación.

5.7.4. El `if`.

A continuación, estudiemos la instrucción `if`, partamos de la forma general de la instrucción:

```
if condicion:
   instrucciones
```

Primero la palabra clave `if`, luego la condición `condicion`, que puede ser algo como `x<y` o `x==y`, etc. La línea termina con : requerido por la sintaxis del `if`. En las líneas siguientes `instrucciones`, viene las instrucciones a seguir si la condición es cierta. Estas instrucciones deben ir con sangría (*indent*).

Un ejemplo de una construcción `if` simple.

```
num = input("Entre su edad")
if num >= 21:
       print "Persona mayor de edad"
       print                        #linea en blanco
       print "Gracias"
```

5.7.5. El `if...else`.

La forma general de la construcción `if...else` a continuación:

```
if condicion:
   instrucciones_1
else:
   instrucciones_2
```

El `else` debe de estar después de una prueba condicional. Sólo se ejecutará cuando la condición evaluada en el `if` sea falsa. Use esta construcción cuando tenga dos conjuntos diferentes de instrucciones a realizar dependiendo de la condición. Un ejemplo

```
x= input("Ingrese un numero: ")
if x%2 == 0:
   print "el numero es par"
else:
   print "el numero es impar"
```

5.7.6. Forma compacta del `if...else`.

Existe una forma compacta de expresar un `if else`. En esta construcción se devuelve `A` si al evaluar la condición `COND` está resulta cierta, si no se cumple se devuelve `B`, es decir, `A if COND else B`. Veamos un ejemplo:

```
num = input("Ingrese un número entero: ")
paridad = "par" if (num % 2 == 0) else "impar"
print paridad
```

5.7.7. El `if...elif...else`.

La forma general de la construcción `if...elif...else`, a continuación:

```
if condicion_1:
   instruciones_1
elif condicion_2:
   instruciones_2
else:
   instruciones_3
```

Para más de dos opciones use la construcción con `elif`. `elif` es la forma acortada de las palabras *else if*. Las instrucciones asociadas a la opción `else` se ejecutarán si todas las otras fallan. Un ejemplo concreto:

```
x=input(''Ingrese un numero : '')
if x<0 :
```

```
    print x," es negativo"
elif x==0 :
    print x," es cero"
else:
    print x," es positivo"
```

Consejo: Los `if` pueden ser anidados. Sea cuidadoso, ya que la anidación puede producir confusión y debería ser usada con moderación y mesura. Recuerde: *Plano es mejor que anidado.*

5.7.8. La palabra clave `pass`.

El comando `pass` no realiza acción alguna, es decir, no hace nada. Un ejemplo

```
if x<0:
   haga_algo()
else:
   pass # no hace nada
```

5.7.9. Operadores lógicos.

A continuación, algunos ejemplos de operadores lógicos:

- `if x>0 and x<10:`
- `if y>0 and x>0:`
- `if pwd=="codigo"  or pwd=="fuente":`
- `if y>0 or x<0:`
- `if not(x<y):`
- `if x>y or not(x<0):`

5.7.10. Forma alternativa, de hacer una pregunta compuesta.

Cuando pruebe valores para < o >, estas pruebas pueden ser escritas como un sólo condicional sin usar el `and`. Veamos ejemplos

```
if 0<x<100:

if 1000 >= x >=0:
```

5.7.11. ¿Qué contiene una lista?

Con la palabra reservada `in` podemos preguntar si un ítem está en un lista, veamos un ejemplo

```
lista = ["rojo", "naranjo", "verde", "azul"]
if "rojo" in lista:
    print "Era rojo"
```

La palabra clave `not` puede ser combinada con `in` para hacer la pregunta contraria, es decir, si un ítem no está en un lista. Veamos un ejemplo

```
lista = ["rojo", "naranjo", "verde", "azul"]
if "purpura" not in lista:
    print "No habia purpura"
```

5.7.12. Iteraciones con `while`.

La palabra reservada `while` puede ser usada para crear una iteración. La instrucción `while` necesita una condición que es cierta y luego deja de serlo, por ejemplo un contador que se incrementa y supera un valor límite. Ejemplo

```
x=0
while x < 10:
     print x
     x = x+1
```

Consejo: Para hacer una sección de código reusable, en vez de usar valores constantes use variables. Primero un ejemplo no generalizado

```
x=0
while x < 12:
     print 2*x
     x = x+1
```

Ahora el mismo ejemplo generalizado

```
max_num=12
num=2
x=0
while x < max_num:
     print num*x
     x = x+1
```

Utilicemos la instrucción `while` para hacer una salida ordenada para un programa. El código de escape del tabulador (`\t`) en un *string* permite hacer tabulaciones. Los tabuladores mantienen los items alineados dando una salida ordenada. Ejemplo, en este caso combinando la instrucción `while` y el código de escape del tabulador haremos una tabla:

```
x=1
while x < 10:
     print x, "\t", x*x
     x = x+1
```

5.7.13. Recorriendo un *string*.

Uno puede desear hacer una prueba sobre cada una de las letras que componen el *string* todas de una vez. Hay dos maneras de hacer esto usando una instrucción `while` o una instrucción `for` para realizar el ciclo o *loop*. Primero veamos el ciclo con `while`:

```
palabra = "computador"
indice = 0
while indice < len(palabra):
      letra = palabra[indice]
      print letra
      indice=indice +1
```

5.7.14. El ciclo `for`.

Una manera más compacta de escribir el ciclo `while` donde se recorrio un *strings* es usando un ciclo `for`, veamos cómo queda el código

```
palabra = "computador"
for letra in palabra:
      print letra
```

Notemos que hemos creado la variable `letra` cuando iniciamos el ciclo `for`. A continuación, un ejemplo más completo del ciclo `for`:

```
#!/usr/bin/env python
# -*- coding: iso-8859-1 -*-
# Programa que cuenta vocales

import string

palabra = raw_input("Entre una palabra : ")
```

```
palabra_min = string.lower(palabra)
vocales="aeiouáéíóú"
contador = 0
for letra in palabra_min:
    if letra in vocales:
       contador=contador +1
    else:
       pass
print "El número de vocales en la palabra que ingresó fueron : ", contador
```

Notemos la segunda línea de este programa que nos permite ingresar e imprimir *strings* con caracteres acentuados y caracteres especiales.

5.7.15. Un ciclo `for` y las listas.

Los ciclos `for` pueden ser usados con listas de la misma manera que lo eran con *strings*, un ejemplo para mostrarlo

```
emails = ["oto@mail.com", "ana@mail.com"]
for item in emails:
    envie_mail(item)
```

5.7.16. Generando listas de números.

La función `range(num_init, num_fin, num_paso)` toma tres argumentos enteros, el número de partida, el numero final y el paso, para generar una lista de enteros que comienza en el número de partida, termina con un número menor que el final saltandose el paso señalado, si se omite el paso el salto será de uno en uno. Veamos ejemplos

```
range(10) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(2,10) = [2, 3, 4, 5, 6, 7, 8, 9]
range(0,11,2) = [0, 2, 4, 6, 8, 10]
```

5.7.17. El comando `break`.

El comando `break` es capaz de salirse de un ciclo `for` o `while`. Un ejemplo que recorre los números calculando su cuadrado mientras sea menor que 50:

```
for n in range(1,10):
      c=n*n
      if c > 50:
            print n, "al cuadrado es ",c," > 50"
            print "PARO"
```

```
            break
      else:
            print n,"su cuadrado es ",c
```

5.7.18. El comando `continue`.

El comando `continue` es usado para que en un ciclo `for` o `while` se salte el resto del fragmento de código y continue con la próxima iteración del ciclo. Un ejemplo

```
for n in range(1,10):
      if n < 3:
        continue
      c=n*n
      if c > 50:
            print n, "al cuadrado es ",c," > 50"
            print "PARO"
            break
      else:
            print n,"su cuadrado es ",c
```

5.7.19. El comando `else`.

Un ciclo puede tener una sección `else`, esta es ejecutada cuando el ciclo termina por haber agotado la lista, en un ciclo `for` o cuando la comparación llega a ser falsa en un ciclo `while`, pero no cuando el ciclo es ternimado con `break`. A continuación, un programa que muestra este hecho y sirve para encontrar números primos

```
for n in range(2,10):
    for x in range(2,n):
        if n % x ==0:
            print n, "igual a", x,"*", n/x
            break
    else:
        print n,"es un numero primo"
```

5.8. Funciones Pre-hechas.

Una función define un conjunto de instrucciones o trozo de código, con un nombre asociado, que realiza una tarea específica devolviendo un valor y que puede ser reutilizado. La función puede ser creada por usted o importada desde algún módulo. Ejemplos de funciones:

- De cálculo matemático

  `log, sen, cos, tan, exp, hypot`.

- Funciones que generan números al azar, funciones de ingreso, funciones que hacen cambios sobre un *string*.

- Código hecho por el usuario que puede ser reciclado.

Hay un grupo de funciones que vienen hechas, es decir, listas para usar. Para encontrar qué funciones están disponibles tenemos la documentación del Python y un sitio web `http://www.python.org/doc/current/modindex.html`

Estas funciones pre-hechas vienen agrupadas en archivos llamados módulos. Una manera de importar, en nuestro programa, el módulo apropiado, que contiene la función que nos interesa, es usar el comando

```
import modulo_name
```

Una vez importado el módulo, cuando queremos llamar a la función para usarla, debemos dar el comando

```
modulo_name.function(arguments)
```

Veamos un ejemplo con la función `hypot` del módulo matemático

```
import math
print math.hypot(8,9)
```

Si analizamos las líneas anteriores de código debemos decir que el módulo que contiene las funciones matemáticas se llama `math` y éste incluye la función `hypot` que devuelve el largo de la hipotenusa. El símbolo `.` separa el nombre del módulo del de la función. Por supuesto `hypot` es el nombre de la función y `()` es el lugar para los argumentos. Una función podría tener o no tener argumentos, pero aún así deben ir los paréntesis, son obligatorios. Los números `8,9` son enviados a la función para que los procese. En el ejemplo, estos números corresponden a los dos catetos de un triángulo rectángulo.

En las secciones anteriores vimos funciones especializadas en el ingreso de *strings* y de números. Nos referimos a `input()` para números y a `raw_input()` para *strings*. En este caso, `input` e `raw_input` corresponden al nombre de las funciones, y entre los paréntesis se acepta un *string* como argumento, el cual es desplegado como *prompt* cuando se da el comando. Como vimos, este argumento es opcional en ambas funciones, sin embargo, lo incluyan o no, siempre se deben poner los paréntesis.

Funciones como `input()` y `raw_input()` están incorporadas al lenguaje y no necesitamos importar ningún módulo para usarlas.

5.8.1. Algunas funciones incorporadas.

- `float(obj)` Convierte un *string* u otro número a un número de punto flotante. Con decimales.
- `int(obj)` Convierte un *string* u otro número a un número entero. Sin decimales.
- `long(obj)` Convierte un *string* u otro número a un número entero largo. Sin decimales.
- `str(num)` Convierte un número a un *string*.
- `divmod(x,y)` Devuelve los resultados de `x/y` y `x%y`.
- `len(s)` Retorna el largo de un *string* u otro tipo de dato (una lista o diccionario).
- `pow(x,y)` Retorna `x` a la potencia `y`.
- `range(start,stop,step)` Retorna un conjunto de números desde `start` hasta `stop`, con un paso igual a `step`.
- `round(x,n)` Retorna el valor del punto flotante `x` redondeado a `n` digitos después del punto decimal. Si `n` es omitido el valor por defecto es cero.

5.8.2. La función que da el largo de un *string* o una lista.

Para encontrar cuántos caracteres tiene un *string* usamos la función `len(string)`. La función `len` requiere un *string* como argumento. Un ejemplo:

```
palabra = "computador"
largo = len(palabra)  # largo = 10
```

La sentencia `len(mylista)` da el largo de la lista `mylista`, es decir, su número de elementos.

5.8.3. Algunas funciones del módulo `math`.

Antes de usar estas funciones se debe importar el módulo `math`.

- `acos(x), asin(x), atan(x)` El arcocoseno, el arcoseno y la arcotangente de un número.
- `cos(x), sin(x), tan(x)` El coseno, el seno y la tangente de un número.
- `log(x), log10(x)` El logaritmo natural y el logaritmo en base 10 de un número.
- `hypot(x,y)` Retorna el largo de la hipotenusa de un triángulo rectángulo de catetos `x` e `y`.

5.8.4. Algunas funciones del módulo `string`.

Antes de usar estas funciones se debe importar el módulo `string`.

- `capitalize(string)` Pone en mayúscula la primera letra de la primera palabra.
- `capwords(string)` Pone en mayúscula la primera letra de todas las palabras.
- `lower(string)` Todas las letras en minúsculas.
- `upper(string)` Todas las letras en mayúsculas.
- `replace(string,old,new)` reemplaza todas las palabras `old` en `string` por las palabras `new`.
- `center(string, width)` Centra el `string` en un campo de un ancho dado por `width`.
- `rjust(string, width)` Justifica a la derecha el `string` en un campo de un ancho dado por `width`.
- `ljust(string, width)` Justifica a la izquierda el `string` en un campo de un ancho dado por `width`.
- `split(oracion)` Convierte un *string*, como una frase, en una lista de palabras.
- `join(lista)` Convierte una lista de palabras en una frase dentro de un *string*.

5.8.5. Algunas funciones del módulo `random`.

Antes de usar estas funciones se debe importar el módulo `random`.

- `randrange(start, stop, step)` Da un número pseudo al azar entre el número `start` y el número `stop-1`. El número `step` es opcional.
- `choice(sequence)` Elige al azar un objeto que pertenece a la secuencia `sequence` (una lista). Por ejemplo `sequence=["a", "b", "c", "d", "e"]`.

5.8.6. Algunos otros módulos y funciones.

Una función del módulo `time`:

- `sleep(x)` El computador queda en pausa por `x` segundos.

Un par de funciones del módulo `calendar`:

- `prcal(year)` Imprime un calendario para el año `year`.
- `prmonth(year, month)` Imprime un calendario para el mes `month` del año `year`.

5.9. Funciones hechas en casa.

Una función define con un nombre a un conjunto de instrucciones que realizan una tarea específica. A menudo son almacenadas en archivos llamados módulos. Pueden o no necesitar argumentos. Pueden o no retornar explícitamente un valor al programa, de no definirlo explícitamente por el programador la función retorna el valor `None`.

5.9.1. Receta para una función.

Para usar una función primero hay que definir la función, darle un nombre y escribir el conjunto de instrucciones que la constituyen. La función realizará las instrucciones cuando es llamada. Después, en el programa, llame la función que ya definió. A continuación veamos la definición formal de una función hecha por nosotros

```
def nombre(argumentos):
        instrucciones
```

Comenzamos con la palabra `def`, la cual es un palabra requerida. Debe ir en minúsculas. Luego `nombre` es el nombre que uno le da a la función. Después vienes los argumentos (`argumentos`) que corresponden a las variables que se le pasan a la función para que las utilice. Finalmente, `:`, requeridos al final de la línea que define una función. El bloque de `instruciones` asociados a la función deben tener sangría para identificarlos como parte de la misma. A continuación, un ejemplo:

```
# Definiendo la funcion
def mi_funcion():
    print "Nos gusta mucho la Fisica"

# Usando la funcion definida
mi_funcion()
```

La definición de una función puede estar en cualquier parte del programa con la salvedad que debe estar antes de que la función misma sea llamada. Una vez definidas las funciónes ellas se ejecutarán cuando sean llamadas. Cuando enviamos valores a nuestras funciones se crean las variables nombradas en la definición. Por ejemplo:

```
def mi_funcion(nombre1, nombre2):
        print nombre1+nombre2
```

Los nombres de las variables de una función sólo serán válidos dentro de la misma función, esto es lo que se conoce como variables *locales*. Todas las funciones usan por defecto variables locales.

5.9.2. Variables globales.

Si desea asignar una variable definida fuera de la función en la función, tiene que utilizar la sentencia `global`. Esta se emplea para declarar que la variable es global es decir que no es local.

Puede utilizar los valores de las variables definidas fuera de la función (y no hay variables con el mismo nombre dentro de la misma). Sin embargo, esto es inapropiado y debe ser evitado puesto que llega a ser confuso al lector del programa, en cuanto a donde se ha realizado dicha definición de variables. Usando la sentencia `global` queda claro que la variable se define en un bloque externo.

```
#!/usr/bin/python
def func():
    global x
    print 'x es', x

    x = 2
    print 'x cambiada a', x

#main
x = 50
func()
print 'El valor de x es', x
```

La salida del programa

```
x es 50
Cambiada a 2
El valor de x es 2
```

5.9.3. Pasando valores a la función.

Para enviar los valores a nuestra función ponga los valores en la llamada de la función. El tipo de los valores debe estar de acuerdo con lo que la función espera. Las funciones pueden tomar variables u otras funciones como argumentos. Veamos un ejemplo:

```
def mi_function(nombre1, nombre2):
    print nombre1,nombre2

mi_function("azul","rojo")
```

5.9.4. Valores por defecto de una función.

En algunas funciones, se puede hacer que el uso de algunos parámetros sean opcionales y usar valores predeterminados si el usuario no desea proporcionarlos (los valores de dichos parámetros). Esto se hace con la ayuda de valores pre-definidos. Puedes especificar los valores por defecto después del nombre del parámetro en la definición de la función con el operador de asignación (=) seguido por el argumento a definir.

```
#!/usr/bin/python

def say(s, times = 1):
        print s * times

say('Hola')
say('Mundo ', 5)
```

Salida del programa

```
Hola
MundoMundoMundoMundoMundo
```

Solamente los parámetros que están en el extremo de la lista de parámetros pueden tener valores por defecto; es decir, no puedes tener un parámetro con un valor por defecto antes de uno sin un valor, en el orden de los parámetros declarados, en la lista del parámetro de la función. Esto se debe a que los valores son asignados a los parámetros por la posición. Por ejemplo `def func(a, b=5)` es válido, pero `def func(a=5, b)` no lo es.

5.9.5. Argumentos claves.

Si se tiene funciones con muchos parámetros y se quiere especificar solamente algunos de ellos, entonces se puede asignar los valores para tales parámetros con sólo nombrarlos, a esto se denomina argumentos claves. Utilizamos el nombre en vez de la posición que se ha estado utilizando. Esto tiene dos ventajas: la primera, es que usar la función es más fácil puesto que no se necesita preocuparnos del orden de los argumentos. La segunda, es que podemos dar valores solamente a los parámetros que deseamos, a condición de que los otros tengan valores por defecto. Usando argumentos claves

```
#!/usr/bin/python

def func(a, b=5, c=10):
        print 'a es', a, 'y b es', b, 'y c es', c

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

La salida es:

```
a es 3 y b es 7 y c es 10
a es 25 y b es 5 y c es 24
a es 100 y b es 5 y c es 50
```

5.9.6. Documentación de una función, *docstrings*.

Python tiene una característica interesante llamada cadenas de documentación que generalmente se conocen por su nombre corto: *docstrings*. *Docstrings* es una herramienta importante de la que se puede hacer uso puesto que ayuda a documentar mejor el programa. Podemos incluso ubicar `docstring` en una función a tiempo de ejecución, es decir cuando el programa está funcionando. Usando *Docstrings*

```
#!/usr/bin/python

def printMax(x, y):
        '''Imprime el maximo de 2 numeros.

        Los dos valores deben ser enteros.  Si hubieran
        decimales, son convertidos a enteros.'''

        x = int(x) # Convierte a enteros, si es posible
        y = int(y)

        if x > y:
                print x, 'es maximo'
        else:
                print y, 'es maximo'

printMax(3, 5)
print printMax.__doc__
```

La salida

```
5 es maximo
Imprime el maximo de 2 numeros.

        Los dos valores deben ser enteros.  Si hubieran
        decimales, son convertidos a enteros.
```

5.9.7. Tuplas y diccionarios como argumentos.

Si la función recibe una tupla debemos declararla de la forma `*nombre_tupla`. Si la función recibe un diccionario debemos declararla de la forma `**nombre_diccionario`. Si una función recibe ambos tipos de argumentos deben ir `*nombre_tupla` antes que los `**nombre_diccionario`. Ejemplo

```
def funcion(*mi_tupla, **nihon_go):
    ..
```

Podemos definir una función con un número variable de argumentos colocando un signo `*` en un último argumento

```
def mi_funcion(argumento_1, *mas_argumentos):
    print argumento_1
    for i in mas_argumentos:
        print i,
    print
# main

mi_funcion(1)
mi_funcion(1,2)
mi_funcion(1,2,3)
mi_funcion(1,2,3,4)
```

Que tiene como salida

```
1

1
2
1
2 3
1
2 3 4
```

El número variable de argumentos se articula creando una tupla, con el nombre que se puso después del `*`, en el que se almacenan todos los argumentos opcionales. Si no se incluye ningún argumento adicional la tupla queda vacía.

5.9.8. La palabra clave `return`.

El comando `return` termina la ejecución de una función. Un ejemplo

```
import math

def raiz(num):
    if num<0:
         print "Ingrese un numero positivo"
         return
    print math.sqrt(num)
```

Consejo: Los condicionales como el `if` son especialmente útil para atrapar y manejar errores. Use el `else` para atrapar el error cuando la condición no es satisfecha.

5.9.9. Funciones que tienen un valor de retorno explícito.

Podemos crear funciones que retorne un valor al programa que las llamó. Por ejemplo:

```
def sumalos(x,y):
   new = x+y
   return new

# Llamada a la funcion
sum = sumalos(5,6)
```

5.9.10. Funciones que tienen más de un valor de retorno.

Podemos crear funciones que retorne más de un valor al programa que las llamó. La forma en que la función lo hace es devolviendo una tupla, por ejemplo:

```
def operaciones(x,y):
    suma = x+y
    resta = x-y
    prod= x*y
    div = x/y
    return suma,resta,prod,div

# Llamada a la funcion
print operaciones(5,6)
a,b,c,d = operaciones (8,4)
print a,b,c,d
```

Al ejecutar el código anterior su salida es

```
(11, -1, 30, 0)
12 4 32 2
```

5.9.11. Recursión.

Se llama recursión cuando una función se llama a si misma. La recursión permite repetir el uso de una función incluso dentro de la misma función. Un ejemplo es

```
def count(x):
   x=x+1
   print x
   count(x)
```

En este caso la función nunca para, este tipo de recursión es llamada recursión infinita. Para prevenir este situación creamos un caso base. El caso base es la condición que causará que la función pare de llamarse a si misma. Un ejemplo

```
def count(x):
   if x<100:
      x=x+1
      print x
      count(x)
   else:
      return
   time.sleep(1)
```

En un ejemplo, más adelante, veremos un programa que calcula el factorial de un número en forma recursiva.

5.9.12. Parámetros desde la línea de comando.

Python puede recibir parámetros directamente de la línea de comando. La lista `sys.argv` los contiene. Supongamos que el programa se llama `main.py` y es como sigue:

```
#!/usr/bin/python
import sys

for i in sys.argv:
    print i

print sys.argv[0]
```

Si ejecutamos el programa con la línea de comando

```
jrogan@huelen:~$ ./main.py h -l --mas xvzf
```

La salida será

```
./main.py
h
-l
--mas
xvzf
./main.py
```

Otro ejemplo, un programa que suma dos números desde la línea de comando,

```
#!/usr/bin/python
import sys

if (len(sys.argv)>2):
     n1=float(sys.argv[1])
     n2=float(sys.argv[2])
     print n1+n2
else:
     pass
```

Si ejecutamos el programa con la línea de comando

```
jrogan@huelen:~$ ./suma.py 1.2 3.5
```

La salida será

```
4.7
```

Si se llama el programa con menos argumentos, el programa no hará nada.

5.10. Ejemplos de funciones: raíz cuadrada y factorial.

5.10.1. Raíz cuadrada.

Con lo visto hasta ahora, ya podemos escribir una función que calcule la raíz cuadrada de un número. En general, para escribir esta función, debemos tener claro qué se espera de ella: cuántos son los argumentos que recibirá, si tendrá o no valor de retorno, y, por cierto, ponerle un nombre adecuado. Para la raíz cuadrada, es claro que el argumento es un número y evidentemente esperamos que el valor de retorno de la función sea también un número. Llamando a la función `raiz`, tenemos la declaración:

```
def raiz(x):
```

Debido a la naturaleza de la función raíz cuadrada, `raiz()` no tendría sentido, y por tanto no corresponde declararla con un valor *default*.

Ahora debemos pensar en cómo calcular la raíz cuadrada. Usando una variante del método de Newton-Raphson, se obtiene que la secuencia

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

converge a $\sqrt{a}$ cuando $n \to \infty$. Por tanto, podemos calcular la raíz cuadrada con aproximaciones sucesivas. El cálculo terminará en el paso N, cuando la diferencia entre el cuadrado de la aproximación actual, x_N, y el valor de a, sea menor que un cierto número pequeño: $|\, x_N^2 - a \,| < \epsilon \ll 1$. El valor de ϵ determinará la precisión de nuestro cálculo. Un ejemplo de código lo encontramos a continuación:

```
#!/usr/bin/python
#
# Programa que calcula la raiz cuadrada

import math

def raiz(a):
  x =a/2.0    # para comenzar
  dx, epsilon = 1e3, 1e-8;

  while (math.fabs(dx)>epsilon):
    x = (x + a/x)/2;
    dx = x*x - a;
    print "x = ", x, ", precision = ", dx
  return x

# main

r=input("Ingrese un numero: ")
print raiz(r)
```

Primero está la función `raiz`, y luego el `main`. En el `main` se pide al usuario que ingrese un número, el cual se aloja en la variable `r`, y se muestra en pantalla el valor de su raíz cuadrada.

En la implementación de la función hay varios aspectos que observar. Se ha llamado `x` a la variable que contendrá las sucesivas aproximaciones a la raíz. Al final del ciclo, `x` contendrá el valor (aproximado) de la raíz cuadrada. `dx` contiene la diferencia entre el cuadrado de `x` y el valor de `a`, `epsilon` es el número (pequeño) que determina si la aproximación es satisfactoria o no.

El ciclo está dado por una instrucción `while`, y se ejecuta mientras `dx>epsilon`, es decir, termina cuando `dx` es suficientemente pequeño. El valor absoluto del real `dx` se obtiene con la función matemática `fabs`, disponible en el módulo `math` incluido al comienzo del programa. Observar que inicialmente `dx=1e3`, esto es un valor muy grande; esto permite que la condición del `while` sea siempre verdadera, y el ciclo se ejecuta al menos una vez.

Dentro del ciclo, se calcula la nueva aproximación, y se envía a pantalla un mensaje con la aproximación actual y la precisión alcanzada (dada por `dx`). Eventualmente, cuando la aproximación es suficientemente buena, se sale del ciclo y la función devuelve al `main` el valor de `x` actual, que es la última aproximación calculada.

5.10.2. Factorial.

Otro ejemplo útil es el cálculo del factorial, definido para números naturales:

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \ , \quad 0! \equiv 1 \ .$$

Una estrategia es utilizar un ciclo `for`, determinado por una variable entera i, que va desde 1 a n, guardando los resultados en una variable auxiliar que contiene el producto de todos los números naturales desde 1 hasta i:

```
#!/usr/bin/python
#
# Programa que calcula el factorial

def factorial(i):
  f = 1
  for j in range(2,i+1):
    f = f*j
  return f

# main

n=input("Ingrese un numero: ")
print "El factorial de ", n, " es: ", factorial(n)
```

Observar que la variable auxiliar `f`, que contiene el producto de los primeros i números naturales, debe ser inicializada a 1. Si se inicializara a 0, `factorial(n)` sería 0 para todo `n`.

Esta función responde correctamente en el caso $n = 0$, pero retorna el valor 1 para todos los enteros negativos.

Otra estrategia para calcular el factorial es hacer uso de su propiedad recursiva

$$n! = n \times (n-1)! \quad 1! = 0! \equiv 1$$

Un programa que calcula el factorial en forma recursiva

```
#!/usr/bin/env python
def fact(n):
   if n<2:
      return 1
   else:
      return n*fact(n-1)
#main
i=input("Ingrese un  natural :")
print "El factorial de",i," es ",fact(i)
```

5.11. Programación orientada a objetos.

El paradigma de programación conocido como programación orientada a objetos (POO) consiste en representar o modelar los conceptos relevantes de nuestro problema a resolver mediante clases y objetos. El programa consistirá de una serie de interacciones entre estos objetos. Hay tres conceptos básicos en un lenguaje que trabaja en POO: la herencia, el polimorfismo y el encapsulamiento. Revisaremos estos conceptos más adelante en esta sección.

5.11.1. Objetos y clases.

Un objeto es una entidad que agrupa un estado y una funcionalidad relacionada. El estado del objeto esta descrito por variables conocidas como atributos. La funcionalidad se modela a través de funciones llamadas métodos del objeto.

Una clase es un plantilla genérica a partir de la cual se pueden crear instancias de los objetos. Esta plantilla define los atributos y métodos que tendrá los objetos de esa clase.

Para crear una clase se parte con la palabra reservada `class`, luego necesita un nombre para la clase. Los nombres de las clases, por convención, tiene la primera letra en mayúscula. Después del nombre se termina la línea con :. Luego de lo anterior, se crea el cuerpo de la clase, las instrucciones que forman este cuerpo deben ir con sangría. Si la primera línea del cuerpo corresponde a una cadena de texto, esta cadena será la documentación de la clase. En el cuerpo se definen las funciones o métodos de la clase, cada una de estas funciones debe incluír a `self` como parámetro. Veamos un ejemplo

```
class ClaseMFM0:
    """Un ejemplo de clase"""
    def __init__(self, nombre):
        self.alumno=nombre
    def saludo(self):
        print "Bienvenido", self.alumno
    def promedio(self,*notas):
        n=len(notas)
```

```
        suma=0
        for i in notas:
            suma +=i
        print "El promedio de",self.alumno,"es", round(suma/float(n),1)


pablo=ClaseMFM0("Pablo Parada")
pablo.saludo()
pablo.promedio(4,5.5,3)
pedro=ClaseMFM0("Pedro Perez")
pedro.saludo()
pedro.promedio(7,6,5,6,7,6,5)
pedro.promedio(6,5,7,4,6,3)
```

Creamos una instancia de una clase, un objeto, al asignarla a una variable, mediante la instrucción `pablo=ClaseMFM0("Pablo Parada")`. Para aplicar una función o método a la nueva instancia debemos especificar en forma completa la instancia y el método `pablo.saludo()`.

5.11.2. Clase de muestra LibretaNotas.

```
class LibretaNotas:
   def __init__(self, name, value):
        self.nombre = name
        self.puntaje = value
        self.evaluaciones = 1
   def sumanota(self, nota):
        self.evaluaciones += 1
        self.puntaje += nota
        self.elpromedio = self.puntaje/float(self.evaluaciones)
   def promedio(self):
        print self.nombre, ": promedio =", self.elpromedio
```

El parámetro `self` permite referirse al objeto actual. Para accededer a los atributos y métodos dentro del objeto se debe usar el `self` y luego el nombre del atributo o método.

El método `__init__` es especial, es la función que se ejecutará si una nueva instancia del objeto es creada. Esta función es especial y permite realizar cualquier proceso de inicialización que sea necesario.

Usando la clase `LibretaNotas`

```
eli   = LibretaNota('Elizabeth', 6.5)
mario =  LibretaNota('Mario', 6.0)
carmen =  LibretaNota('Carmen', 6.1)
```

```
eli.sumanota(6.2)
mario.sumanota(6.1)
carmen.sumanota(6.3)
eli.sumanota(6.8)
mario.sumanota(6.7)
carmen.sumanota(6.6)

eli.promedio()
mario.promedio()
carmen.promedio()
```

Cada nueva instancia de la clase `LibretaNotas` debe tener un nombre y una primera nota porque así lo requiere el método `__init__`. Notemos que cada instancia tiene su propio promedio.

5.11.3. Valores por defecto.

Un método puede usar valores por defecto, estos valores son usados cuando la función es llamada sin especificar los argumentos. Veamos un ejemplo de valores por defecto en una clase

```
class Dinero:
    def __init__(self, cantidad = 0) :
           self.cantidad=cantidad
    def imprime(self):
           print "Tienes", self.cantidad, "de dinero"
# Llamadas posibles

mi_dinero = Dinero(100)
tu_dinero = Dinero()
mi_dinero.imprime()
tu_dinero.imprime()
```

5.11.4. Herencia.

Al modelar un conjunto heterogeneo que comparte ciertos atributos y funcionalidades, pero que además, tiene subconjuntos que poseen propiedades en común entre ellos pero no con los otros aparece el concepto de herencia. Podemos definir una clase (clase madre o superclase) que contenga los atributos y métodos en común y luego definimos otras clases (clases hijas o subclases) que herede los atributos y métodos comunes y que defina los atributos y métodos específico de los distintos subconjuntos.

Para indicar cuál es la clase madre de la cual hereda la clase hija se coloca el nombre de la clase madre entre paréntesis después del nombre de la clase hija. Veamos un ejemplo de clase madre. Supongamos que queremos modelar recetas de cocinas

```
#Herencia en Python
class Libro:
    """Clase madre de la cual heredan las demas"""

    def __init__(self, autor, titulo, isbn, paginas, precio) :
        self.autor = autor
        self.titulo = titulo
        self.isbn = isbn
        self.paginas = paginas
        self.precio = precio

    def printPublicita(self) :
        print "Del afamado autor", self.autor,
        print "su nueva obra", self.titulo,
        print "un volumen de", self.paginas,
        print "paginas a solo: $", self.precio

class Diccionarios(Libro) :
    """Clase hija, hereda de Libro sus atributos y sus metodos"""
    pass

class Novelas(Libro):
    """Clase hija, hereda de Libro sus atributos y sus metodos"""
    pass

class Manual(Libro):
    """Clase hija, hereda de Libro sus atributos y sus metodos
       de manuales de programacion"""
    pass

#main
mi_libro=Manual("Guido van Rossum", "An Introduction to Python",
"9780954161767", 164, 7000)
mi_libro.printPublicita()
```

Si queremos especificar un nuevo parámetro a la hora de crear una clase hija debemos reescribir el método `__init__` en la clase hija. Si sólo necesitamos ejecutar sólo un par de nuevas instrucciones y usar el método de la clase madre entonce usamos la sintaxis:

`ClaseMadre.metodo(self, args)` para llamar al método de igual nombre de la clase madre. Supongamos, en nuestro ejemplo, que para la subclase `Manual` queremos especificar el lenguaje, al cual se refiere el manual, es ese caso debemos sobre escribir el método `__init__` como sigue

```
#Herencia en Python
class Libro:
    """Clase madre de la cual heredan las demas"""

    def __init__(self, autor, titulo, isbn, paginas, precio) :
        self.autor = autor
        self.titulo = titulo
        self.isbn = isbn
        self.paginas = paginas
        self.precio = precio

    def printPublicita(self) :
        print "Del afamado autor", self.autor,
        print "su nueva obra", self.titulo,
        print "un volumen de", self.paginas,
        print "paginas a solo: $", self.precio

class Diccionarios(Libro) :
    """Clase hija, hereda de Libro sus atributos y sus metodos"""
    pass

class Novelas(Libro):
    """Clase hija, hereda de Libro sus atributos y sus metodos"""
    pass

class Manual(Libro):
    """Clase hija, hereda de Libro sus atributos y sus metodos
       de manuales de programacion"""

    def __init__(self, lenguaje, autor, titulo, isbn, paginas, precio) :
        self.lenguaje=lenguaje
        Libro.__init__(self, autor, titulo, isbn, paginas, precio)

    def printPublicita(self) :
        print "Del afamado autor", self.autor,
        print "su nueva obra de", self.lenguaje,
```

```
        print ":",self.titulo,
        print "un volumen de", self.paginas,
        print "paginas a solo: $", self.precio

#main
mi_libro=Manual("Python", "Guido van Rossum", "An Introduction to Python",
"9780954161767", 164, 7000)
mi_libro.printPublicita()
```

Notemos que tambien sobreescribimos, para particularizarlo, el método `printPublicita(self)`.

5.11.5. Herencia multiple.

En Python existe la posibilidad de herencia múltiple, es decir, una clase puede heredar de varias clases madres simultaneamente. Para implementar esta posibilidad basta con especificar las clases madres de las que se hereda separándolas por comas en la declaración de la clase hija. En el caso de que las clases madres tuvieran métodos con el mismo nombre y número de parámetros el método que se hereda es el de la clases más a la derecha en la definición.

Por ejemplo, podríamos tener una clase Cerveza que heredara de la clase `Bebida_Alcoholica`, métodos como `beber()` y atributos como `porcentaje_de_alcohol` y de la clase `Alimento`, con métodos como `comer()` y atributos como `aporte_carbohidratos` o `aporte_proteinas`:

```
class Cerveza(Bebida_Alcoholica, Alimento):
      pass
```

Existe ambiguedad, si en ambas clases madres esta presente un método con el mismo nombre y con el mismo número de argumento, digamos un método`aporte_calorias(cantidad)` está presente en nuestro ejemplo. El método que heredara, la clase hija, será el de la clase `Alimento`, por ser la clase madre que está más a la derecha en la definición.

5.11.6. Polimorfismo.

Polimorfismo se refiere a la capacidad de los objetos de diferentes clases de responder al mismo mensaje. Esto se suele conseguir a través de herencia, de la siguiente manera: un objeto de una clase hija es simultaneamente un objeto de la clase madre, de esta forma donde se requiere un objeto de la clase madre se puede utilizar uno de la clase hija.

Python no impone restricciones a los tipos de los argumentos que se le pueden pasar a una función, por lo tanto, el polimorfismo en Python no es de gran importancia.

El término, polimorfismo, también se utiliza para referirse a la sobrecarga de métodos, que se define como la capacidad del lenguaje de determinar qué método ejecutar de entre varios métodos con igual nombre según el tipo o número de los parámetros que se le pasa. En

Python no existe sobrecarga de métodos (el último método sobreescribría la implementación de los anteriores).

5.11.7. Encapsulación.

La encapsulación consiste en restringir el acceso a determinados métodos o atributos de los objetos, estableciendo así qué puede utilizarse desde fuera de la clase y qué no. En otros lenguajes existen modificadores de acceso que definen si cualquiera puede acceder a esa función o variable (por ejemplo, `public` en `c++`) o si está restringido el acceso sólo a la propia clase (por ejemplo, `private` en `c++`). En Python, no existen este tipo de modificadores de acceso, y lo que se suele hacer es que se restringe el acceso a un atributo o método usando un tipo de nombre especial. Si el nombre comienza con dos guiones bajos (y no termina también con dos guiones bajos, en ese caso sería un método especial) se trata de un atributo o método privado, en caso contrario es público.

```
# Encapsulacion en Python
class Libro:
    def __init__(self, costo) :
        self.__costo = costo

    def printPrecioPublico(self) :
        print 1.5*self.__costo

    def __printPrecioAmigos(self) :
        print 1.3*self.__costo
```

Al tratar de acceder el atributo

```
milibro=Libro(100)
print milibro.__costo
```

la salida es

```
Traceback (most recent call last):
  File "encap.py", line 15, in <module>
    print milibro.__costo
AttributeError: Libro instance has no attribute '__costo'
```

Al tratar de acceder el método

```
milibro=Libro(100)
milibro.printPrecioPublico()
milibro.__printPrecioAmigos()
```

la salida es

```
150.0
Traceback (most recent call last):
  File "encap.py", line 17, in <module>
    milibro.printPrecioAmigos()
AttributeError: Libro instance has no attribute 'printPrecioAmigos'
```

Los errores indican que el objeto no acepta tener ese atributo o ese método, respectivamente. Lo anterior, es debido a que las estamos declarando privados. Sin embargo, los atributos o métodos no son realmente privados y pueden ser accesados usando llamadas "especiales"

```
print milibro._Libro__costo
milibro._Libro__printPrecioAmigos()
```

5.11.8. Atributos comunes a toda una clase.

Si necesitamos que algún atributo lo puedan acceder todas las instancias de una clase, éste puede ser almacenada como atributo de la clase. Para crear atributos de la clase completa ellos se crean fuera de todas las definiciones de los métodos de la clase. Estos atributos pueden ser accesado dentro de las definición de los métodos de la clase usando la notación `NombredelaClase.NombredelaVariable`. Un ejemplo

```
#!/usr/bin/env python

class Cuentas:
    lista_usuarios=[]
    def __init__(self,nombre):
        self.nombre=nombre
        Cuentas.lista_usuarios.append(self.nombre)
    def egreso(self):
        Cuentas.lista_usuarios.remove(self.nombre)
        print "Felicitaciones por tu egreso", self.nombre,"\b."
        print
    def eliminado(self):
        Cuentas.lista_usuarios.remove(self.nombre)
        print "Lo sentimos", self.nombre,"usted ha sido eliminado."
        print

def imprime_usuarios():
    if len(Cuentas.lista_usuarios)>0:
        print "Los usuarios con cuenta son:"
```

```
        for i in Cuentas.lista_usuarios:
            print "\t","\t","\t",i
        print
    else:
        print "No hay  usuarios con cuenta."
# main

hugo = Cuentas("Hugo Hurtado")
paco = Cuentas("Paco Paredes")
luis = Cuentas("Luis Luco")

imprime_usuarios()

hugo.egreso()
imprime_usuarios()
luis.egreso()
imprime_usuarios()
paco.eliminado()
imprime_usuarios()
```

Notese que la lista de usuarios es siempre llamada por su nombre completo `Cuentas.alumnos`. Para acceder a la lista fuera de la clase, use su nombre completo `Cuentas.alumnos`.

5.11.9. Métodos especiales.

Existen un conjunto de métodos con significados especiales, tal como el método `__init__` que ya vimos, todos ellos se caracterizan porque sus nombres siempre comienzan y terminan con dos guiones bajos

A continuación, los métodos especiales utilizado en la inicialización y en el borrado de los objetos:

`__init__(self, args)` Método invocado después de crear el objeto, realizar las tareas de inicialización.

`__del__(self)` Método invocado cuando el objeto va a ser borrado. También llamado el destructor, se utiliza para realizar tareas de limpieza.

Las clases pueden ampliar los métodos regulares de Python para que funciones sobre los nuevos objetos que crean. Para definir estas ampliaciones debemos usar otros nombres especiales cuando definimos los métodos en la clase. Veamos algunos de ellos:

`__str__(self)` Método invocado para crear una cadena de texto que represente al objeto. Se utiliza cuando usamos print para mostrar el objeto o cuando usamos la función str(obj) para crear una cadena a partir del objeto.

`__cmp__(self, otro)` Método invocado cuando se utilizan los operadores de comparación para comprobar si el objeto es menor, mayor o igual a un segundo objeto pasado como

parámetro. Este método debe devolver un número, negativo si el objeto es menor, cero si son iguales, y positivo si el objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores <, <=, >o >= se lanzará una excepción. Si se utilizan los operadores == o != para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto.

`__len__(self)` Método invocado para comprobar la longitud del objeto. Se utiliza, por ejemplo, cuando se llama a la función `len(obj)` sobre el objeto. El método devuelve la longitud del objeto.

`__repr__` Método invocado para imprimir el objeto.

`__add__(self, otro)` Método invocado cuando se utiliza el operador `+` entre el objeto y un segundo objeto que es pasado como parámetro.

`__sub__(self, otro)` Método invocado cuando se utiliza el operador `-` entre el objeto y un segundo objeto que es pasado como parámetro.

`__mul__(self, otro)` Método invocado cuando se utiliza el operador `*` entre el objeto y un segundo objeto que es pasado como parámetro.

`__div__(self, otro)` Método invocado cuando se utiliza el operador `/` entre el objeto y un segundo objeto que es pasado como parámetro.

`__pow__(self, otro)` Método invocado cuando se utiliza el operador `**` entre el objeto y un segundo objeto que es pasado como parámetro.

Los anteriores, no son los únicos métodos especiales que existen. Para una enumeración sistematica revise la documentación.

5.11.10. Ejemplos.

Veamos un primer ejemplo para mostrar la acción de los métodos `__init__` y `__del__`

```
#!/usr/bin/env python

class Puramente_informativa:
    def __init__(self,nombre):
        print "Objeto inicializado"
    def __del__(self):
        print "Objeto destruido"

# main

a = Puramente_informativa()
b = Puramente_informativa()

print "Termine"
```

Para ilustrar el uso de algunos de estos métodos especiales, escribimos una clase de vectores bidimensionales en la cual se definen la suma, la resta, el producto y la impresión, entre sus métodos.

```
#!/usr/bin/env python

from math import sqrt


class Vec2d:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def module(self):
        return sqrt(self.x**2+self.y**2)
    def __repr__(self):
        return "(%9.5f,%9.5f)" % (self.x,self.y)
    def __add__(self,newvec):
        return Vec2d(self.x+newvec.x,self.y+newvec.y)
    def __sub__(self,newvec):
        return Vec2d(self.x-newvec.x,self.y-newvec.y)
    def __mul__(self,newvec):
        return self.x*newvec.x+self.y*newvec.y

#main

a=Vec2d(1.3278,2.67)
b=Vec2d(3.1,4.2)

print "Primer vector:", a, "Segundo vector:",b
print a+b
print a-b
print a*b
print a.module(),b.module()
```

5.11.11. El __main__.

Al final de un archivo independiente que contiene una clase, es decir en un módulo, se suele poner un par de líneas que permiten probar la clase cuando este archivo independiente es llamado como si fuera un programa. Las líneas toman la forma

```
if __name__ == '__main__':
```

```
   metodo_que_hace_pruebas()
```

Lo anterior, también permite definir una función principal en un programa

```
#!/usr/bin/env python

def main():
    pass

if __name__ == '__main__':
   main()
```

5.12. Objetos conocidos.

Algunos tipos básicos, que vimos en la sección 5.4, y todos los contenedores o tipos avanzados, que vimos en la sección 5.6, son realmente objetos. A continuación, revisaremos algunos métodos de los diferentes objetos ya vistos.

5.12.1. *String.*

Para ilustrar algunos de los métodos del objeto `string`, supongamos que lo asignamos a `esta es una cadena espaciada`, supongamos, además, que usaremos como `substring` a la cadena `es`, como `separador` a la cadena y como `sequencia` a la cadena `=*=`, es decir, definimos en modo interactivo:

```
>>> string = "esta es una cadena espaciada"
>>> substring = "es"
>>> secuencia="=*="
>>> separador="cadena"
```

`string.count(substring[, start[, end]])` Devuelve el número de veces que se encuentra `substring` en `string`. Los parámetros `start` y `end` son opcionales y definen la porción de `string` donde se realizara la busqueda. Ejemplo

```
>>> string.count(substring)
3
>>> string.count(substring,1)
2
>>> string.count(substring,1,7)
1
```

`string.find(subtring[, start[, end]])` Devuelve la posición en la que se encontró, por primera vez, `substring` en `string`. Si no se encontró devuelve -1.

```
>>> string.find(substring)
0
>>> string.find(substring,2)
5
>>> string.find(substring,8,21)
19
>>> string.find("kadena")
-1
```

`string.join(secuencia)` Devuelve una cadena que resulta de concatenar los caracteres de la secuencia separadas por la cadena `string`, es decir, sobre la que se llama el método.

```
>>> string.join(secuencia)
'=esta es una cadena espaciada*esta es una cadena espaciada='
```

`string.partition(separador)` Busca la cadena `separador` en `string` y devuelve una tupla con una subcadena desde el inicio hasta el `separador`, el `separador`, y una subcadena desde el `separador` hasta el final del`string`. Si no encuentra el `separador`, la tupla contendrá el `string` completo y dos cadenas vacías.

```
>>> string.partition(separador)
('esta es una ', 'cadena', ' espaciada')
>>> string.partition(secuencia)
('esta es una cadena espaciada', '', '')
```

`string.replace(substring, secuencia[, veces])` Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena `substring` por la cadena `secuencia`. Si el parámetro `veces` se especifica explícitamente, este indica el número máximo de ocurrencias a reemplazar.

```
>>> string.replace(substring, secuencia)
'=*=ta =*= una cadena =*=paciada'
>>> string.replace(substring, secuencia,1)
'=*=ta es una cadena espaciada'
```

`string.split([separador [,maximo]])` Devuelve una lista conteniendo las subcadenas en las que se divide `string` al dividirlo con el delimitador `separador`. SI no se especifica el `separador`, se usa espacio. Si se especifica `maximo`, indica el número máximo de particiones a realizar.

```
>>> string.split(separador)
['esta es una ', ' espaciada']
>>> string.split()
['esta', 'es', 'una', 'cadena', 'espaciada']
>>> string.split(" ",2)
['esta', 'es', 'una cadena espaciada']
>>>
```

5.12.2. Listas.

Ilustremos algunos de los métodos del objeto `lista` en modo interactivo:
`lista.append(objeto)` Añade `objeto` al final de la lista `lista`.

```
>>> lista=['a','e','i','o','u','a','e']
>>> objeto="B"
>>> lista.append(objeto)
>>> lista
['a', 'e', 'i', 'o', 'u', 'a', 'e', 'B']
>>>
```

`lista.count(objeto)` Devuelve el número de veces que se encontró `objeto` en la lista `lista`.

```
>>> lista=['a','e','i','o','u','a','e']
>>> objeto="a"
>>> lista.count(objeto)
2
```

`lista.extend(otra_lista)` Agrega cada uno de los elementos de `otra_lista` a, como elementos a la lista `lista`.

```
>>> lista=['a','e','i','o','u','a','e']
>>> otra_lista=['1','2','3']
>>> lista.extend(otra_lista)
>>> lista
['a', 'e', 'i', 'o', 'u', 'a', 'e',  '1', '2', '3']
>>> lista=['a','e','i','o','u','a','e']
>>> lista.append(otra_lista)
>>> lista
['a', 'e', 'i', 'o', 'u', 'a', 'e', ['1', '2', '3']]
```

`lista.index(objeto[, start[, stop]])` Devuelve la posición en la que se encontró la primera ocurrencia de `objeto`. Si se especifican, start y stop definen las posiciones de inicio y fin de una sublista en la que buscar.

```
>>> lista=['a','e','i','o','u','a','e']
>>> objeto="e"
>>> lista.index(objeto)
1
>>> lista.index("a",3,6)
5
```

`lista.insert(indice, objeto)` Inserta `objeto` en la posición `indice`.

```
>>> lista=['a','e','i','o','u','a','e']
>>> objeto="B"
>>> indice=2
>>> lista.insert(indice, objeto)
>>> lista
['a', 'e', 'B', 'i', 'o', 'u', 'a', 'e']
```

`lista.pop([indice])` Devuelve el valor en la posición `indice` y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.

```
>>> lista=['a','e','i','o','u','a','e']
>>> indice=2
>>> lista.pop(indice)
'i'
>>> lista
['a', 'e', 'o', 'u', 'a', 'e']
>>> lista.pop()
'e'
>>> lista
['a', 'e', 'o', 'u', 'a']
```

`lista.remove(objeto)` Eliminar la primera ocurrencia `objeto` de en la lista.

```
>>> lista=['a','e','i','o','u','a','e']
>>> objeto="e"
>>> lista.remove(objeto)
>>> lista
['a', 'i', 'o', 'u', 'a', 'e']
>>> lista.remove(objeto)
>>> lista
['a', 'i', 'o', 'u', 'a']
```

`lista.reverse()` Invierte la lista. Esta función trabaja sobre la propia lista desde la que se invoca el método, no sobre una copia.

```
>>> lista
['a', 'i', 'o', 'u', 'a']
>>> lista=['a','e','i','o','u','a','e']
>>> lista.reverse()
>>> lista
['e', 'a', 'u', 'o', 'i', 'e', 'a']
```

`lista.sort(cmp=None, key=None, reverse=False)` Ordena la lista. Si se especifica `cmp`, este debe ser una función que tome como parámetro dos valores `x` e `y` de la lista y

devuelva -1 si `x` es menor que `y`, 0 si son iguales y 1 si `x` es mayor que `y`. El parámetro `reverse` es un booleano que indica si la lista se ordenará de forma inversa o no, lo que sería equivalente a llamar primero a `lista.sort()` y después a `lista.reverse()`. Por último, si se especifica, el parámetro `key` debe ser una función que tome un elemento de la lista y devuelva una clave a utilizar a la hora de comparar, en lugar del elemento en si.

```
>>> lista=['a','e','i','o','u','a','e']
>>> lista.sort()
>>> lista
['a', 'a', 'e', 'e', 'i', 'o', 'u']
```

5.12.3. Diccionarios.

`diccionario.get(k[, defecto])` Busca el valor de la clave `k` en el diccionario. Es equivalente a utilizar `diccionario[k]` pero al utilizar este método podemos indicar un valor a devolver por defecto `defecto` si no se encuentra la clave, mientras que con la sintaxis `D[k]`, de no existir la clave se lanzaría una excepción.

```
>>> diccionario={ 1:"uno", 2:"dos", 3:"tres"}
>>> diccionario.get(2)
'dos'
>>> diccionario.get(4,"no esta")
'no esta'
>>> diccionario[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 4
```

`diccionario.has_key(k)` Comprueba si el diccionario tiene la clave `k`.

```
>>> diccionario={ 1:"uno", 2:"dos", 3:"tres"}
>>> diccionario.has_key(2)
True
>>> diccionario.has_key(4)
False
```

`diccionario.items()` Devuelve una lista de tuplas con pares clave:valor.

```
>>> diccionario={ 1:"uno", 2:"dos", 3:"tres"}
>>> diccionario.items()
[(1, 'uno'), (2, 'dos'), (3, 'tres')]
```

`diccionario.keys()` Devuelve una lista de las claves del diccionario.

```
>>> diccionario={ 1:"uno", 2:"dos", 3:"tres"}
>>> diccionario.keys()
[1, 2, 3]
```

`diccionario.pop(k[, default])` Borra la clave `k` del diccionario y devuelve su valor. Si no se encuentra dicha clave se devuelve `default` si se especificó el parámetro, sino se especificó lanza una excepción.

```
>>> diccionario={ 1:"uno", 2:"dos", 3:"tres"}
>>> diccionario.pop(2,"no esta")
'dos'
>>> diccionario
{1: 'uno', 3: 'tres'}
>>> diccionario.pop(4,"no esta")
'no esta'
>>> diccionario.pop(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 4
```

`diccionario.values()` Devuelve una lista de los valores almacenados en el diccionario.

```
>>> diccionario={ 1:"uno", 2:"dos", 3:"tres"}
>>> diccionario.values()
['uno', 'dos', 'tres']
```

5.13. Programación Funcional.

La programación funcional es otro paradigma de programación que se basa casi en su totalidad en el manejo de funciones, entendiendo el concepto de función según su definición matemática. En los lenguajes funcionales puros un programa debería consistir exclusivamente en la aplicación de distintas funciones a un valor de entrada para obtener un valor de salida. Python, no es un lenguaje funcional puro pero incluye características de los lenguajes funcionales como son las funciones de orden superior y las funciones anónimas.

5.13.1. Funciones de orden superior.

En las funciones de orden superior pueden aceptar funciones como parámetros de entrada o devolver funciones como valor de retorno. Las funciones, como parámetro o valor de retorno, son usadas como si se trataran de objetos cualesquiera.

Como en Python todo son objetos, las funciones también lo son, lo que permite manipularlas como si se tratara de cualquiera otro objeto. Veamos algunos ejemplos

```
import string
def add(num1,num2):
    print num1+num2
def mult(num1,num2):
    print num1*num2

# Programa
num1 = input("Entre el primer numero: ")
num2 = input("Entre el segundo numero: ")
menu = {'S':add, 'M':mult}
print "S para sumar y M para multiplicar: "
choice = string.upper(raw_input())
menu[choice] (num1,num2)
```

Otro ejemplo

```
def hola_mundo(idioma="es"):
    def hola_mundo_es():
        print "Hola Mundo"
    def hola_mundo_en():
        print "Hello World"
    def hola_mundo_de():
        print "Hallo Welt"

    eleccion = {"es":hola_mundo_es,
    "en":hola_mundo_en,
    "de":hola_mundo_de}
    return eleccion[idioma]

f = hola_mundo()
f()
g = hola_mundo("de")
g()
hola_mundo("en")()
```

5.13.2. Iteraciones sobre listas.

Entre las funciones de orden superior que reciben una función como argumento, están `map`, `filter` y `reduce`. Su aplicación nos permiten reducir muchos de los ciclos `for` de la programación imperativa.

Función `map`.

`map(funcion, secuencia[, secuencia, ...])`. La función de orden superior `map` devuelve una lista como resultado de aplicar `funcion` a cada uno de los elemento de `secuencia`. Si la función, usada como parámetro, necesita más de un argumento entonces se necesita más de una secuencia. Si alguna de las secuencias, pasadas como parámetros, tiene menos elementos que las otras el valor que le recibe la función es `None`. A continuación ejemplos:

```
>>> import math
>>> lista1, tupla1= [1,2,3,4], (10,20,30)
>>> map(math.sqrt,lista1)
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
>>> map(math.sqrt,tupla1)
[3.1622776601683795, 4.4721359549995796, 5.4772255750516612]
>>> lista2=[1,3,5,7]
>>> map(math.hypot,lista1,lista2)
[1.4142135623730951, 3.6055512754639891, 5.8309518948453007, 8.0622577482985491]
>>> def sumar(x,y):
...     return x+y
...
>>> map(sumar,lista1,lista2)
[2, 5, 8, 11]
>>> lista2=[1,3,5,7,9]
>>> map(sumar,lista1,lista2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sumar
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Función `filter`.

`filter(funcion, secuencia)`. La función de orden superior `filter` devuelve una lista como resultado de verificar si los elementos de `secuencia` cumplen o no una determinada condición. A cada elemento de `secuencia` se le aplica `funcion` que debe retornar un valor booleano, si el resultado de aplicar `funcion` al elemento de `secuencia` es `True` se incluye el elemento en la lista que retorna, si el resultado es `False` no se incluye. A continuación un ejemplo:

```
>>> def mayor_que_10(x):
...     return x>10
...
>>> lista=[3,6,9,12,15]
>>> filter(mayor_que_10, lista)
[12, 15]
```

Función `reduce`.

`reduce(funcion, secuencia[, inicial])`. La función de orden superior `reduce` devuelve un valor que es el resultado de aplicar `funcion` a pares consecutivos de elementos de `secuencia` hasta que la lista se reduzca a un solo valor. Si `inicial` está presente es puesto antes de los elementos de la `secuencia` a reducir. Sirve como un valor por defecto cuando la secuencia es muy corta o esta vacia. Si `inicial` no es especificado y `secuencia` contiene un solo elemento este es el devuelto por `reduce`.

La función `reduce` será removida de las bibliotecas standard en `Python 3.0`. Para usarla se le deberá importar del módulo `functools`. A continuación, ejemplos en que se suman y/o se multiplican todos los elementos de una lista.

```
>>> def sumar(x,y):
...     return x+y
...
>>> def mult(x,y):
...     return x*y
...
>>> lista=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
>>> reduce(sumar,lista)
120
>>> reduce(mult,lista)
1307674368000L
>>> reduce(mult,lista,0)
0
>>> reduce(mult,[2],3)
6
>>> reduce(mult,[1])
1
```

5.13.3. Las funciones lambda.

Las funciones lambda son funciones anónimas, es decir, sin nombre y por lo tanto no pueden ser llamada más tarde.

La sintaxis de las funciones lambda parte por la palabra `lambda` luego el o los parámetros se especifican, separados por comas y sin paréntesis, después van dos puntos : y, finalmente, el código de la función. Las funciones lambda están restringidas, por sintaxis, a una sola expresión.

Esta construcción es muy útil para reducir código. Volvamos a revisar los ejemplos para `map`, `filter` y `reduce` usando, ahora, funciones lambda

```
>>> lista1=[1,2,3,4,5]
```

```
>>> lista2=[1,3,5,7,9]
>>> map(lambda x, y : x+y, lista1,lista2)
[2, 5, 8, 11, 14]
>>> lista=[3,6,9,12,15]
>>>
>>> filter(lambda x: x>10, lista)
[12, 15]
>>> lista=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
>>> reduce(lambda x, y : x+y, lista)
120
>>> reduce(lambda x, y : x*y, lista)
1307674368000L
```

5.13.4. Compresión de listas.

En `Python` 3.0 `map`, `filter` y `reduce` perderán importancia a favor de la comprensión de listas. La comprensión de listas consiste en una construcción que permite crear listas a partir de otras listas.

Cada construcciones consta de una expresión que determina cómo modificar los elementos de la lista original, seguida de una o más sentencias `for` y opcionalmente uno o más `if`.

Veamos los ejemplos anteriores usando compresión de listas.

```
>>> lista1=[1,2,3,4,5]
>>> lista2=[1,3,5,7,9]
>>> [x+y for x in lista1
...     for y in lista2
[2, 4, 6, 8, 10, 3, 5, 7, 9, 11, 4, 6, 8, 10, 12, 5, 7, 9, 11, 13, 6, 8, 10, 12, 14]
>>> [x+y for x in lista1
...     for y in lista2]
...     if lista1.index(x)==lista2.index(y)]
[2, 5, 8, 11, 14]
>>> lista=[3,6,9,12,15]
>>> [x for x in lista
...     if x>10]
[12, 15]
```

5.13.5. Expresiones generadoras y generadores.

Las expresiones generadoras comparten la misma sintaxis que la compresión de lista salvo que se utilizan paréntesis `( )` en lugar de parénteis de corchetes `[ ]`. Sin embargo, la diferencia, es que no devuelven una lista sino un objeto llamado generador.

```
>>> lista=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
>>> [2*n for n in lista]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
>>> (2*n for n in lista)
<generator object <genexpr> at 0xb73cab6c>
```

Por otra parte, los generador son una clase especial de función que generan valores sobre los cuales se puede iterar. Para devolver los valores a iterar se usa la palabra reservada `yield`. Veamos un ejemplo para aclararlo

```
>>> def fn_generadora(inicial, final, paso):
...     iterador=inicial
...     while (iterador< final):
...             yield iterador
...             iterador+=paso
...
>>> fn_generadora(0,10,2)
<generator object fn_generadora at 0xb73caedc>
```

El generador puede utilizarse donde se necesite un objeto iterable. Por ejemplo, en un ciclo `for-in`, siguiendo con el ejemplo anterior

```
>>> for i in fn_generadora(0,10,2):
...     print i,
...
0 2 4 6 8
```

Los generadores ocupan menos memoria que una lista explícita con todos los valores a iterar, ya que los valores se generan sólo un valor a la vez. Siempre podemos generar un lista a partir de un generador usando la función `list`:

```
>>> list(fn_generadora(0,10,2))
[0, 2, 4, 6, 8]
```

5.13.6. Decoradores.

Un decorador es una función, de orden superior, que recibe una función como parámetro y devuelve otra función como resultado. Los decoradores se usan habitualmente para agregarle funcionalidades extras a funciones ya existentes. Por ejemplo, si se tiene un conjunto de funciones ya escritas y por razones de optimización se quiere que estan funciones informen el tiempo que consumen cuando se ejecutan podemos usar un decorador. A continuación, se muestra un código completo en que se programa un decorador llamado `decorador_tiempo`

```
#!/usr/bin/env python

import time

# En esta variable global se acumulara el tiempo total de ejecucion
Tiempo_total=0.0

# Este es el decorador
def decorador_tiempo(funcion):
    def nueva(*lista_args):
        tiempo_ini = time.time()
        valor = funcion(*lista_args)
        tiempo_fin = time.time()
        delta = tiempo_fin - tiempo_ini
        global Tiempo_total
        Tiempo_total+=delta
        print "tardo en ejecutarse", delta,"segundos."
        return valor
    return nueva

@decorador_tiempo
def func1(a, b):
    return a + b

@decorador_tiempo
def func2(a, b):
    return a - b

def main():
    func1(2, 3)
    func2(4, 3)
    global Tiempo_total
    print "El tiempo total es", Tiempo_total

if __name__ == '__main__':
    main()
```

Podemos aplicar más de un decorador, esta vez se requiere que informe el nombre de la función en ejecución. Para ello escribimos una segunda versión del programa que incluya un nuevo decorador llamado `decorador_nombre` y luego añidiendo una nueva línea con el nombre del nuevo decorador sobre las funciones ya existentes. Los decoradores se ejecutarán de abajo hacia arriba. A continuación la segunda versión del código.

```
#!/usr/bin/env python
import time
# En esta variable global se acumulara el tiempo total de ejecucion
Tiempo_total=0.0

def decorador_nombre(funcion):
    def nueva(*lista_args):
        valor = funcion(*lista_args)
        print "La funcion", funcion.__name__,
        return valor
    return nueva

def decorador_tiempo(funcion):
    def nueva(*lista_args):
        tiempo_ini = time.time()
        valor = funcion(*lista_args)
        tiempo_fin = time.time()
        delta = tiempo_fin - tiempo_ini
        global Tiempo_total
        Tiempo_total+=delta
        print "tardo en ejecutarse", delta,"segundos."
        return valor
    return nueva
@decorador_tiempo
@decorador_nombre
def func1(a, b):
    return a + b

@decorador_tiempo
@decorador_nombre
def func2(a, b):
    return a - b

def main():
    func1(2, 3)
    func2(4, 3)
    global Tiempo_total
    print "El tiempo total es", Tiempo_total

if __name__ == '__main__':
    main()
```

5.14. Excepciones.

Los errores producidos durante la ejecución de un programa se conocen como excepciones, si estas excepciones no se capturan el programa se interrunpe mostrando un error en la consola. Sin embargo, Python es capaz de gestionar estas excepciones capturandolas mediante bloques `try...except` y también permite que el programador genere sus propias excepciones mediante el uso de la sentencia `raise`.

Las excepciones no son raras, aparecen a menudo en Python, por ejemplo, cuando se accesa a una clave de diccionario que no existe, se provoca una excepción del tipo `KeyError`; cuando en una lista se busca un valor que no existe se provoca una excepción del tipo `ValueError`; cuando se invoca a un método que no existe se provoca una excepción del tipo `AttributeError`; cuando en el programa se refieren a una variable que no existe se provoca una excepción del tipo `NameError`; cuando se mezclan tipos de datos, sin convertirlos previamente, se provoca una excepción del tipo `TypeError`; cuando se trata de abrir un archivo que no existe se provoca una excepción del tipo `IOError`.

5.14.1. Bloque `try ... except`.

Primero el bloque que corresponde a `try`, que es el fragmento de código en que podría producirse una excepción, se tratará de ejecutar. Si alguna excepción se produce la ejecución saltará al bloque de comandos encabezados por `except`. Lo anterior, puede ser usado en programas que pueden presentar una falla bajo alguna circunstancia y de esta forma que la ejecución no se detiene. Veamos un código sin `try...except`:

```
#!/usr/bin/env python
# Sin Try / Except

filename = raw_input("Entre el nombre de un archivo que NO existe: ")
file = open(filename) # instruccion que veremos mas adelante
```

Con salida

```
Entre el nombre de un archivo que NO existe: nn
Traceback (most recent call last):
  File "./error1.py", line 4, in <module>
    file = open(filename)
IOError: [Errno 2] No such file or directory: 'nn'
```

Ahora el mismo código con un bloque `try...except`:

```
#!/usr/bin/env python
# Con Try / Except
```

```
filename = raw_input("Entre el nombre de un archivo que NO existe: ")
try:
   file = open(filename)
except:
   print "Archivo no encontrado."
```

Con salida

```
Entre el nombre de un archivo que NO existe: nn
Archivo no encontrado.
```

Incluso puede reescribirse en forma más interactiva

```
#!/usr/bin/env python
# Con Try / Except, segunda version

def open_file(filename=""):
    if filename=="":
        filename = raw_input("Entre el nombre del archivo: ")
    try:
        myfile = open(filename)
    except:
        print "Archivo %s() no encontrado." % filename
        myfile = open_file()
    return myfile

# main

file1 = open_file()
file2 = open_file("algo.txt")
```

Python permite utilizar más de uno `except` con un solo bloque `try`, de forma que podamos tomar acciones distintas en caso de diferentes excepciones. Par esto basta indicar el nombre del tipo excepción a continuación del `except`. A continuación una mejora del código anterior

```
#!/usr/bin/env python
# Con Try / Except, tercera version

def open_file(filename=""):
    if filename=="":
        filename = raw_input("Entre el nombre del archivo: ")
    try:
        myfile = open(filename)
```

```
    except IOError:
        print "Archivo %s() no encontrado." % filename
        myfile = open_file()
    except TypeError:
        print "El nombre del archivo necesita ser un string"
        myfile = open_file()
    return myfile

# main

file1 = open_file()
file2 = open_file("algo.txt")
file3 = open_file(1)
```

Cuando aparece una excepción en el bloque `try`, se busca en cada uno de los `except` por el tipo de error que se produjo. En caso de que no se encuentre, se propaga la excepción. Además, se puede hacer que un mismo `except` sirva para tratar más de una excepción usando una tupla para listar los tipos de errores que manejará el bloque:

```
#!/usr/bin/env python
# Con Try / Except, otra version

def open_file(filename=""):
    if filename=="":
        filename = raw_input("Entre el nombre del archivo: ")
    try:
        myfile = open(filename)
    except (IOError, TypeError):
        print "Problemas con el archivo %s, reintentelo." % filename
        myfile = open_file()
    return myfile

# main

file1 = open_file()
file2 = open_file("algo.txt")
file3 = open_file(1)
```

Los bloques `try...except` puede contar, además, con un bloque `else`, que corresponde a un fragmento de código que se ejecutará sólo si no se ha producido ninguna excepción en el bloque `try`.

```
#!/usr/bin/env python
```

```
# Con Try / Except, cuarta version

def open_file(filename=""):
    if filename=="":
        filename = raw_input("Entre el nombre del archivo: ")
    try:
        myfile = open(filename)
    except IOError:
        print "Archivo %s() no encontrado." % filename
        myfile = open_file()
    except TypeError:
        print "El nombre del archivo necesita ser un string para el nombre"
        myfile = open_file()
    else:
        print "El archivo se leyo sin problemas"
    return myfile

# main

file1 = open_file()
file2 = open_file("algo.txt")
file3 = open_file(1)
```

Existe, también, la posibilidad de incluir una clausula `finally` que se ejecutará siempre, se produzca o no una excepción. Esta clausula se suele utilizar, entre otras cosas, para tareas de limpieza.

5.14.2. Comando `raise`.

El programador puede crear sus propias excepciones, para esto basta crear una clase que herede de `Exception` o de cualquiera de sus clases hijas y se lanza con `raise`.

```
#!/usr/bin/env python
# Con Try / Except, quinta version
import string


class ErrorDeTipoDeArchivo(Exception):
    def __init__(self,codigo):
        self.codigo=codigo
    def __str__(self):
        return "El archivo no es del tipo correcto, error %s " % self.codigo
```

```
def open_file(filename=""):
    if filename=="":
        filename = raw_input("Entre el nombre del archivo: ")
    try:
        l=len(filename)
        if  string.upper(filename[l-3:l])!="XYZ":
            raise ErrorDeTipoDeArchivo(33)
        myfile = open(filename)
    except IOError:
        print "Archivo %s() no encontrado." % filename
        myfile = open_file()
    except TypeError:
        print "El nombre del archivo necesita ser un string para el nombre"
        myfile = open_file()
    else:
        print "El archivo se leyo sin problemas"
    return myfile

# main

file1 = open_file()
file2 = open_file("algo.txt")
file3 = open_file(1)
```

Cuya saliada es

```
Entre el nombre del archivo: entry.png
Traceback (most recent call last):
  File "./error6.py", line 31, in <module>
    file1 = open_file()
  File "./error6.py", line 17, in open_file
    raise ErrorDeTipoDeArchivo(33)
__main__.ErrorDeTipoDeArchivo: El archivo no es del tipo correcto, error 33
```

Escribamos un ejemplo sencillo donde la excepción es lanzada directamente

```
#!/usr/bin/env python

inicial = 33

while True:
    print inicial,
    inicial -= 5
```

```
    if inicial < 0:
        print
        raise Exception, 'La variable es menor que cero'
```

La salida de este código es

```
33 28 23 18 13 8 3
Traceback (most recent call last):
  File "./raise.py", line 10, in <module>
    raise Exception, 'La variable es menor que cero'
Exception: La variable es menor que cero
```

5.14.3. La instrucción `assert`.

La instrucción `assert` en `Python` permite definir condiciones que debe cumplirse siempre. En caso que la expresión booleana sea `True` `assert` no hará nada, pero en caso de que sea `False` `assert` lanzará una excepcion. Reescribimos el ejemplo final de la subsección anterior con `assert`.

```
inicial = 33

while inicial<100:
    print inicial,
    inicial -= 5
    assert inicial >= 0, 'La variable es menor que cero'
```

La salida de este código es

```
#!/usr/bin/env python

33 28 23 18 13 8 3
Traceback (most recent call last):
  File "./raise2.py", line 8, in <module>
    assert inicial >= 0, 'La variable es menor que cero'
AssertionError: La variable es menor que cero
```

Es más corto que escribir el `if`, es claro para el que lo lee y adicionalmente la sentencia no se ejecuta en caso que el intérprete se invoque con `-O`.

5.15. Modulos.

Algunos problemas que se presentan a medida que los códigos crecen son: con cientos de líneas es muy fácil perderse; trabajar en equipo se hace difícil con códigos muy grandes. Entonces, sería bueno poder separar el código en pequeños archivos independientes.

5.15.1. Dividiendo el código.

El código puede ser dividido en archivos separados llamados modulos. Se ha usado varios de estos módulos anteriormente, los módulos `string`, `math`, `random`, entre otros. Para crear un módulo sólo se debe sálvar el código de la manera usual. El archivo salvado, con extensión `.py`, puede ser usado como módulo al importarlo desde otro programa. El nuevo módulo es usado tal como cualquier otro módulo de Python.

5.15.2. Creando un módulo.

Lo principal a tener en cuenta cuando se almacena código en un módulo es que ese código sea reusable. El código almacenado debe ser, principalmente, clases y funciones. Hay que evitar tener variables o comandos fuera de las definiciones de las funciones. Las funciones pueden requerir valores desde el programa quien las llama. Se salva el código como un archivo regular `.py`. Luego en el programa principal, importa el módulo y se usa.

5.15.3. Agregando un nuevo directorio al *path*.

Cuando Python busca módulos sólo lo hace en ciertos directorios. La mayoría de los módulos que vienen con Python son salvados en `/usr/lib/python`. Cuando salve sus propios módulos seguramente lo hará en un lugar diferente, luego es necesario agregar el nuevo directorio a `sys.path`. Hay que consignar que el directorio desde el cual se invoca Python, si está en el `path`. Para editar el `sys.path`, en el modo interactivo tipee

```
>>> import sys
>>> sys.path #imprime los actuales directorios en el path
>>> sys.path.append('/home/usuario/mis_modulos')
```

Dentro de un *script* usamos para importar mi módulos `mis_funciones` que está salvado en mi directorio de módulos

```
import sys
sys.path.append('/home/usuario/mis_modulos')

import mis_funciones
```

5.15.4. Documentando los módulos.

Como ya vimos los comentarios con triple comilla son usados para agregar documentación al código, ejemplo

```
def mi_funcion(x,y):
    """mi_funcion( primer nombre, ultimo nombre) """
```

Se debe usar triple comilla al principio y al final del texto. El texto entre las triples comillas debería explicar lo que la función, clase o módulo hace. Estas líneas de documentación se pueden ver, en el modo interactivo, si se da el comando `help(module.mi_funcion)`.

5.15.5. Usando un módulo.

Para cargar un módulo se debe incluir la palabra reservada `import` y el nombre del módulo, sin extensión, del archivo en que fue almacenado el módulo. Cuando se llama a una función que pertenece al módulo se debe incluir el nombre del módulo . el nombre de la función, Esta no es la única manera de importarlos y usarlos, veamos unos ejemplo, primero la forma habitual:

```
# Sean f(x,y) una funcion y C una clase con un metodo m(x) del modulo stuff

import stuff

print stuff.f(1,2)
print stuff.C(1).m(2)
```

una segunda forma

```
# Sean f(x,y) una funcion y C una clase con un metodo m(x) del modulo stuff

from stuff import f, C

print f(1,2)
print C(1).m(2)
```

una tercera forma

```
# Sean f(x,y) una funcion y C una clase con un metodo m(x) del modulo stuff

from stuff import *

print f(1,2)
print C(1).m(2)
```

una última manera

```
# Sean f(x,y) una funcion y C una clase con un metodo m(x) del modulo stuff

import stuff as st

print st.f(1,2)
print st.C(1).m(2)
```

5.15.6. Trucos con módulos.

La clausula `import` permite importar varios módulos en la misma línea,

```
import math, time
```

En el caso de que Python no encontrara un módulo con el nombre especificado, se lanza una excepción de tipo `ImportError`.

Un módulo puede ser corrido como programa independiente si se incluye las siguientes líneas al final del módulo

```
if __name__ == '__main__':
   funcion_a_correr()
```

Se puede sustituir `funcion_a_correr()` por el nombre de la función principal en el módulo.

5.15.7. Paquetes

Los módulos sirven para organizar y dividir el código, los paquetes y subpaquetes sirven para organizar los módulos. Los paquetes y subpaquetes son tipos especiales de módulos que permiten agrupar módulos relacionados. Mientras los módulos se corresponden con los archivos, los paquetes y subpaquetes se corresponden con los directorios y subdirectorios.

Para que Python considere que un directorio es un paquete basta crear un archivo con nombre `__init__.py`, habitualmente vacío, en el directorio en cuestión. Para que un módulo pertenezca a un paquete determinado, basta con copiar el archivo, que corresponde al módulo, en el directorio que corresponde al paquete. Para importar paquetes se utiliza, también, las palabras reservadas `import` y `from ... import ...`. El caracter `.` sirve para separar paquetes, subpaquetes y módulos. Un ejemplo:

```
#!/usr/bin/env python

import paquete.subpaquete.modulo
```

```
# Para usar alguna funcion que pertenece al modulo

paquete.subpaquete.modulo.func()
```

5.16. Pickle y Shelve.

Otra dificultad que se presenta al programar en Python es como salvar las estructuras de datos, es bien cuando se crea un diccionario, este se puede salvar como un archivo de texto, y luego leerlo, el problema es que no podemos leerlo como un diccionario. Sería importante poder salvar las listas como listas, los diccionarios como diccionarios y así luego poder leerlos e incorporarlos al código en forma fácil y directa.

5.16.1. Preservando la estructura de la información.

Existen dos métodos de preservar la data:

- El módulo `pickle` que almacena una estructura de datos de Python en un archivo binario. Está limitado a sólo una estructura de datos por archivo.
- El módulo `shelve` que almacena estructuras de datos de Python pero permite más de una estructura de datos y puede ser indexado por una llave.

5.16.2. ¿Cómo almacenar?

Se importa el módulo `shelve`, se abre un archivo *shelve*, se asigna un item, por llave, al archivo *shelve*. Para traer la data de vuelta al programa, se abre el archivo *shelve* y se accesa el item por la llave. Los *shelve* trabajan como un diccionario, se puede agregar, accesar y borrar items usando sus llaves.

5.16.3. Ejemplo de *shelve*.

```
import shelve
colores = ["verde", "rojo", "azul"]
equipos = ["audax", "union", "lachile"]
shelf = shelve.open('mi_archivo')
# Almacenendo items
shelf['colores'] = colores
shelf['equipos'] = equipos
# trayendolos de vuelta
newlist = shelf['colores']
# Cerrando
shelf.close()
```

5.16.4. Otras funciones de *shelve*.

- Para tomar una lista de todas las llaves disponibles en un archivo *shelve*, use la función `keys()`:
  `lista = shelf.keys()`
- Para borrar un item, use la función `del`:
  `del shelf('ST')`
- Para ver si una llave existe, use la función `has_key()`:
  `if shelf.has_key('ST'): print "si"`

5.17. Trabajando con archivos.

El lenguaje Python puede ser usado para crear programas que manipulan archivos sobre un sistema de archivos en un computador. El módulo `os` contiene las funciones necesarias para buscar, listar, renombrar y borrar archivos. El módulo `os.path` contiene unas pocas funciones especializadas para manipular archivos. Las funciones necesarias para abrir, leer y escribir archivos son funciones intrínsecas de Python.

5.17.1. Funciones del módulo `os`.

Funciones que sólo dan una mirada.

- `getcwd()` Retorna el nombre el directorio actual.
- `listdir(path)` Retorna una lista de todos los archivos en un directorio.
- `chdir(path)` Cambia de directorio. Mueve el foco a un directorio diferente.

Función que ejecuta un comando del sistema operativo.

- `system('comando')` Ejecuta el `comando`

Funciones que agregan.

- `mkdir(path)` Hace un nuevo directorio con el nombre dado.
- `makedirs(path)` Hace un subdirectorio y todos los directorios del `path` requeridos.

Funciones que borran o remueven.

- `remove(path)` Borra un archivo.
- `rmdir(path)` Borra un directorio vacío.
- `removedirs(path)` Borra un directorio y todo dentro de él.

Funciones que cambian.

- `rename(viejo,nuevo)` Cambia el nombre de un archivo de `viejo` a `nuevo`
- `renames(viejo,nuevo)` Cambia el nombre de un archivo de `viejo` a `nuevo` cambiando los nombres de los directorios cuando es necesario.

5.17.2. Funciones del módulo `os.path`.

Funciones que verifican.

- `exists(file)` Retorna un booleano si el archivo `file` existe.
- `isdir(path)` Retorna un booleano si el `path` es un directorio.
- `isfile(file)` Retorna un booleano si el `file` es un archivo.

5.17.3. Ejemplo de un código.

Un programa que borra todo un directorio

```
import os, os.path
path = raw_input("Directorio a limpiar : ")
os.chdir(path)
files= os.listdir(path)
print files
for file in files:
    if os.path.isfile(file):
        os.remove(file)
        print "borrando", file
    elif os.path.isdir(file):
        os.removedirs(file)
        print "removiendo", file
    else:
        pass
```

5.17.4. Abriendo un archivo.

Para abrir un archivos debemos dar la instrucción `open(filename,mode)` donde `filename` es el nombre del archivo y el `mode` corresponde a una de tres letras `"r"` para lectura solamente del archivo, `"w"` para escritura y `"a"` para agregar al final del archivo para poder manejar un archivo abierto hay que crear una variable con él. Ejemplo

```
salida= open("datos.txt","w")
salidaAppend= open("programa.log","a")
entrada= open("archivo.txt","r")
# A continuación se deberian usar los archivos abiertos

# Y finalmente cerrarlos
```

5.17.5. Leyendo un archivo.

- `entrada.read()` Lee el archivo completo como un *string*.
- `entrada.readline()` Lee una línea en un *string*.
- `entrada.readlines()` Lee el archivo completo, cada línea llega a ser un item tipo *string* en una lista.

5.17.6. Escribiendo a un archivo.

- `salida.write(string)` Escribe el `string` al archivo. Cómo se escribira este depende de en qué modo el archivo fue abierto.
- `salida.writelines(list)` Escribe todos los items tipo *string* en la lista `list`. Cada elemento en la lista estará en la misma línea a menos que un elemento contenga un caracter de `newline`

Si queremos usar la instruccion `print` para escribir sobre un archivo abierto, digamos `salida`, podemos usar la instrucción

```
salida = open("resultados.txt", "w")
print >> salida, datos  # Imprime datos en el archivo resultados.txt
print >> salida         # Imprime una linea en blanco en el archivo resultados.txt
salida.close()
```

5.17.7. Cerrando un archivo.

Los archivos que han sido abiertos, para lectura o escritura, que ya se ocuparon, deben de ser cerrados. El comando para cerrar el archivo `file`, previamente abierto con un comando `open`, se da a continuación:

- `file.close()` Cierra un archivo previamente abierto.

5.17.8. Archivos temporales.

Las funciones en el módulo `tempfile` puede ser usadas para crear y manejar archivos temporales. La instrucción `tempfile.mkstemp()` devuelve una lista en el que el segundo item es un nombre al azar que no ha sido usado. Los archivos temporales estarán localizados en el directorio temporal por defecto.

5.17.9. Ejemplo de lectura escritura.

```
# Programa que reemplaza una palabra vieja por otra nueva

import string, tempfile, os

# Preguntarle al usuario por Informacion
filename = raw_input("Nombre del archivo: ")
find = raw_input("Busque por: ")
replace = raw_input("Reemplacelo por: ")
# Abra el archivo del usuario, lealo y cierrelo
file = open(filename, "r")
text = file.readlines()
file.close()
# Edite la informacion del archivo del usuario

nueva = []

for item in text:
    line = string.replace(item, find, replace)
    nueva.append(line)

# Cree un nuevo archivo temporal
newname=tempfile.mkstemp()
temp_filename=newname[1]
newfile = open(temp_filename, "w")
newfile.writelines(nueva)
newfile.close()

# Cambie los nombres de los archivos y borra los temporales
oldfile=filename+"~"
os.rename(filename, oldfile)
os.system(" cp "+temp_filename+" "+filename)
os.remove(temp_filename)
```

5.18. Algunos módulos interesantes.

Hay muchos módulos que le pueden ser útiles, aquí le sugerimos unos pocos particularmente importantes.

5.18.1. El módulo `Numeric`.

Extensión numérica de Python que agrega poderosos arreglos multidimensionales.

```
>>> import Numeric as num
>>> a = num.zeros((3,2), num.Float)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> a[1]=1
>>> a
array([[ 0.,  0.],
       [ 1.,  1.],
       [ 0.,  0.]])
>>> a[0][1]=3
>>> a
array([[ 0.,  3.],
       [ 1.,  1.],
       [ 0.,  0.]])
>>> a.shape
(3, 2)
```

5.18.2. El módulo `Tkinter`

Este paquete es el encargado de ayudarnos a desarrollar la interfaz gráfica con el usuario de nuestro programa. Para comenzar a utilizarlo veamos el siguiente ejemplo:

Ejemplo

```
from Tkinter import *
vent=Tk()
etiqueta=Label(vent, text="Hola Mundo!!")
etiqueta.grid(row=1, column=1)
vent.mainloop()
```

Al ejecutar el programa

En la primera linea importamos el módulo Tkinter, en la segunda creamos la ventana, la que es proporcionada por el window manager, en la tercera linea generamos una etiqueta, y que en particular para este ejemplo ocupamos la opción de texto "`text`", en la cuarta linea ubicamos la etiqueta en la fila 1 y la columna 1 de la ventana (como no hay mas elementos en el ejemplo, la ventana se distribuye como una matriz de 1×1) y en la última linea se dice al computador que ejecute la orden (e.d. la ventana no aparecerá antes de leer esta linea), por lo que toda acción escrita después de esta linea no aparecerá en la ventana.

Elementos importantes en una ventana.

La forma de configurar un elemento en la ventana es de la forma:

```
Funcion(master, opcion1=valor1, opcion2=valor2, ...)
```

Donde cada elemento esta determinado por una `Funcion`, la variable `master` es la que determina a la ventana particular (como en el ejemplo anterior lo era `vent`), y todas las opciones siguientes tienen un valor por defecto que podemos cambiar solo enunciándolas y dandole el nuevo valor. Algunos de los elementos que podemos agregar son:

1. **Etiquetas**

   Se enuncian llamando a la función `Label(...)`, cuyas opciones son:

   `text` : El texto que llevará escrito la etiqueta. (p.e. `text="Hola!"`)

   `image` : Si quiero agregar una imagen. (p.e. `image=imagen`[3])

   `fg` : El color del texto, si es que la etiqueta esta conformada por texto (p.e. `fg="red"`)

   `bg` : El color del fondo de la etiqueta (p.e. `bg="black"`)

   `width` : Nos dice el largo de la etiqueta en la ventana (p.e. width=100)

2. **Botones**

   Se enuncian llamando a la función `Button(...)`, tiene las mismas opciones que la etiqueta y además se le agragan las siguientes:

   `command` : Aqui ponemos el nombre de la función que se ejecuta cuando se clickea el boton.

   `relief` : Es el relieve que tendra el boton, esta variable puede tomar los valores FLAT, RAISED, SUNKEN, GROOVE y RIDGE, el valor por defecto es RAISED, estéticamente se ven:

[3] Donde la variable `imagen` está determinada por la función `imagen=PhotoImage(file="archivo.gif")`

`cursor` : Es el nombre del cursor que aparece cuando pasamos el mouse sobre el boton, algunos de los valores que puede tomar esta opción son: `arrow, mouse, pencil, question_arrow, circle, dot, star, fleur, hand1, heart, xterm, etc`.

`bitmap` : Los bitmaps son imágenes prehechas que vienen incorporadas en Tkinter, algunos de los valores que puede tomar ésta variable son: `error, hourglass, info, questhead, question, warning, etc`, estéticamente se ven:

3. Input

Es equivalente a la opción `raw_input`, y se enuncia llamando a la función `Entry(...)`, tiene disponibles las opciones `witdh, bg, fg, cursor`, entre otros. Se ve de la forma:

4. Boton de Checkeo

Se llama con la función `Checkbutton(...)` y tiene las mismas opciones que `Button()`, se ve así:

5. Menú

Para crear un menú debemos generar una variable similar al `master` que determinaba la ventana, pero esta vez para determinar al menú, la forma de hacer esto es con la función `Menu(...)`, cuyo argumento debe ser la variable que representa a la ventana. Luego, con esta variable puedo crear un menú, veamos el siguiente ejemplo:

```
vent=Tk()
.
.
.
men=Menu(vent)

archivo=Menu(men, tearoff=0)
men.add_cascade(label="Archivo", men=archivo)
archivo.add_command(label="Abrir", command=A)
archivo.add_command(label="Nuevo", command=B)
archivo.add_command(label="Salir", command=C)

editar=Menu(men, tearoff=0)
men.add_cascade(label="Editar", men=editar)
editar.add_command(label="Copiar", command=D)
editar.add_command(label="Cortar", command=E)
vent.config(menu=men)
.
.
.
vent.mainloop()
```

El código anterior generará un menú llamado Archivo, que contiene las opciones Abrir (cuya funcion es A()), Nuevo (cuya función es B()) y Salir (con función C()), y otro menú llamado Editar, de similares características.

Ubicación de un elemento en la ventana.

Para ubicar en la ventana los distintos elementos existe la función `grid(... )` cuyos argumentos son desde que fila (`row`) hasta que fila (`rowspan`) y desde que columna (`column`) hasta que columna (`columnspan`) se encuentra el lugar que determinaré para el elemento particular, es importante saber que las filas se enumeran de arriba hacia abajo y las columnas de izquierda a derecha. Para ejecutar `grid()` correctamente debemos escribirla seguida de un punto despues de la variable que tiene asignada el elemento, por ejemplo:

```
vent=Tk()
.
.
.
A=Label(vent, text="hola")
```

```
B=Button(vent, text="aceptar", command=F)
A.grid(row=1, rowspan=2, column=1, columnspan=3)
B.grid(row=3, column=1)
.
.
.
vent.mainloop()
```

El código anterior pondrá el texto "hola" desde la primera fila hasta la segunda y desde la primera columna hasta la tercera, y debajo, en la fila 3 y la columna 1 pondrá un botón que dice "aceptar" y ejecuta la función `F`.

Atributos del `master`.

El `master` como ya lo hemos visto, es la variable que está determinada por la función `Tk()`, y la forma de darle distintos atributos es:

`master.atributo(valor)` ó `master.atributo(opcion1=valor1,...)`

Algunos de los atributos que le puedes asignar a una ventana son.

Título : La forma de ponerle el título a la ventana es dando la orden:

`master.title("texto en el titulo de mi ventana")`

Geometría : Se refiere a las dimensiones de la ventana (la unidad que se ocupa es 0,023 cm), se debe dar la orden:

`master.geometry("nxm")`, donde `n`=ancho y `m`=alto.

Configuración : Aquí ponemos algunas de las características generales de la ventana, como por ejemplo, el color de fondo, el cursor que aparece cuando el mouse pasa sobre ella, etc. Se debe dar la orden:

`master.configure(cursor="nombre cursor", background= "color")`

Un ejemplo más elaborado.

Veamos un ejemplo usando los elementos explicados.

```
#IMPORTO EL PAQUETE
from Tkinter import *

#DEFINO UNA FUNCION
def salir():
```

```
    vent.destroy()

#DEFINO EL MASTER
vent=Tk()

#ATRIBUTOS DEL MASTER
vent.title("Ejemplo")
vent.geometry("350x120")
vent.configure(cursor="spider")

#GUARDO UNA IMAGEN EN LA VARIABLE dibujo1
dibujo1  = PhotoImage(file="tux.gif")

#DEFINO LOS ELEMENTOS DE LA VENTANA
Dibujo1  = Label(vent, image=dibujo1, bg="white")
Nombre   = Label(vent, text="Nombre", fg="white", bg="red")
nombre   = Entry(vent, width=20, bg="white")
Apellido = Label(vent, text="Apellido", fg="white", bg="red")
apellido = Entry(vent, width=20, bg="white")
boton    = Button(vent,width=30, text="Salir", cursor="hand1", \
           fg="white", bg="black", command=salir)

#UBICO LOS ELEMENTO EN LA VENTANA
Dibujo1 .grid(row=1, rowspan=3, column=1)
Nombre  .grid(row=1, column=2)
nombre  .grid(row=1, column=3)
Apellido.grid(row=2, column=2)
apellido.grid(row=2, column=3)
boton   .grid(row=3, column=2, columnspan=3)

#EJECUTO EL CODIGO
vent.mainloop()
```

Al ejecutar el programa el resultado es:

5.18.3. El módulo `Visual`.

Un módulo que permite crear y manipular objetos 3D en un espacio 3D.

Capítulo 6

Ejercicios Propuestos

6.1. Sistema Operativo

1. En las placas madres actuales existen elementos llamados puertos que permiten la conexion entre distintos dipositivos. A continuación se nombran algunos de ellos. ¿Cuáles son los elementos más comunes que se conectan en dichos puertos?

   - COM.
   - LPT.
   - USB.
   - IDE.
   - SATA.
   - PS/2.

2. Enumere los elementos básicos de *hardware* que se necesitan para el funcionamiento de un sistema GNU/Linux. Describa brevemente cada uno de ellos.

3. En un ataque de curiosidad, el ambicioso y retirado Bill quiere conocer más sobre el sistema operativo UNIX, y en particular sobre Linux. Para esto, se pone en contacto con usted y le hace varias preguntas:

   - ¿Qué es un sistema operativo?
   - ¿Cuáles son las características del sistema operativo UNIX?
   - ¿Qué es Linux? ¿Qué es una distribución de Linux? y ¿cuáles son sus ventajas? Nombre 5 distribuciones.
   - ¿Qué es kernel? ¿Donde está en la red? ¿Quién lo mantiene?
   - ¿Qué es *software* libre?

- ¿Qué es la GNU?
- ¿Cuál es la diferencia entre Linux y GNU/Linux?
- ¿Por qué los virus no ejercen efecto en Linux?
- ¿Cuál es la distribución de GNU/Linux favorita de Sheldon?
- ¿Cuáles son los nombres de los computadores de la sala de computación de Física?

4. En un momento de claridad, el ambicioso Bill quiere conocer más sobre el proyecto Debian. Para esto, se pone en contacto con Ian Murdock y le hace tres preguntas:

   - ¿Cuál es el significado de una versión estable en Debian? ¿Cuántas versiones han sido estables y cuáles han sido sus nombres claves?
   - ¿Cuáles son 5 ventajas que ofrece esta distribución con respecto a las otras?

   Ante la pregunta de Bill, Ian entra en un colapso y no puede contestarlas. ¿Qué le contestará usted a Bill?

5. Después de tener claros los conceptos que rodean a una distribución, Bill contacta a Linus para conocer cómo instalar el sistema Debian GNU/Linux en su computador personal. Linus le responde que debe seguir los siguientes pasos:

   - Conocer la arquitectura en que va a ser instalado el sistema.
   - Conseguir la distribución.
   - Crear las particiones.
   - Instalar el sistema base.

   Describa cada paso para que Bill logre instalar un sistema básico en su computador.

6. Entre los distintos *software* para la visualización de archivos `pdf` existe el desarrollado por la empresa `Adobe`.

   a) Averigüe el nombre de este *software*.

   b) En términos de portabilidad, ¿qué ventajas tiene el formato de archivo `pdf` con respecto a otros?

   c) ¿Por qué este *software* no se incluye en Debian GNU/Linux?

   d) Explique detalladamente una manera de instalar este *software* en Debian GNU/Linux.

7. Uno de los *software* que más han causado sensación en la comunidad es el explorador de Internet Google Chrome.

a) Éste se distribuye únicamente en formato `deb`. Averigüe qué es un formato `deb` y nombre al menos dos distribuciones que utilicen este sistema de paquetes.

b) Averigüe qué comando debe realizar para instalar este *software* en su computador.

8. Averigüe dos formas de instalar una tarjeta gráfica `nvidia` en su computador.

6.2. Comandos básicos

1. Haga una secuencia de directorios de la siguiente manera: Primero cree una carpeta llamada `horoscopo`, dentro de ella cree dos más llamadas `chino` y `occidental`. En el directorio `chino` haga doce carpetas, cada una con el nombre de un animal distinto de tal horóscopo y dentro de cada una de ellas cree un archivo de texto con el mismo nombre de la carpeta y cuyo contenido sea los años correspondientes a tal signo entre 1900 y 2007. Ahora, en la carpeta `occidental` cree 12 archivos de texto con los nombres de cada signo zodiacal, que contengan las fechas del aõ que abarcan. En su `home` haga un directorio con su fecha de nacimiento, dentro de éste cree un *hard link* llamado `chino` dirigido al archivo de texto correspondiente a su signo de dicho horóscopo además cree un *simbolic link* llamado `occidental` al archivo de texto correspondiente a su signo.

2. Escriba en un archivo `txt` los comandos necesarios para crear la siguiente estructura de directorios en su `home`. En cada directorio cree un archivo de texto con el nombre del directorio y con extensión `.txt` en el que se describa brevemente el significado del nombre del directorio.

3. Usando la estructura de directorios que usted creó en 2 escriba en un archivo `txt` los comandos necesarios para:

   a) Borrar el directorio `Inoperativo`.

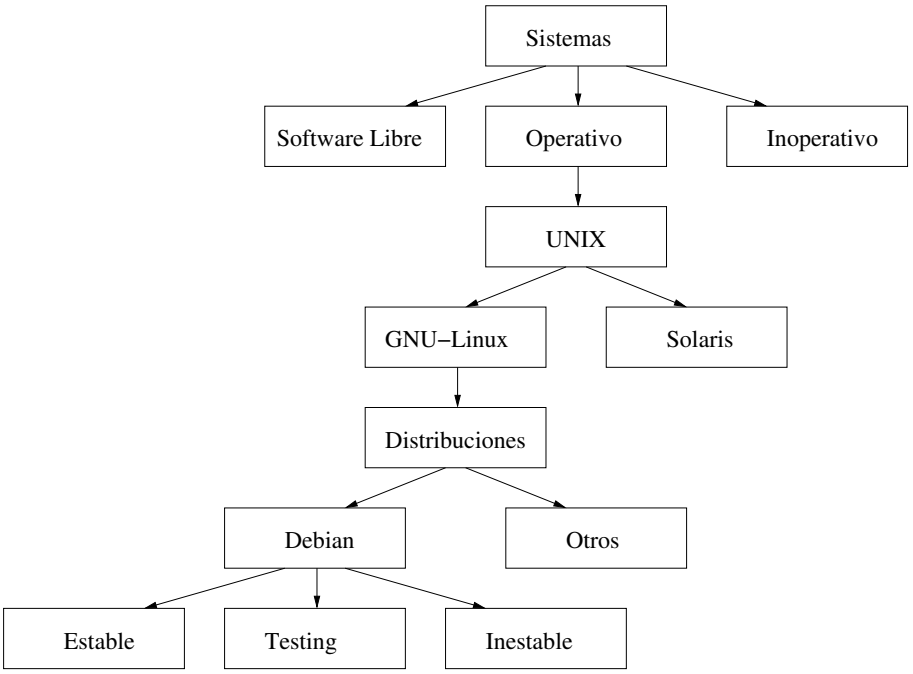
   b) Además, copie el directorio `Debian`, con sus subdirectorios, a un nuevo directorio llamado `debian` que cuelgue del mismo directorio `Distribuciones`, luego, en `debian` borre los directorios `Estable`, `Testing` e `Inestable` y entonces cree tres `link` simbólicos, uno llamado `lenny` que apunte a `Estable`, otro llamado `squeeze` que apunte a `Testing` y finalmente uno llamado `sid` que apunte a `Inestable`.

   c) Cambie el nombre del directorio `UNIX` a `unix`.

   d) Finalmente, deberá cambiarle los permisos a los directorios que cuelgan de Debian tal que permitan ejecución y lectura para el usuario y el resto del mundo.

   e) Dentro del directorio `Solaris` cambiarle los permisos al archivo `Solaris.txt` a sólo lectura para el usuario y su grupo y nada para el resto del mundo.

   f) Además, suponiendo que usted es `root` deberá cambiar el grupo del directorio `unix` a `users` y la propiedad del directorio `Sistema` a `root`.



4. Averigüe cómo podría cambiar su `prompt` de `nombre@maquina` a `ayudantes@geniales` "en colores", no olvide respaldar el archivo original.

5. Cree una secuencia de directorios anidados como sigue: primero su `nombre de pila`, el segundo su `apellido`, el tercero, `Licenciatura`, el cuarto `en`, y el quinto `fisica`, `matematica` o `exactas`. Ahora, desde su `home`, copie el primer y quinto directorio en su `home`, dejelos en su `home` y luego borre los directorios creados con las secuencias anteriores a excepción de las dos últimas.

6. Vea como redactar en una sola línea de comando lo siguiente: Cambiese de directorio al directorio `/usr`, vea que contiene (incluyendo los archivos ocultos) para luego ver detalladamente el contenido del mismo. Como está perdido pregunte al computador donde está, y como es "copuchento" (desde el mismo directorio), aproveche de preguntar cuánto disco libre queda (tome conciencia). Vuelva a su directorio `home` comente en cada paso si sube o baja del árbol del sistema.

7. Muévase al directorio `/proc` y lea el archivo `cpuinfo`. ¿Qué información contiene el archivo? Repita esto con otros archivos del directorio (al menos 3). Comente. Luego muévase al directorio `/etc` y lea el archivo `fstab`. ¿Qué información contiene el archivo? Luego dirijase a "alfven", monte el CD de la tarea que se encontrara en la sala y abra el archivo `yo_monte.txt` y explique su contenido. Retire el CD, y dejelo en su lugar para uso de otro compañero (si no puede sacarlo, no olvide desmontarlo).

8. Tome el archivo `yo_monte.txt` sacado del CD y copielo en el directorio con su nombre

en su `home` (el del ejercicio 5). Luego haga un *link* llamado `CD` que apunte hacia ese archivo en su directorio llamado `fisica`, `matematica` o `exacta` (también del ejercicio 5). Luego cambiele los permisos, cosa que usted pueda leer y escribir, el grupo solamente leer, y el resto no pueda hacer nada.

9. Haga `ssh` a `alfven`, monte el `cd` que dejaremos en la sala y copie el archivo que está en él, cambie los permisos y ejecútelo. Escriba todos los comandos que usó y las salidas.

10. En su computador, vea que usuarios están conectados, esta información reedirecciónela a un archivo llamado: `users@computador.txt`, luego agregue al final de ese archivo, los datos del usuario `mramirez`, comente. Averigüe como editar su `Plan` de tal manera que diga "`Yo aprendí a editar mi Plan`", y agréguelo al final del archivo `users@computador.txt`.

11. Nombre programas que sirvan para cumplir estas tareas:

    - Para ver archivos `pdf`.
    - Manejar hojas de cálculos.
    - Confeccionar gráficos.
    - Navegar en la *Web*.
    - Editar de imágenes.
    - Grabar `cd`'s.

12. De la página web de los ayudantes, descarge el archivo `prueba.tbz`. Descomprímalo y diga de qué tipo de archivo se trata. Luego, cámbiele los permisos de lectura, escritura y ejecución para el usuario, de ejecución para el grupo y de lectura y escritura para el resto del mundo.

13. El irresponsable Bill desea tener un "respaldo" de la tarea de su amigo Linus. Para esto, Bill sabe que Linus tiene su tarea en

    `/home/linus/cursos/2006/2sem/programacion/tareas/t02/tarea2-Linus.tbz`.

    Bill se encuentra en `/home/bill`, y desea dejar el archivo en `~/tarea2-bill.tar.bz2`. De tres formas de hacer esto en una sola línea.

14. Describa los comandos necesarios:

    a) Para crear la siguiente estructura de directorios: un directorio llamado `horoscopo` que contiene cuatro directorios llamados `agua`, `tierra`, `aire`, y `fuego`.

b) En cada uno de estos directorios cree tres archivos, uno para cada signo, que correspondan al elemento en cuestión.

c) Escriba en cada uno de los archivos que creó la fecha de inicio y término del signo correspondiente.

d) Cámbiese al directorio `horoscopo` y cree un *link* simbólico entre su signo y un archivo llamado `mi_signo`.

15. Bill recibe un mail su mejor amigo Linus el cual incluye los gratos recuerdos de sus vacaciones. Bill primero debe revisar su mail en un navegador web. Al abrir el mail, se encuentra con tres archivos adjuntos: una foto en formato `tiff`, un archivo de audio en `ogg` y un video en `mpeg`. Al final del texto del mail Linus escribe la direccion de su cuenta jabber para mantenerse en contacto. ¿Qué *software* debe tener Bill en su sistema GNU/Linux para lograr recordar los gratos momentos y mantenerse en contacto con su amigo?

16. Consultando los manuales del sistema, explique brevemente los comandos detallados a continuación

    - `less`
    - `tail`
    - `tee`
    - `g++`
    - `pidgin`
    - `amsn`
    - `gnuplot`
    - `bc`
    - `ascii`
    - `gunzip`
    - `top`
    - `touch`
    - `display`

17. Suponga que el *output* de un comando `ls` es

```
user@localhost:~/directorio $ ls
arbol.jpeg  arbol.jpg  arco.TXT  dani.bak.txt dani.bak.txt~
dani.txt.bak max.txt yo.tXt
```

¿Cuál es el *output* de los siguientes comandos?

a) `ls ar*`

b) `ls *?t`

c) `ls *.t?t*`

d) `ls *.j?*g`

18. Bill, al tratar de compilar un programa en lenguaje C++, se da cuenta que el compilador acusa no encontrar el archivo `cosas_importantes.h`. Tratando de resolver este problema, Bill decide buscar este archivo en su sistema local de archivos. ¿Cuál es el comando para saber si el archivo está en su sistema?

    Suponga que como resultado de la búsqueda Bill encontró el archivo mencionado en `/usr/include/mis_includes/`. Le consulta a uno de sus ayudantes de Programación y Métodos Numéricos el motivo de este error de compilación, el cual responde amablemente que el compilador lo busca en `/usr/include` y le suguiere que haga un `link` desde este directorio como `root` hacia el archivo encontrado. Describa el comando que le permite hacer esta acción y lograr, finalmente, compilar su programa.

    El error que se presentaba al compilar por primera vez fue resuelto. Sin embargo, al tratar de compilar el programa se presenta un nuevo error, esta vez la salida que nos entrega el compilador es

    ```
    programa.cc:2:18: error: cosas_importantes.h: Permission denied
    ```

    Claramente esto es un problema de permisos. Escriba el comando que nos permite averiguar los permisos de este archivo. La salida de este comando es

    ```
    -rwxr-x--- 1 root bin 5 2006-08-23 18:53 cosas_importantes.h
    ```

    Indique de quién es el archivo, a qué grupo pertenece, cuáles son los permisos del usuario, del grupo y del resto del mundo. Escriba el comando que le permitirá a este usuario cambiar los permisos (como `root`) y poder compilar el programa.

19. Escriba una línea de comando, que determine el tamaño y la ubicación del archivo más grande que tengan los usuarios de zeth. Dé su respuesta en unidades humanas.

20. Escriba la(s) línea(s) de comando(s) que cambie todos los archivos con extensión `.txt` a extensión `.dat`.

21. En un futuro cercano la sala de computación, que usted tiene el privilegio de usar, funcionará como un *cluster*. Imagine que este *cluster* ya está operacional y que usted está en el nodo 0 (zeth) y necesita saber que procesos consumen más recursos en los nodos del 1 al 15 (nodo01 al nodo15). Construya un comando que le permita hacer esto.

6.3. Filtros

1. Ingénieselas para crear un `archivo.dat` que contenga 2 columnas y 2 filas de números, y otro `archivo2.dat` con 3 columnas y 3 filas de números, para luego intercambiar (de alguna forma, usando filtros, los que quiera, y, ojalá, de forma original) éstos por letras de manera que logre escribir: `YO SOY`, con estos archivos. A continuación, pegue estos archivos y agreguele al mensaje su nombre, así obtendrá en la pantalla de su terminal: `YO SOY NOMBRE`. Adjunte todos estos archivos a su tarea.

2. Tome el archivo `ejer3.txt` (ubicado en el servidor `zeth`) y a través del uso de filtros, en una sola línea de comando, realice:

   a) Intercambie la segunda fila con la séptima

   b) Cambie los `9` por `A`

   c) Cambie en la séptima fila la cuarta aparición de `X` por s

   d) Cambie las `X` por `e`

   e) Cambie los `6` de la tercera fila por `a`

   f) Cambie en la secta fila los `8` por `i`

   g) Cambie `l` del octavo elemento (palabra) de la quinta fila por `a`

   h) Cambie `2` por `u`

   i) Cambie `4` por `s`

   j) Elimine todas las líneas que contengan `jua`

   Esto reedirecciónelo a un archivo llamado `resultado1.txt`. Luego de hacer esto, lea el archivo, y complete la instrucción que dice éste. Esta instrucción redireccionela también a `resultado2.txt`. Siga los pasos en orden, de lo contrario no le resultará.

3. Usando filtros, encuentre la velocidad de cada partícula para el archivo `velo3part.dat` que se encuentra en el servidor `zeth`. Además, encuentre su raíz cuadrática media (V_{rms}), el promedio, y la cuadrática media. Si las unidades son MKS, dé sus resultados (también) en CGS. Trate de interpretar lo que esta haciendo (física o matemáticamente; recuerde que el archivo contiene las velocidades por componentes de cada partícula chocando dentro de una caja).

4. Averigüe la cantidad de disco usado por dos de sus compañeros y uno de sus ayudantes. Si sumamos estas tres cantidades ¿Qué porcentaje de la capacidad total del `/home` de `zeth` utilizan?

5. Utilizando filtros, guarde en un archivo de texto el `path` de todos los archivos dentro del `/home` de `zeth` cuyo tamaño sea menor o igual a 12 Kbytes.

6. Cree el archivo `lista_07.txt`, con el siguiente texto

Alejandro Varas	7.0	6.5	6.0
Max Ramírez	7.0	6.0	6.5
Daniela Cornejo	6.0	7.0	6.5

Usando filtros, cree el archivo `lista_07_final.txt`, que ordene a los alumnos por orden alfabético y calcule sus promedios, de la forma siguiente

Alumno				Promedio
Daniela Cornejo	6.0	7.0	6.5	6.5
Max Ramírez	7.0	6.0	6.5	6.5
Alejandro Varas	7.0	6.5	6.0	6.5

Las palabras "Alumno" y "Promedio" no deben ir en el archivo final. Su(s) línea(s) de comando deben ser válidas para cualquier número de alumnos.

7. Un día como hoy, Bill recibe un correo electrónico con un archivo adjunto, llamado `mensaje.txt`, que contiene el siguiente texto:

```
L8UWAM %@?RDWAJ8LA AHZABAT AL@?RAC
L@?S %S@?W M8J@?R8S AYUDAWZ8S D8L MUWD@?!
PARA Y 8L R8SZ@? D8L MUWD@?.
8L PARA USUARI@?, 8L GRUP@?
8SZ@?S D8 DIR8CT@?RI@?S
D8SCRIBA LU8G@? L@?S P8RMIS@?S
LIWUX %AM@?
$ /media
$ /proc
$ /lib
L@?S D8 SIGUI8WZ8S DIR8CT@?RI@?S:
MI %D8 VIDA
8S %8SZ8 8L M8J@?R CURS@?
C@?MAWD@? D8L ls -l
8L 8SCRIBA @?UZPUT
```

Además, su emisor, le escribe las siguientes instrucciones en el mensaje:

a) Invierta el orden de las líneas, es decir, cambie la primera por la última, la segunda por la penúltima, etc.

b) Cambie en todo el mensaje los nï¿½meros `8` por letras `E`, las `W` por `N`, y los sï¿½mbolos `@?` por `O`.

c) Permute la primera columna del archivo con la segunda.

d) Cambie la primera aparición de la letra `Z` por `T` en todas las líneas del archivo.

e) Elimine todos las apariciones del símbolo `$` del archivo.

f) Elimine todas las líneas que contengan el símbolo `%`.

Bill, en medio de su desesperación, recurre a usted para lograr decifrar el mensaje. Escriba en un archivo `txt` todos los comandos necesarios, en el orden descrito, para poder decifrar el mensaje.

Una vez descifrado el mensaje realice las instrucciones que contiene y escriba su respuesta en un archivo llamado `archivo.txt`.

8. Explique que hacen los siguientes comandos:

   a) `grep -v ^$ archivo.txt`

   b) `sed -e '/^$/d' archivo.txt`

   c) `sed 'y/[1234]/[abcd]/' archivo.txt`

   d) `awk 'length > 10' archivo.txt`

   e) `awk '$1 != prev {print; prev = $1 }' archivo.txt`

   f) `awk '$2 > $1 {print i + ''1'';i++}' archivo.txt`

9. Unos alumnos de tercer año realizan un experimento con una fuente radioactiva de ^{137}Cs, y obtienen una columna de datos que corresponde al número de electrones emitidos. Para posteriormente poder graficarlos se necesita:

   - Eliminar los ceros que aparecen al principio de la lista de datos.
   - Encontrar el valor máximo de los datos obtenidos.
   - Dividir todos los datos por el máximo recién encontrado para normalizar la curva, es decir que los valores están entre 0 y 1.
   - Insertar una columna a la izquierda de la existente de modo que enumere los datos, esto es, al lado izquierdo del primer dato experimental debe haber un número 1, al lado del segundo un 2 y asi suscesivamente.

   Busque en la página de los ayudantes `http://zeth.ciencias.uchile.cl/mfm0/` el archivo `filtros_9.txt` y realice el procedimiento anteriomentre descrito utilizando filtros. Escriba detalladamente cada comando utilizado.

10. Suponga que usted tiene una lista en un archivo `recordario.txt` con la siguiente estructura

    ```
    Daniela Cornejo Jan 26 madaniela.cornejo@gmail.com
    Max Ramirez Feb 17 max.ramirez@gmail.com
    Jose Rogan Feb 18 jrogan@macul.ciencias.uchile.cl
    Alejandro Varas Oct 13 alejandro.varas@gmail.com
    ```

    Escriba la(s) línea(s) de comando(s) que le mande el archivo `felicitaciones.txt` al mail correspondiente si cuando se ejecute es el día de su cumpleaños.

6.4. *Scripts*

1. Nos piden ser `root` del servidor `tux.org` donde Bill es usuario con *username* `badbill`. Cada vez que Bill corre su programa `hagamos_dinero.bin` genera un gran archivo temporal con nombre `cuentas` en el directorio `/tmp` que deja al servidor casi sin espacio en disco. Nuestra misión, como `root`, es proteger al servidor. Para ello debemos escribir un *script* que verifique si Bill está conectado y si est corriendo su malévolo programa. De ser así nuestro *script* debe capturar el número del proceso y eliminarlo. Además, debe comprimir el archivo temporal con `bzip2` y moverlo al directorio `home` de Bill, mandándole un correo, a su cuenta en `tux.org`, diciéndole lo mal que funciona su programa y, finalmente, debe imprimir un aviso, en la impresora `orky`, dirigido a la comunidad de usuarios de `tux.org`, contándoles que hay poco espacio en el disco, otra vez, gracias a Bill y sus fechorías.

2. Haga un *script* que chequee el `hardware` de su máquina cada una hora, y si eventualmente encuentra alguna diferencia en éste, mande un mail adjuntando el archivo `diferencias.hw` a `pmaldona@zeth`.

3. Explique detalladamente en cada línea lo que hace el siguiente *script*:

    ```
    #!/bin/sh
    echo -e ‘‘introduzca una frase :  \c’’
    read n
    lg=‘echo $n | wc -c‘
    lg=‘expr $lg - 1‘
    while [ $lg -gt 0 ]
      do
        nr=’’$nr’’‘echo $n | cut -c $lg‘
        lg=‘expr $lg - 1‘
      done
    echo -e ‘‘$n \n$nr’’
    #$mmm?
    ```

Teniendo este *script* como ejemplo, haga otro que le devuelva la palabra que uno escriba en el terminal, escrita en "jerigonza", para luego devolverla escrita al revés.

4. Haga un *script* que cordialmente les pida el nombre que quieren ponerle al archivo que contendrá una columna con los números del 1 al 1000, y que le asigne el argumento a otro que tendrá los mismos números en desorden. Luego haga un menú que pregunte si uno los quiere pegar, verlos, ponerles la fecha al final, borrarlos, o salir del programa.

5. Escriba un *script* que les pida desde el terminal la posición y velocidad iniciales de una masa con movimiento de proyectil, además del intervalo de tiempo que quiere. A continuación, haga el cálculo para por lo menos 500 datos que enviará a un `archivo.dat` para graficarlo. Grafique dos como ejemplo, para así mostrarlos en la pantalla cuando halla terminado el cálculo; con un intervalo de separación entre ellos de 5 segundos.

6. Bill necesita una agenda para anotar los teléfonos de los pocos amigos que tiene. Los productos comerciales disponibles en el mercado son demasiado caros y no quiere gastar ni un sólo peso. Le pregunta a Linus si se puede hacer un *script* en `bash`, llamado por ejemplo `agenda.bash`, con las siguientes funcionalidades:

   - Para agregar un contacto uno use la siguiente sintaxis
     `$ ~/bin/agenda.bash A ‘‘Maria Daniela’’ 095555555 celular`
   - Para buscar un contacto uno debe use la siguiente sintaxis
     `$ ~/bin/agenda.bash B ‘‘maria daniela’’`
   - Para mostrar la lista completa uno use la siguiente sintaxis
     `$ ~/bin/agenda.bash M`
   - Para eliminar un contacto uno use la siguiente sintaxis
     `$ ~/bin/agenda.bash E Daniela`
   - Además tenga una opción de ayuda, la cual muestra todos los argumentos que se le pueden dar al *script*.
     `~/bin/agenda.bash H`

   Los contactos se guarden en un archivo `~/.contactos.txt`.

   El genial Linus se da cuenta, de inmediato, que el *script* puede tener problemas en los siguientes casos:

   - Si se trata de eliminar a alguien que no está en la lista.
   - Si se trata de eliminar un contacto que aparece más de una vez.
   - Si se ingresa una opción que no está definida, el *script* deberá avisar que no está definida y mostrar la ayuda, por ejemplo.

- Si...

Ayude a nuestro amigo Linus en su labor de generar una buena agenda, es decir, escriba un *script* con las funcionalidades pedidas por Bill, trate además de corregir los problemas visualizados por Linus. Sea riguroso en estudiar otros posible problemas, sea creativo en corregirlos.

7. Haga un `script` que calcule el doble factorial de un número natural n:

$$(2n+1)!! = 1 \cdot 3 \cdot 5 \cdots (2n+1)$$
$$(2n)!! = 2 \cdot 4 \cdot 6 \cdots (2n)$$

8. Haga un `script` que elimine todos los `tags` de un documento HTML, pruébelo con el archivo `prueba.html` que estará disponible en el `public` de `mfm0`[1].

9. Cree los archivos: `lista.txt`, con el siguiente texto

Alejandro Varas	7.0	6.5	7.0
Diego Guzmán	6.0	7.0	7.0
M. Daniela Cornejo	7.0	7.0	6.5
M. Carolina Guarachi	6.5	7.0	6.0
Víctor Araya	7.0	6.0	6.5
Nicole Miller	6.5	6.5	6.5
J. Ignacio Pinto	6.5	6.0	6.5

y `correos.txt`, con el siguiente texto

Alejandro Varas	avaras@jmail.com
Diego Guzmán	dguzman@jmail.com
M. Daniela Cornejo	madaniela@kmail.com
M. Carolina Guarachi	macarola@kmail.com
Víctor Araya	varaya@jmail.com
Nicole Miller	nmiller@kmail.com
J. Ignacio Pinto	ipinto@jmail.com

Escriba un archivo de comandos o *script*, que cree el archivo `listaFinal.txt`, conteniendo en la primeras columnas el apellido y luego el nombre de los alumnos ordenados alfabéticamente por apellido en otra columa debe incluir los respectivos promedios. Además, su archivo de comandos debe enviar un `e-mail` a cada uno de los alumno de la lista sólo con su nota final. Su archivo de comandos debe ser válido para cualquier número de alumnos. Note que los `e-mails` son ficticios.

[1] `https://zeth.ciencias.uchile.cl/mfm0`

10. Una salida típica del comando `ls -l` es la siguiente:

```
drwxr-xr-x 2 bill bill    4096 2008-07-19 10:58 fotos
-rw-r--r-- 1 bill bill   42781 2008-06-06 19:27 verlet.cc
-rwxrwxrwx 1 bill bill   44881 2008-08-31 19:27 verlet.py
lrwx------ 1 bill bill   35172 2008-07-28 19:44 mitarea -> /home/bill/tarea_linus.pdf
```

Escriba un *script* que reemplace la acción del comando `ls -l`, desplegando la información anterior de la siguiente manera:

```
Directorio 755 fotos
Archivo 644 verlet.cc
Archivo 777 verlet.py
Link 700 mitarea -> /home/bill/tarea_linus.pdf
```

Comente en el script lo que realiza cada línea del mismo.

11. Un archivo de datos llamado `datos.ini` tiene la forma:

```
#Archivo de datos
distancia 1.1
```

El programa `energia.bin` lee este archivo de datos al ejecutarse. Cuando corremos el programa con el comando `energia.bin < datos.ini` escribe en la pantalla la siguiente salida:

```
#Parameters file : gm2.ini
SPECIES pure
SYMBOL  Pd
NUMBER_OF_PARTICLES     2
CONF_IN_BANK    1
HOW_2_GENERATE  ONLY_FILES
POTENTIAL       Finnis-Sinclair
FILE_MULTI_XYZ  prueba.xyz
GM      EVALUATE
2
Configuracion 000 E= 2.93998082e+01 [eV] F= 6.64763359e+01 [eV/A] M= 0.0000 [muB]
Pd -0.55000 0.0000 0.0000
Pd 0.55000 0.0000 0.0000
```

Escriba un *script* que ejecute el programa `energia.bin` para diferentes valores de la distancia en el intervalo $[1, 3]$, y con una modificacion en el valor de la distancia de 0.1. Además, el *script* deberá filtrar la salida, de tal manera que se genere un archivo `datos_grafico.dat` el cual contendrá dos columnas: la primera columna donde estará la distancia entre las particulas, y la otra la energía del sistema. Esta salida será utilizada posteriormente en un gráfico de energía versus distancia. Por tanto, el archivo debe poseer el siguiente formato

```
#distancia Energia
1.0         35.00
1.1         29.39980
.
.
.
3.0         -0.005
```

12. Escriba un *script* que haga un respaldo de todos sus archivos, y que tenga el formato `user-respaldo_fecha.tar.bz2`.

13. Escriba un *script* que resuelva la ecuación $ax^2 + bx + c = 0$ para valores arbitrarios de a, b y c. Si los valores no pertenecen al conjunto de los reales, infórmeselo al usuario.

14. Haga un *script* que pida dos números. Si el primero es impar, calcule el seno de ese número; si es par, calcule su coseno; y si es cero, calcule la exponencial de uno. Calcule esto usando las representaciones en serie

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots \quad \operatorname{sen} x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots \quad \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots ,$$

usando el segundo número para determinar cuál es el último coeficiente a usar de la serie. Por supesto, informe al usuario el resultado del cálculo.

15. Escriba un *script* que lea todos los *e-mails* nuevos de su cuenta, luego vea si tienen archivos adjuntos. Si los tiene, que agregue este archivo adjunto a un archivo `tar`.

6.5. Gráfica y LaTeX

1. Genere una figura en el programa `xfig` que contenga como mínimo lo siguiente:

   a) Una figura importada.

   b) Dos figuras de las librerías de `xfig`.

   c) Un gráfico generado en `gnuplot`.

d) Un dibujo creado por usted en `xfig`.

En un documento `pdf` escriba los pasos necesarios para realizar la figura e inclúyala en éste.

2. Con los datos `algo.txt` (disponibles en la página web de los ayudantes)

   a) Haga un gráfico con los datos, luego importe (con el comando `import`) este gráfico con el nombre `primitivo.jpg`.

   b) Modifique el gráfico, colocándole como título `Ejemplo de regresión lineal`, en el eje de las abcisas `Tiempo [s]` y en el eje de las ordenadas `Posición [Ξ]`. Luego, una los puntos obtenidos y exporte este gráfico a formato `fig` con el nombre `original.fig`.

   c) Ahora, calcule la regresión lineal para estos datos. Grafique la regresión lineal obtenida y los datos experimentales en un sólo gráfico y luego exporte este resultado a un archivo `regresion.fig`.

   d) Por último, en un archivo en LaTeX, compilado en pdf, agregue una tabla resumen con los datos, todos los gráficos obtenidos y la regresión lineal del último gráfico, con la ecuación obtenida a un costado del mismo.

3. Reproduzca el siguiente texto en LaTeX

   Un punto material se mueve por el espacio bajo la influencia de una fuerza derivable de un potencial generalizado de la forma

   $$U(\mathbf{r}, v) = V(r) + \sigma \cdot \mathbf{L}, \tag{6.1}$$

   donde $\mathbf{r}$ es el vector de posición trazado desde un punto fijo, $\mathbf{L}$ es el momento cinético respecto a dicho punto y σ es un vector fijo en el espacio.

   a) Hallar los componentes de la fuerza que se ejerce sobre la partícula en coordenadas cartesianas y en coordenadas polares esféricas, basándose en la ecuación

   $$Q_j = -\frac{\partial U}{\partial q_j} + \frac{d}{dt}\left(\frac{\partial U}{\partial \dot{q}_j}\right).$$

   b) Demostrar que las componentes en los dos sistemas de coordenadas están relacionadas entre sí como en la ecuación

   $$Q_j = \sum_i \mathbf{F}_i \cdot \frac{\partial \mathbf{r}_i}{\partial q_j}.$$

c) Obtener las ecuaciones del movimiento en coordenadas polares esféricas.

4. Escriba dos páginas de los apuntes del curso de Fenomenología. No es necesario encerrar las ecuaciones en cajas. Las figuras, diagramas y gráficos deben incluirse[2].

   Adjunte los archivos `tex` y `pdf` y las figuras correspondientes.

5. De los apuntes del curso de Métodos de la Física Matemï¿½ática II `http://llacolen.ciencias.uchile.cl/~vmunoz/cursos/mfm2/mfm2.pdf`, haga una copia en LaTeX de dos páginas. Recuerde hacer sus figuras en `xfig`.

6. Escriba la página 51 del libro "Mathematical Methods for Physicists" de Arfken y Weber tal cual aparece en LaTeX, se exculpa si el número del capítulo es distinto, pero nada más.

7. Escriba en LaTeX la página 89 del mismo libro de la pregunta 6, sólo se perdonará que las fórmulas tengan otro número (en este caso).

8. Haga una copia libre de la sección 9 del capítulo 2 del libro "*Classical Electrodynamics*" de John J. Jackson, segunda edición. Adjunte la copia en LaTeX y la figura en `fig`. Los márgenes de la hoja **deben** ser: superior e inferior 3 cm, derecho e izquierdo 2.5 cm.

9. Redacte en LaTeX el siguiente problema: una masa M deslizándose por un plano inclinado con coeficiente de fricción μ_e. Invente las preguntas y posibles *hints*. Enuncie su problema en un `tex`, un `pdf` con una figura explicativa del problema en formato `fig`. Su problema debe ser solucionable.

10. Resuelva el problema que propuso en el ejercicio 9 y escriba la solución a éste en LaTeX, en el estilo Beamer.

11. Dada la siguiente serie de Fourier

$$f(x) = \frac{4}{\pi}\sum_{n=0}^{\infty}\frac{1}{2n+1}\,\text{sen}\left(\frac{(2n+1)\pi x}{L}\right)$$

    Grafique el valor obtenido de la función para distintos n en el intervalo [0,1].

    a) Grafique, en `gnuplot` las 6 primeras sumas parciales de esta serie en el intervalo $x \in [-10, 10]$ para $L = 1$.

    b) Realice un informe en LaTeX, comparando los resultados obtenidos.

[2] en `xfig` y/o `gnuplot`

12. Dada la expansión de Fourier para la función "diente de sierra"

$$f(x) = \frac{1}{2} - \frac{1}{\pi}\sum_{n=1}^{\infty}\frac{1}{n}\operatorname{sen}\left(\frac{n\pi x}{L}\right) \tag{6.2}$$

   a) Grafique, en `gnuplot` las 6 primeras sumas parciales de esta serie en el intervalo $x \in [-10, 10]$ para $L = 1$.

   b) Realice un informe en LaTeX, comparando los resultados obtenidos.

6.6. Introducción a la Programación

1. De la lista de lenguajes que se nombra a continuación, clasifique cuáles son interpretados y cuáles son compilados.

   - Fortran
   - Pascal
   - Cobol
   - Ruby
   - Python
   - Perl
   - C++
   - C
   - Ada
   - Eiffel
   - Basic
   - Oberon

2. Investigue en qué lenguajes están escritos los siguientes programas y si corresponden a lenguajes interpretados o compilados:

   a)
```
#!/usr/bin/wish -f

wm title . "Hello world!"

frame .h -borderwidth 2
frame .q -borderwidth 2
button .h.hello -text "Hello world" \
```

```
        -command "puts stdout \"Hello world!\"" -cursor gumby
button .q.quit -text "Quit" -command exit -cursor pirate

pack .h -side left
pack .q -side right
pack .h.hello
pack .q.quit
```

b)
```
with Text_Io; use Text_Io;

procedure hello is
begin
  put ("Hello world!");
end hello;
```

c)
```
c
c   Hello, world.
c
      Program Hello

      implicit none
      write(*,10)
   10 format('Hello, world.')
      END
```

d)
```
for i in 1..1
    puts "Hello World!"
end
```

e)
```
#include <iostream>

int main(){
   std::cout << "Hello World!" << std::endl;
   return 0;
}
```

6.7. Python

1. Reconozca que acciones hacen las siguientes líneas

```
#!/usr/bin/env python

#a = input(``Hola Mundo'')
```

```
while 1>0:
    print ‘‘Hola querido alumno’’
```

2. Escriba un programa en `Python` que le pida su nombre y a cambio le responda amablemente "Buenas tardes, señor", "Buenas tardes, señora" o "Buenas tardes, señorita", según sea el caso, y a continuación su nombre.

3. Confeccione un programa en `Python` que le diga si un número es par o impar.

4. Escriba un programa en `Python` que imprima el mayor, el menor, la suma y la media aritmética de diez números aceptados por teclado.

5. Escriba un programa en `Python` que le pida dos números desde el terminal, luego de instrucciones al usuario, es decir, despliegue un menú que de las opciones de dividir los números (donde en esta opción además diga si es aceptable el cero como divisor), sumarlos, restarlos, incrementarlos en 2, 1, y 10 (usando las abreviaciones aprendidas en clases), y multiplicarlos; al final de cada operación de el resultado correspondiente. Incluya también la opción de salir del programa en el menú (use `if` para el menú).

6. Escriba un programa en `Python` que le pida al usuario el día, mes y año de su nacimiento y, además, la fecha actual. El programa debe devolver:

   a) La edad del usuario, en años.

   b) El tiempo de vida del usuario, en meses.

   c) El tiempo de vida del usuario, en días.

   d) El signo del usuario en el horóscopo occidental.

   e) El signo del usuario en el horóscopo oriental.

7. Escriba un programa en `Python`, que le pida al usuario una fecha: día, mes y año, y que a partir de ella evalúe:

   - Si el año dado es o no bisiesto.
   - El número de días transcurridos desde el principio del año.
   - El número de días que faltan para la próxima Navidad.

8. Escriba un programa en `Python` que le pida una cantidad, un entero positivo, y le devuelva el número y denominaciones de los billetes y monedas para dar cuenta de esa cantidad, minimize el número de billetes y monedas.

9. Confeccione un programa en `Python` que calcule el máximo común divisor entre dos números enteros.

10. Confeccione un *software* en `Python` que solicite dos números enteros al usuario n, m. Luego, genere un tercer número entero al azar en el intervalo $[n, m + 10]$, con las funciones intrínsecas de `Python`. Finalmente, verifique si este trío de números pueden ser las medidas de un triángulo rectángulo.

11. Confeccione un programa en `Python` que solicite tres puntos en $\mathbb{R}^3$ y que retorne el área del triángulo que forman éstos tres puntos.

12. Confeccione un programa en `Python` que cree una lista con n ingredientes de una receta de cocina, ingresables por el usuario.

13. Usando listas, haga un programa en `Python` que le pida al usuario ingresar dos vectores y luego despliegue un menu con las opciones sumar, restar, producto punto y producto cruz. Luego mande lo pedido a pantalla (si eligió suma o resta la salida debe parecer un vector).

14. Escriba un programa que pida el ingreso de tres puntos en $\mathbb{R}^3$ y con ellos calcule un vector unitario perpendicular al plano que contiene los tres puntos. Si los tres puntos ingresados fueran colineales, el programa debe avisar al usuario y no seguir calculando.

15. Utilizando listas, haga un programa que inicialice una matriz de dos por dos. Luego verifique si el determinante es distinto de cero, si es distinto de cero, calcule la inversa de la matriz, si no lo es calcule el producto consigo misma.

16. Usando listas, confeccione un programa en `Python` que cree dos matrices de 3 columnas y 3 filas. Luego, rellene estos espacios con números al azar entre 0 y 60. Por último, imprima en pantalla sus matrices, la suma de ellas y el determinante de cada una.

17. Utilizando listas escriba un programa en `Python` tal que el usuario ingrese una oración, y el computador la devuelva escrita al revés.

18. Cree los archivos `A.txt` y `B.txt` los cuales deben contener una matriz de números reales y de tamaño 3×3 cada uno. Luego escriba un programa en `Python` que lea estos archivos y calcule la suma, la multiplicación y la inversa de cada matriz. Su programa debe escribir estos resultados en los archivos `sumaAB.txt`, `multiplicacionAB.txt`, `inversaA.txt` e `inversaB.txt` respectivamente. Si alguna matriz no tiene inversa debe informarlo en su archivo.

19. Escriba un programa que dada una base cualquiera de $\mathbb{R}^4$ la someta al proceso de ortonormalización de Gram-Schmidt y la despliega en pantalla. Debe usar listas.

20. Escriba el programa llamado `gato` que funcione de la misma manera que el comando `cat` del sistema. Es decir, cuando lo invoque con un nombre de archivo este despliegue en la pantalla el contenido de dicho archivo. Agreguele una opción adicional, invocada con `-r`, para la cual despliegue el archivo de atrás para adelante, tal como lo hace el comando `tac`.

21. Jugando Ludo: Escriba un programa en `Python` que juegue ludo, para esto utilize los programas de azar que son incluidas en los módulos de `Python`. La idea es que simule el lanzamiento de un dado, y como ud. juega contra el computador, el número más grande parte. La meta estará en el casillero 50, y en (por lo menos) cinco casilleros se vuelve a la partida, para que así, el primero en llegar a 50, gana.

22. Escriba un programa en `Python` que acepten que le ingresen un cierto número de pares (x, y) y luego calcule la regresión lineal. Use la ecuación $y = mx + b$ donde

$$m = \frac{\sum xy - \bar{y}\sum x}{\sum xx - \bar{x}\sum x} \ ,$$

y $\bar{x}$ corresponde al promedio de las x.

23. En este problema Bill necesita una agenda para anotar los teléfonos de los pocos amigos que tiene. Los productos comerciales disponibles en el mercado no lo satisfacen y no quiere gastar ni un sólo peso. Le pregunta a Linus si se puede hacer un programa en `Python`, llamado por ejemplo `agenda`, con las siguientes funcionalidades:

    - Que permita agregar un contacto: `"Carola Paz" 0911111111 celular`.
    - Que permita buscar un contacto.
    - Que muestre la lista completa .
    - Que elimine un contacto.
    - Además, tenga una opción de ayuda, la cual muestra todos los argumentos que se le pueden dar al programa. `agenda -h`

    Los contactos deben guardarse en un archivo `~/.contactos.txt` en la raiz de su `home`.

    El genial Linus se da cuenta, de inmediato, que el programa puede tener problemas en los siguientes casos:

    - Si se trata de eliminar a alguien que no está en la lista.
    - Si se trata de eliminar un contacto que aparece más de una vez.
    - Si se ingresa una opción que no está definida, el programa deberá avisar que no está definida y mostrar la ayuda, por ejemplo.
    - Si...

Ayude a nuestro amigo Linus en su labor de generar una buena agenda, es decir, escriba un programa con las funcionalidades pedidas por Bill, trate además de corregir los problemas visualizados por Linus. Sea riguroso en estudiar otros posible problemas, sea creativo en corregirlos.

24. Escriba un programa en `Python` que le pida al usuario un numéro entero positivo menor que un millón (10^6) y devuelva a pantalla el número en palabras.

25. Implemente una función combinatoria que devuelva el número de subconjuntos (desordenados) de k elementos de un conjunto de n elementos. Use una función factorial.

26. Escriba un programa en `Python` que pida un número al usuario. De acuerdo a si este número es par, evalúe la expansión en serie de la función coseno para el número ingresado con un error menor que 10^{-5}; si el número es impar, evalúe la expansión en serie de la función seno para el número ingresado con un error menor que 10^{-5} y finalmente si el número no es ni par ni impar, evalúe la expansión en serie de la función exponencial de menos el número ingresado con un error menor que 10^{-5}.

27. Escriba un programa en `Python` que pida un número N y evalúe

$$\sum_{i=1}^{N} n \ , \sum_{i=1}^{N} n^2 \ , \sum_{i=1}^{N} n^3 \ , \sum_{i=2}^{2N} \log(n) \ .$$

28. Escriba un programa que explique **brevemente** y calcule el doble factorial de un número natural positivo, es decir,

$$\begin{aligned} (2n+1)!! &= 1 \times 3 \times 5 \times \cdots (2n+1) \ , \\ (2n)!! &= 2 \times 4 \times 6 \times \cdots (2n) \ . \end{aligned}$$

Si el usuario no ingresa un número natural positivo, hágaselo notar, amablemente.

29. Escriba un nuevo programa que repita el cálculo de doble factorial, esta vez usando una función en forma recursiva, es decir, que la función se llame a sí misma.

30. Escriba un programa que evalúe la suma infinita,

$$S(x) = \sum_{\nu=0}^{\infty} x^{\nu} \ ,$$

para un $|x| < 1$ y tal que el usuario ingrese la precisión deseada. Si el número x ingresado no cumple con la condición $|x| < 1$, el programa debe informarlo.

31. Sea la función $f(x)$ definida por

$$f(x) = \begin{cases} -x^2 + x + 1 & x < 0 \\ \ln(x+1) & x > 0 \end{cases}$$

Escriba un programa en `Python` tal que:

a) Calcule valor de la integral superior de Riemann en el intervalo $[-5, 5]$, de manera tal que el valor de la integral tenga un error menor al pedido por el usuario.

b) Calcule valor de la integral inferior de Riemann en el intervalo $[-5, 5]$, de manera tal que el valor de la integral tenga un error menor al pedido por el usuario.

En este ejercicio, el error pedido por el usuario ΔE corresponde a

$$\Delta E = |\text{Integral Analítica} - \text{Integral numérica}|$$

32. En este ejercicio calcularemos numéricamente la siguiente integral

$$I = \int_0^1 4\sqrt{1-x^2}\, dx$$

mediante el método del trapecio, es decir, aproximaremos la integral por la suma finita

$$I = \int_a^b f(x)\, dx \approx \frac{1}{2}h(f_0 + f_N) + h\sum_{i=1}^{N-1} f_i \ ,$$

donde $f_0 = f(x_0 = a)$, $f_N = f(x_N = b)$, y $f_i = f(x_i)$, donde $x_i = a + ih$, con

$$h = \frac{b-a}{N} \ .$$

N es el número de particiones.

Usted debe hacer un programa que implemente este algoritmo. El usuario debe proporcionar N al programa y el integrando debe ser implementado como una función. Identifique el resultado y estudie la convergencia de éste en función de N. Haga este estudio en un archivo separado de la fuente de su programa, es decir, en un archivo `estudio.txt` incluya dos columnas con los resultados para $N = 10, 100, 1000, 5000, 50000, 100000$.

33. Confeccione un programa en `Python` que solicite un número. Con este número, evalúe la función seno usando su expansión en Taylor centrada en cero para el número ingresado con un error menor que 10^{-5}. Use las propiedades de periodicidad y simetría de esta función, para que el argumento de la evaluación esté en $[0, \pi/2]$.

Extra crédito: Reduzca el argumento a $[0, \pi/4]$.

34. Confeccione un programa en `Python` que resuelva la ecuación

$$ax^2 + bx + c = 0$$

con a, b, y c parámetros ingresables (reales).

35. Confeccione un *software* en `Python` que resuelva la ecuación cuadrática $ax^2+bx+c = 0$ para valores de a, b y c arbitrarios. A diferencia de tareas anteriores, calcule tanto las raíces reales como complejas.

36. Escriba un programa en `Python` que, usando la recursión

$$\Gamma(z+1) = z\Gamma(z) \qquad z \in \mathbb{N},$$

calcule $\Gamma(z)$ para cualquier valor de z.

37. Programe una función en `Python` que calcule el término n de la sucesión de Fibonacci, donde n es ingresado por el usuario. Además, dado un número m verifique si este número es un número perteneciente a esta sucesión, e informe al usuario la posición en la sucesión de este número.

38. Escriba un programa que calcule, a través de una función, la siguiente sumatoria

$$S = \sum_{i=1}^{M} i^n$$

con M y n proporcionados por el usuario.

39. Escriba un programa en `Python` que, usando la representación de las funciones de Bessel

$$J_\nu(x) = \sum_{s=0}^{\infty} \frac{(-1)^s}{s!(s+\nu)!} \left(\frac{x}{2}\right)^{\nu+2s}$$

calcule la función $J_1(x)$ para cualquier valor de x.

40. Escriba un programa que evalúe las sumas infinitas,

$$\sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n}\ , \qquad \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{2n+1}\ , \qquad \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n^2}\ ,$$

el usuario debe ingresar la precisión deseada.

41. Evalue en un programa `Python`, para diferentes N, ingresados por el usuario, la convergencia de la siguiente expresión:

$$\frac{\pi}{2} = \sum_{k=0}^{N} \frac{(2k-1)!!}{(2k+1)(2k)!!}$$

42. Escriba un programa en `Python` que le permita analizar, de manera numérica, la convergencia de la serie

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

43. Evalue la raíz n-ésima para cualquier número ingresado por el usuario con un programa en `Python` utilizando funciones y `while`. Use el algoritmo visto en clases.

44. Usando el método de *Newton-Raphson* visto en clases, haga un programa en `Python` que calcule las soluciones de la ecuación trascendente

$$x \tan x = x \ .$$

Grafique las soluciones en el intervalo $[0, 8\pi]$.

45. Dada la expansión de Fourier para la función "diente de sierra"

$$f(x) = \frac{1}{2} - \frac{1}{\pi} \sum_{n=1}^{\infty} \frac{1}{n} \operatorname{sen}\left(\frac{n\pi x}{L}\right) \tag{6.3}$$

a) Grafique, en `gnuplot` las 6 primeras sumas parciales de esta serie en el intervalo $x \in [-10, 10]$ para $L = 1$.

b) Programe la serie (6.3) para los primeros 20 términos en `python`. Luego, genere un archivo `datos.dat` en el mismo intervalo de la parte (a) y grafique su resultado en `xmgrace`.

c) Realice un informe en LaTeX, comparando ambos resultados.

46. Desde la página web de los ayudantes descargue el archivo `regresion.txt`. Luego, aplicando las fórmulas de regresión lineal que usted conoce, desarrolle un programa en `Python` que encuentre la pendiente y el intercepto a éstos datos. Por supuesto, el archivo `input` puede ser cualquier archivo de texto.

47. El código morse es una forma de comunicarse a grandes distancias. Éste se basa en combinaciones de puntos y rayas. Desarrolle un programa en `Python` que lea desde un archivo de texto todas las palabras contenidas en él y que devuelva un `archivo_morse`

con las letras correspondientes pero en alfabeto morse. Tenga precaución con la separación de cada letra, ya que su traducción puede volverse inteligible.

48. Escriba un programa que repita el cálculo del ejercicio 32, integración numérica con la regla del trapecio. Esta vez el programa debe leer desde un archivo `datos.txt` los diferentes valores de N pedidos en el ejercicio anterior y la tabla escrita en el archivo `estudio.txt` en la prueba, esta vez debe ser generada y escrita desde el programa en el archivo `estudio.txt`.

49. Copie los archivos `velinicial.dat` y `velfinal.dat` que contiene 1000 datos de las velocidades de 1000 partículas, con respecto a cada una de sus coordenadas. Realice un programa que tome éstos datos y le calcule una distribución de velocidades (la cantidad de partículas que llevan una cierta velocidad), enviándo estas distribuciones a dos archivos diferentes. De un rango de error en las velocidades para poder graficar sus distribuciónes, y comente ambos graficos.

50. Escriba un programa que acepte dos argumentos desde la línea de comando. El primer argumento corresponde al monto y el segundo un porcentaje. El programa calculará el porcentaje pedido del monto dado y lo escribirá en pantalla.

51. Reescriba el programa del ejercicio 19, donde se pedia que dada una base cualquiera de $\mathbb{R}^4$ la someta al proceso de ortonormalización de Gram-Schmidt y la despliege en pantalla. Esta vez use una clase, hecha por usted, de vectores en $\mathbb{R}^4$.

52. Cree una clase de Alumnos de la Universidad de Chile, con Facultad, carrera, año de ingreso, RUT, número de TUCH y lista de notas (30 notas). A continuación, cree un programa que lea desde el archivo `alumnos_masivos.dat` (disponible en la página de los ayudantes) todos los alumnos existentes en la Universidad. Luego, elija un alumno al azar (de la carrera con más alumnos y de la Facultad con más carreras) y arroje al `stdout` sus datos.

53. Escriba una clase de vectores de dimension 3, sobrecargando el producto, la suma, la resta y creando una función para el producto punto. Además, haga un constructor de copia, y sobrecargue la salida y entrada. El programa `main` deberá recibir dos vectores y dar las opciones de sumarlos, restarlos, multiplicarlos (los dos productos), y el modulo, para luego, desplegarlos en pantalla.

54. Escriba un programa, usando el desarrollo del ejercicio 53, genere n vectores al azar en dos dimensiones, y cuyas componentes estén entre 1 y -1, usando la función de azar disponible en `Python`. Para cada vector determine si su módulo es mayor o menor que uno, si es mayor que uno no lo use, si es menor o igual a uno cuéntelo y escriba sus componentes en un archivo. Luego, haga el cociente entre el número contado y el número total, multiplíquelo por cuatro (que es el área del cuadrado) e imprímalo.

Usando `Tkinter`, grafique los vectores que cumplen la condición pedida. ¿Le parece familiar el número obtenido? Estudie este número en función de n.

55. Escriba un programa con clases de pares ordenados tal que el usuario ingrese tres puntos en el espacio de dos dimensiones y el computador diga si estos puntos conforman un triángulo isóseles, escaleno o equilátero. Hecho esto, que diga si los pares ordenados conforman un ángulo agudo, obtuso o recto.

56. Programe una agenda, es decir, que todos los dias el computador le diga su itinerario a realizar. Puede completar hasta donde quiera el programa; le puede ser útil el comando `cron`.

57. Escriba un programa con clases de matrices de dimensión 3×3 que realize las siguientes operaciones : sume las matrices, las multiplique, calcule las trazas, y sus determinantes.

58. Cree una clase de cargas eléctricas en un espacio de dos dimensiones con las siguientes características:

    - Los elementos deben ser de la forma `carga(posicionx,posiciony,modulo,signo)` donde `posicionx` y `posiciony` representan las coordenadas en $\hat{x}$ e $\hat{y}$ de la posición en el plano, `modulo` es el valor del módulo de la carga, es decir, $|q|$ y `signo` es el signo de la carga que puede ser ± 1.
    - Su clase debe tener las funciones `F(carga1,carga2)` que es la fuerza que siente la `carga1` debido a la `carga2` y la función `U(carga1,carga2)` que es la energía del sistema constituido por estas dos cargas.

    Además, haga un programa que utilice su clase con un sistema de 3 cargas.

59. Implemente una clase de números racionales y un pequeño programa en que muestre cada una de las características implementadas.

60. La representación integral de la funciones de Bessel de orden entero es:

$$J_n(x) = \frac{1}{\pi} \int_0^\pi \cos(n\theta - x \operatorname{sen} \theta)\, d\theta \ .$$

    Además, las primeras dos funciones de Bessel satisfacen

$$J_0'(x) = -J_1(x) \ .$$

    Encuentre los tres primeros ceros de $J_0(x)$ para $x > 0$ con una precisión de 10^{-4}. Use la representación integral dada para evaluar las funciones de Bessel. Para asegurarse que su semilla y luego su resultado sea correcto, se sugiere graficar la función $J_0(x)$ en su graficador favorito. Exponga sus resultados en un archivo LaTeX. Adjunte su programa.

61. a) Encuentre analíticamente las soluciones de la ecuación $z^3 = 1$, y demuestre que éstas constituyen los vértices de un triángulo equilátero en el plano complejo.

b) Utilizando el método de Newton, encuentre las raíces de este problema numéricamente. Observe que, para distintas elecciones de las semillas, se converge a una de las tres raíces halladas en (a).

c) Grafique en el plano complejo las "cuencas de atracción" para cada raíz, constituidas por todas las semillas que convergen a una determinada raíz. Considere las semillas en $-2 < x < 2$ e $-2 < y < 2$, y grafique las tres cuencas de atracción con distinto color. Observe la frontera entre dos cuencas. Comente.

62. *a*) El potencial de Lennard-Jones entre dos partículas separadas una distancia r es el siguiente:

$$U(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

Grafique el potencial en función de r, con $r \in [1, 4]$ y utilizando $\varepsilon = \sigma = 1$. Realice un gráfico en formato `eps` incluyendo título y etiquetas en los ejes.

b) Nos interesa encontrar la distancia de equilibrio de dos partículas, es decir, el mínimo de este potencial o lo que es equivalente, encontrar el cero de la derivada.

Grafique la derivada analítica en función de r con $r \in [1, 4]$, realice un gráfico en formato `png` incluyendo título y etiquetas en los ejes.

c) Escriba un programa en `Python` que use el algoritmo de Newton-Raphson[3] sobre la derivada[4]

del potencial para encontrar donde ésta se anula. Use como valor inicial 0,5 y una precisión de 10^{-7}.

d) Grafique la sucesión de valores de r en función de las iteraciones que converge al cero de la derivada. Realice un gráfico en formato `pdf` incluyendo título y etiquetas en los ejes.

---

[3] El algoritmo para encontrar un cero de $f(x)$ por Newton-Raphson se escribe como:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

[4] Para la derivada de $f(x)$ use la forma numérica:

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

con h del orden de 10^{-7}

Apéndice A

Transferencia a diskettes.

La filosofía de diferentes unidades (A:, B:,...) difiere de la estructura única del sistema de archivos que existe en UNIX. Son varias las alternativas que existen para la transferencia de información a diskette.

- Una posibilidad es disponer de una máquina WIN9X con ftp instalado y acceso a red. Empleando dicha aplicación se pueden intercambiar archivos entre un sistema y el otro.

- Existe un conjunto de comandos llamados `mtools` disponible en multitud plataformas, que permiten el acceso a diskettes en formato WIN9X de una forma muy eficiente.

  `mdir a:` Muestra el contenido de un diskette en `a:`.

  `mcopy file a:` Copia el archivo `file` del sistema de archivos UNIX en un diskette en `a:`.

  `mcopy a:file file` Copia el archivo `a:file` del diskette en el sistema de archivos UNIX con el nombre `file`.

  `mdel a:file` Borra el archivo `a:file` del diskette.

  Con `a:` nos referimos a la primera diskettera `/dev/fd0` y luego al archivo que se encuentra en el diskette. Su nombre se compone de `a:filename`. Si se desea emplear el caracter comodín para un conjunto de archivos del diskette, estos deben rodearse de dobles comillas para evitar la actuación del *shell* (p.e. `mcopy ''a:*.dat''`). La opción `-t` realiza la conversión necesaria entre UNIX y WIN9X, que se debe realizar **sólo** en archivos de texto.

- Una alternativa final es montar el dispositivo `/dev/fd0` en algún directorio, típicamente `/floppy`, considerando el tipo especial de sistema de archivos que posee `vfat` y luego copiar y borrar usando comandos UNIX. Esta forma suele estar restringida sólo a `root`, el comando:

  ```
  mount -t vfat /dev/fd0 /floppy
  ```

no puede ser dado por un usuario. Sin embargo, el sistema aceptará el comando `mount /floppy` de parte del usuario. Una vez terminado el trabajo con el floppy éste debe ser desmontado, antes de sacarlo, mediante el comando: `umount /floppy`.

Apéndice B

Las shells csh y tcsh.

Son dos de los Shells interactivos más empleados. Una de las principales ventajas de `tcsh` es que permite la edición de la línea de comandos, y el acceso a la historia de órdenes usando las teclas de cursores.[1]

B.1. Comandos propios.

Los comandos propios o intrínsecos, *Built-In Commands*, son aquéllos que proporciona el propio *shell* [2].

`alias name def`
Asigna el nombre `name` al comando `def`.

`history`
Muestra las últimas órdenes introducidas en el *shell*. Algunos comandos relacionados con el *Command history* son:

- `!!`
  Repite la última orden.
- `!n`
  Repite la orden n-ésima.
- `!string`
  Repite la orden más reciente que empiece por la cadena `string`.
- `!?string`
  Repite la orden más reciente que contenga la cadena `string`.

[1] `bash` también lo permite.

[2] A diferencia de los comandos que provienen de un ejecutable situado en alguno de los directorios de la variable `PATH`.

- `^str1^str2` o `!!:s/str1/str2/`
  (*substitute*) Repite la última orden reemplanzando la primera ocurrencia de la cadena `str1` por la cadena `str2`.
- `!!:gs/str1/str2/`
  (*global substitute*) Repite la última orden reemplazando todas las ocurrencias de la cadena `str1` por la cadena `str2`.
- `!$`
  Es el último argumento de la orden anterior que se haya tecleado.

`repeat count command`
Repite `count` veces el comando `command`.

`rehash`
Rehace la tabla de comandos (*hash table*).

`set variable = VALUE`
Asigna el valor de una variable del *shell*.

`set`
Muestra el valor de todas las variables.

`setenv VARIABLE VALUE`
Permite asignar el valor de una variable de entorno.

`source file`
Ejecuta las órdenes del fichero `file` en el *shell* actual.

`unset variable`
Borra la asignación del valor de una variable del *shell*.

`unsetenv VARIABLE VALUE`
Borra la asignación del valor de una variable de entorno.

`umask value`
Asigna la máscara para los permisos por omisión.

`unalias name`
Elimina un alias asignado.

B.2. Variables propias del shell.

Existe un conjunto de variables denominadas *shell variables*, que permiten modificar el funcionamiento del *shell*.

`filec` (*FILE Completion*)

Es una variable *toggle* que permite que el *shell* complete automáticamente el nombre de un archivo o un directorio[3]. Para ello, si el usuario introduce sólo unos cuantos caracteres de un archivo y pulsa el `TAB`, el *shell* completa dicho nombre. Si sólo existe una posibilidad, el completado es total y el *shell* deja un espacio tras el nombre. En caso contrario hace sonar un pitido. Pulsando `Ctrl-D` el *shell* muestra las formas existentes para completar.

`prompt`

Es una variable de cadena que contiene el texto que aparece al principio de la línea de comandos.

`savehist`

Permite definir el número de órdenes que se desea almacenar al abandonar el *shell*. Esto permite recordar las órdenes que se ejecutaron en la sesión anterior.

[3] `bash` permite no sólo completar ficheros/directorios sino también comandos.

Apéndice C

Editores tipo emacs.

Los editores tipo emacs se parecen mucho y en su mayoría sus comandos son los mismos. Para ejemplificar este tipo de editores nos centraremos en `XEmacs`, pero los comandos y descripciones se aplican casi por igual a todos ellos. Los editores tipo emacs constan de tres zonas:

- La zona de edición: donde aparece el texto que está siendo editado y que ocupa la mayor parte de la pantalla.
- La zona de información: es una barra que esta situada en la penúltima línea de la pantalla.
- La zona de introducción de datos: es la última línea de la pantalla.

Emacs es un editor que permite la edición visual de un archivo (en contraste con el modo de edición de `vi`). El texto se agrega o modifica en la zona de edición, usando las teclas disponibles en el teclado.

Además, existen una serie de comandos disponibles para asistir en esta tarea.

La mayoría de los comandos de emacs se realizan empleando la tecla de `CONTROL` o la tecla `META`[1]. Emplearemos la nomenclatura: `C-key` para indicar que la tecla `key` debe de ser pulsada junto con `CONTROL` y `M-key` para indicar que la tecla `META` debe de ser pulsada junto a `key`. En este último caso NO es necesario pulsar simultáneamente las teclas `ESC` y `key`, pudiendo pulsarse secuencialmente `ESC` y luego `key`, sin embargo, si se usa `ALT` como `META` deben ser pulsadas simultáneamente. Observemos que en un teclado normal hay unos 50 caracteres (letras y números). Usando `SHIFT` se agregan otros 50. Así, usando `CONTROL` y `META`, hay unos $50 \cdot 4 = 200$ comandos disponibles. Además, existen comandos especiales llamados *prefijos*, que modifican el comando siguiente. Por ejemplo, `C-x` es un prefijo, y si `C-s` es un comando (de búsqueda en este caso), `C-x C-s` es otro (grabar archivo). Así, a través de un prefijo, se duplican el número de comandos disponibles sólo con el teclado, hasta llegar a unos $200 \cdot 2 = 400$ comandos en total.

[1] Dado que la mayoría de los teclados actuales no poseen la tecla `META` se emplea ya sea `ESC` o `ALT`.

Aparte de estos comandos accesibles por teclas, algunos de los cuales comentaremos a continuación, existen comandos que es posible ejecutar por nombre, haciendo así el número de comandos disponibles virtualmente infinito.

Revisemos los comandos más usuales, ordenados por tópico.

Abortar y deshacer

En cualquier momento, es posible abortar la operación en curso, o deshacer un comando indeseado:

C-g	abortar
C-x u	deshacer

Archivos

C-x C-f	cargar archivo
C-x i	insertar archivo
C-x C-s	grabar archivo
C-x C-w	grabar con nombre
C-x C-c	salir

Ventanas

`Emacs` permite dividir la pantalla en varias ventanas. En cada ventana se puede editar texto e ingresar comandos independientemente. Esto es útil en dos situaciones: a) si necesitamos editar un solo archivo, pero necesitamos ver su contenido en dos posiciones distintas (por ejemplo, el comienzo y el final de archivos muy grandes); y b) si necesitamos editar o ver varios archivos simultáneamente. Naturalmente, aunque son independientes, sólo es posible editar un archivo a la vez. A la ventana en la cual se encuentra el cursor en un momento dado le llamamos la "ventana actual".

C-x 2	dividir ventana actual en 2 partes, con línea horizontal
C-x 3	dividir ventana actual en 2 partes, con línea vertical
C-x 1	sólo 1 ventana (la ventana actual, eliminando las otras)
C-x 0	elimina sólo la ventana actual
C-x o	cambia el cursor a la siguiente ventana

El cambio del cursor a una ventana cualquiera se puede hacer también rápidamente a través del *mouse*.

Comandos de movimiento

Algunos de estos comandos tienen dos teclas asociadas, como se indica a continuación.

C-b o ←	izquierda un carácter	**C-f** o →	derecha un carácter
C-p o ↑	arriba una línea	**C-n** o ↓	abajo una línea
C-a o `Home`	principio de la línea	**C-e** o `End`	fin de la línea
M-< o **C-**`Home`	principio del documento	**M-**> o **C-**`End`	fin del documento
M-f o **M-**→	avanza una palabra	**M-b** o **M-**←	retrocede una palabra
C-v o `Page Up`	avanza una página	**M-v** o `Page Down`	retrocede una página
M-g (número)	salta a la línea (número)	**C-l**	refresca la pantalla

Comandos de inserción y borrado

Al ser un editor en modo visual, las modificaciones se pueden hacer en el texto sin necesidad de entrar en ningún modo especial.

C-d o `Delete`	borra un carácter después del cursor
`Backspace`	borra un carácter antes del cursor
C-k	borra desde la posición del cursor hasta el fin de línea (no incluye el cambio de línea)
M-d	borra desde el cursor hacia adelante, hasta que termina una palabra
M-`Backspace`	borra desde el cursor hacia atrás, hasta que comienza una palabra
C-o	Inserta una línea en la posición del cursor

Mayúsculas y minúsculas

M-u	Cambia a mayúscula desde la posición del cursor hasta el fin de la palabra
M-l	Cambia a minúscula desde la posición del cursor hasta el fin de la palabra
M-c	Cambia a mayúscula el carácter en la posición del cursor y a minúscula hasta el fin de la palabra

Por ejemplo, veamos el efecto de cada uno de estos comandos sobre la palabra `EmAcS`, si el cursor está sobre la letra `E` (¡el efecto es distinto si está sobre cualquier otra letra!):

M-u : `EmAcS` ⟶ `EMACS`
M-l : `EmAcS` ⟶ `emacs`
M-c : `EmAcS` ⟶ `Emacs`

Transposición

Los siguientes comandos toman como referencia la posición actual del cursor. Por ejemplo, **C-t** intercambia el carácter justo antes del cursor con el carácter justo después.

C-t	Transpone dos caracteres
M-t	Transpone dos palabras
C-x C-t	Transpone dos líneas

Búsqueda y reemplazo

C-s	Búsqueda hacia el fin del texto
C-r	Búsqueda hacia el inicio del texto
M-%	Búsqueda y sustitución (pide confirmación cada vez)
M-&	Búsqueda y sustitución (sin confirmación)

Definición de regiones y reemplazo

Uno de los conceptos importantes en `emacs` es el de región. Para ello, necesitamos dos conceptos auxiliares: el *punto* y la *marca*. El punto es simplemente el cursor. Específicamente, es el punto donde *comienza* el cursor. Así, si el cursor se encuentra sobre la letra `c` en `emacs`, el punto está entre la `a` y la `c`. La marca, por su parte, es una señal que se coloca en algún punto del archivo con los comandos apropiados. La *región* es el espacio comprendido entre el punto y la marca.

Para colocar una marca basta ubicar el cursor en el lugar deseado, y teclear **C-**`Space` o **C-@**. Esto coloca la marca donde está el punto (en el ejemplo del párrafo anterior, quedaría entre las letras `a` y `c`. Una vez colocada la marca, podemos mover el cursor a cualquier otro lugar del archivo (hacia atrás o hacia adelante respecto a la marca). Esto define una cierta ubicación para el punto, y, por tanto, queda definida la región automáticamente.

La región es una porción del archivo que se puede manipular como un todo. Una región se puede borrar, copiar, pegar en otro punto del archivo o incluso en otro archivo; una región se puede imprimir, grabar como un archivo distinto; etc. Así, muchas operaciones importantes se pueden efectuar sobre un bloque del archivo.

Por ejemplo, si queremos duplicar una región, basta con definir la región deseada (poniendo la marca y el punto donde corresponda) y teclear **M-w**. Esto copia la región a un buffer temporal (llamado *kill buffer*). Luego movemos el cursor al lugar donde queremos insertar el texto duplicado, y hacemos **C-y**. Este comando toma el contenido del *kill buffer* y lo inserta en el archivo. El resultado final es que hemos duplicado una cierta porción del texto.

Si la intención era mover dicha porción, el procedimiento es el mismo, pero con el comando **C-w** en vez de **M-w**. **C-w** también copia la región a un *kill buffer*, pero borra el texto de la pantalla.

Resumiendo:

C-Space o **C-@**	Comienzo de región
M-w	Copia región
C-w	Corta región
C-y	Pega región

El concepto de *kill buffer* es mucho más poderoso que lo explicado recién. En realidad, muchos comandos, no sólo **M-w** y **C-w**, copian texto en un *kill buffer*. En general, cualquier comando que borre más de un carácter a la vez, lo hace. Por ejemplo, **C-k** borra una línea. Lo que hace no es sólo borrarla, sino además copiarla en un *kill buffer*. Lo mismo ocurre con los comandos que borran palabras completas (**M-d**, **M-Backspace**), y muchos otros. Lo interesante es que **C-y** funciona también en todos esos casos: **C-y** lo único que hace es tomar el último texto colocado en un *kill buffer* (resultado de la última operación que borró más de un carácter a la vez), y lo coloca en el archivo. Por lo tanto, no sólo podemos copiar o mover "regiones", sino también palabras o líneas. Más aún, el *kill buffer* no es borrado con el **C-y**, así que ese mismo texto puede ser duplicado muchas veces. Continuará disponible con **C-y** mientras no se ponga un nuevo texto en el *kill buffer*.

Además, `emacs` dispone no de uno sino de muchos *kill buffers*. Esto permite recuperar texto borrado hace mucho rato. En efecto, cada vez que se borra más de un carácter de una vez, se una un nuevo *kill buffer*. Por ejemplo, consideremos el texto:

```
La primera linea del texto,
la segunda linea,
y finalmente la tercera.
```

Si en este párrafo borramos la primera línea (con **C-k**), después borramos la primera palabra de la segunda (con **M-d**, por ejemplo), y luego la segunda palabra de la última, entonces habrá tres *kill buffers* ocupados:

buffer 1 : `La primera linea del texto,`
buffer 2 : `la`
buffer 3 : `finalmente`

Al colocar el cursor después del punto final, **C-y** toma el contenido del último *kill buffer* y lo coloca en el texto:

```
 segunda linea,
y  la tercera. finalmente
```

Si se teclea ahora **M-y**, el último texto recuperado, `finalmente`, es reemplazado por el penúltimo texto borrado, y que está en el *kill buffer* anterior:

```
 segunda linea,
y  la tercera. la
```

Además, la posición de los *kill buffers* se rota:

```
buffer 1  : finalmente
buffer 2  : La primera linea del texto,
buffer 3  : la
```

Sucesivas aplicaciones de **M-y** después de un **C-y** rotan sobre todos los *kill buffers* (que pueden ser muchos). El editor, así, conserva un conjunto de las últimas zonas borradas durante la edición, pudiendo recuperarse una antigua a pesar de haber seleccionado una nueva zona, o borrado una nueva palabra o línea. Toda la información en los *kill buffers* se pierde al salir de `emacs` (**C-c**).

Resumimos entonces los comandos para manejo de los *kill buffers*:

C-y	Copia el contenido del último *kill buffer* ocupado
M-y	Rota los *kill buffers* ocupados

Definición de macros

La clave de la configurabilidad de `emacs` está en la posibilidad de definir nuevos comandos que modifiquen su comportamiento o agreguen nuevas funciones de acuerdo a nuestras necesidades. Un modo de hacerlo es a través del archivo de configuración `$HOME/.emacs`, para lo cual se sugiere leer la documentación disponible en la distribución instalada. Sin embargo, si sólo necesitamos un nuevo comando en la sesión de trabajo actual, un modo más simple es definir una *macro*, un conjunto de órdenes que son ejecutados como un solo comando. Los comandos relevantes son:

C-x (	Comienza la definición de una macro
C-x)	Termina la definición de una macro
C-x e	Ejecuta una macro definida

Todas las sucesiones de teclas y comandos dados entre **C-x (** y **C-x)** son recordados por `emacs`, y después pueden ser ejecutados de una vez con **C-x e**.

Como ejemplo, consideremos el siguiente texto, con los cinco primeros lugares del ránking ATP (sistema de entrada) al 26 de marzo de 2002:

```
1 hewitt, lleyton (Aus)
2 kuerten, gustavo (Bra)
3 ferrero, juan (Esp)
4 kafelnikov, yevgeny (Rus)
5 haas, tommy (Ger)
```

Supongamos que queremos: (a) poner los nombres y apellidos con mayúscula (como debería ser); (b) poner las siglas de países sólo en mayúsculas.

Para definir una macro, colocamos el cursor al comienzo de la primera línea, en el `1`, y damos **C-x (**. Ahora realizamos todos los comandos necesarios para hacer las tres tareas solicitadas para el primer jugador solamente: **M-f** (avanza una palabra, hasta el espacio antes de `hewitt`; **M-c M-c** (cambia a `Hewitt, Lleyton`); **M-u** (cambia a `AUS`); `Home` (vuelve el cursor al comienzo de la línea); ↓ (coloca el cursor al comienzo de la línea siguiente, en el `2`). Los dos últimos pasos son importantes, porque dejan el cursor en la posición correcta para ejecutar el comando nuevamente. Ahora terminamos la definición con **C-x)**. Listo. Si ahora ejecutamos la macro, con **C-x e**, veremos que la segunda línea queda modificada igual que la primera, y así podemos continuar hasta el final:

```
1 Hewitt, Lleyton (AUS)
2 Kuerten, Gustavo (BRA)
3 Ferrero, Juan (ESP)
4 Kafelnikov, Yevgeny (RUS)
5 Haas, Tommy (GER)
```

Comandos por nombre

Aparte de los ya comentados existen muchas otras órdenes que no tienen necesariamente una tecla asociada (*bindkey*) asociada. Para su ejecución debe de teclearse previamente:

`M-x`

y a continuación en la zona inferior de la pantalla se introduce el comando deseado. Empleando el `TAB` se puede completar dicho comando (igual que en `bash`).

De hecho, esto sirve para cualquier comando, incluso si tiene tecla asociada. Por ejemplo, ya sabemos **M-g** `n` va a la línea n del documento. Pero esto no es sino el comando `goto-line`, y se puede también ejecutar tecleando: `M-x goto-line n`.

Repetición

Todos los comandos de `emacs`, tanto los que tienen una tecla asociada como los que se ejecutan con nombre, se pueden ejecutar más de una vez, anteponiéndoles un argumento numérico con

```
M-(number)
```

Por ejemplo, si deseamos escribir 20 letras `e`, basta teclear `M-20 e`. Esto es particularmente útil con las macros definidos por el usuario. En el ejemplo anterior, con el ránking ATP, después de definir la macro quedamos en la línea 2, y en vez de ejecutar **C-x e** 4 veces, podemos teclear `M-4 C-x e`, con el mismo resultado, pero en mucho menos tiempo.

Para terminar la discusión de este editor, diremos que es conveniente conocer las secuencias de control básico de `emacs`:
`C-a, C-e, C-k, C-y, C-w, C-t, C-d`, etc.,
porque funcionan para editar la línea de comandos en el *shell*, como también en muchos programas de texto y en ventanas de diálogo de las aplicaciones *X Windows*. A su vez, los editores `jed`, `xjed`, `jove` también usan por defecto estas combinaciones.

Apéndice D

Una breve introducción a Octave/Matlab

D.1. Introducción

Octave es un poderoso software para análisis numérico y visualización. Muchos de sus comandos son compatibles con Matlab. En estos apuntes revisaremos algunas características de estos programas. En realidad, el autor de este capítulo ha sido usuario durante algunos años de Matlab, de modo que estos apuntes se han basado en ese conocimiento, considerando los comandos que le son más familiares de Matlab. En la mayoría de las ocasiones he verificado que los comandos descritos son también compatibles con Octave, pero ocasionalmente se puede haber omitido algo....

Matlab es una abreviación de *Matrix Laboratory*. Los elementos básicos con los que se trabaja con matrices. Todos los otros tipos de variables (vectores, texto, polinomios, etc.), son tratados como matrices. Esto permite escribir rutinas optimizadas para el trabajo con matrices, y extender su uso a todos los otros tipos de variables fácilmente.

D.2. Interfase con el programa

Con Octave/Matlab se puede interactuar de dos modos: un modo interactivo, o a través de *scripts*. Al llamar a Octave/Matlab (escribiendo `octave` en el prompt, por ejemplo), se nos presenta un prompt. Si escribimos `a=1`, el programa responderá `a=1`. Alternativamente, podemos escribir `a=3;` (con punto y coma al final), y el programa no responderá (elimina el eco), pero almacena el nuevo valor de `a`. Si a continuación escribimos `a`, el programa responderá `a=3`. Hasta este punto, hemos usado el modo interactivo.

Alternativamente, podemos introducir las instrucciones anteriores en un archivo, llamado, por ejemplo, `prueba.m`. En el prompt, al escribir `prueba`, y si nuestro archivo está en el path de búsqueda del programa, las líneas de `prueba.m` serán ejecutadas una a una. Por ejemplo, si el archivo consta de las siguientes cuatro líneas:

```
a=3;
```

```
a
a=5
a
```

el programa responderá con

```
a=3
a=5
a=5
```

`prueba.m` corresponde a un *script*. Todas las instrucciones de Octave/Matlab pueden ejecutarse tanto en modo interactivo como desde un *script*. En Linux se puede ejecutar un archivo de comandos Octave de modo *stand-alone* incluyendo en la primera línea:
`#!/usr/bin/octave -q`.

D.3. Tipos de variables

D.3.1. Escalares

A pesar de que éstos son sólo un tipo especial de matrices (ver subsección siguiente), conviene mencionar algunas características específicas.

- Un número sin punto decimal es tratado como un entero exacto. Un número con punto decimal es tratado como un número en doble precisión. Esto puede no ser evidente en el output. Por *default*, `8.4` es escrito en pantalla como `8.4000`. Tras la instrucción `format long`, sin embargo, es escrito como `8.40000000000000`. Para volver al formato original, basta la instrucción `format`.

- Octave/Matlab acepta números reales y complejos. La unidad imaginaria es `i`: `8i` y `8*i` definen el mismo número complejo. Como `i` es una varible habitualmente usada en iteraciones, también está disponible `j` como un sinónimo. Octave/Matlab distinguen entre mayúsculas y minúsculas.

- Octave/Matlab representa de manera especial los infinitos y cantidades que no son números. `inf` es infinito, y `NaN` es un no-número (Not-a-Number). Por ejemplo, escribir `a=1/0` no arroja un error, sino un mensaje de advertencia, y asigna a `a` el valor `inf`. Análogamente, `a=0/0` asigna a `a` el valor `NaN`.

D.3.2. Matrices

Este tipo de variable corresponde a escalares, vectores fila o columna, y matrices convencionales.

Construcción

Las instrucciones:

```
a = [1 2 ; 3 4]
```

ó

```
a = [1, 2; 3, 4]
```

definen la matriz $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. Las comas (opcionales) separan elementos de columnas distintas, y los punto y coma separan elementos de filas distintas. El vector fila $(1 \quad 2)$ es

```
b = [1 2]
```

y el vector columna $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$ es

```
c = [1;2]
```

Un número se define simplemente como `d = [3]` ó `d = 3`.

Nota importante: Muchas funciones de Octave/Matlab en las páginas siguientes aceptan indistintamente escalares, vectores filas, vectores columnas, o matrices, y su output es un escalar, vector o matriz, respectivamente. Por ejemplo, `log(a)` es un vector fila si `a` es un vector fila (donde cada elemento es el logaritmo natural del elemento correspondiente en `a`), y un vector columna si `a` es un vector columna. En el resto de este manual *no se advertira* este hecho, y se pondrán ejemplos con un solo tipo de variable, en el entendido que el lector está conciente de esta nota.

Acceso y modificación de elementos individuales

Accesamos los elementos de cada matriz usando los índices de filas y columnas, que parten de uno. Usando la matriz `a` antes definida, `a(1,2)` es 2. Para modificar un elemento, basta escribir, por ejemplo, `a(2,2) = 5`. Esto convierte a la matriz en $\begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}$. En el caso especial de vectores filas o columnas, basta un índice. (En los ejemplos anteriores, `b(2)` = `c(2)` = 2.)

Una característica muy importante del programa es que toda matriz es redimensionada automáticamente cuando se intenta modificar un elemento que sobrepasa las dimensiones actuales de la matriz, llenando con ceros los lugares necesarios. Por ejemplo, si `b = [1 2]`, y en seguida intentamos la asignación `b(5) = 8`, `b` es automáticamente convertido al vector fila de 5 elementos `[1 2 0 0 8]`.

Concatenación de matrices

Si $a = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $b = \begin{pmatrix} 5 & 6 \end{pmatrix}$, $c = \begin{pmatrix} 7 \\ 8 \end{pmatrix}$, entonces

```
d = [a c]
```

$d = \begin{pmatrix} 1 & 2 & 7 \\ 3 & 4 & 8 \end{pmatrix}$

```
d = [a; b]
```

$d = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$

```
d = [a [0; 0] c]
```

$d = \begin{pmatrix} 1 & 2 & 0 & 7 \\ 3 & 4 & 0 & 8 \end{pmatrix}$

D.3.3. Strings

Las cadenas de texto son casos particulares de vectores fila, y se construyen y modifican de modo idéntico.

Construcción

Las instrucciones

```
t = ['un buen texto']
t = ["un buen texto"]
t = 'un buen texto'
t = "un buen texto"
```

definen el mismo string `t`.

Acceso y modificación de elementos individuales

```
r = t(4)
```

```
r = 'b'
t(9) = 's'

texto = 'un buen sexto'
```

Concatenación

```
t = 'un buen texto';
t1 = [t ' es necesario']

t1 = 'un buen texto es necesario'
```

D.3.4. Estructuras

Las estructuras son extensiones de los tipos de variables anteriores. Una estructura consta de distintos campos, y cada campo puede ser una matriz (es decir, un escalar, un vector o una matriz), o una string.

Construcción

Las líneas

```
persona.nombre = 'Eduardo'
persona.edad = 30
persona.matriz_favorita = [2 8;10 15];
```

definen una estructura con tres campos, uno de los cuales es un string, otro un escalar, y otro una matriz:

```
persona =
{
nombre = 'Eduardo';
edad = 30;
matriz_favorita = [2 8; 10 15];
}
```

Acceso y modificación de elementos individuales

```
s = persona.nombre

s = 'Eduardo'
persona.nombre = 'Claudio'
persona.matriz_favorita(2,1) = 8

persona =
{
nombre = 'Claudio';
edad = 30;
matriz_favorita = [2 8; 8 15];
}
```

D.4. Operadores básicos

D.4.1. Operadores aritméticos

Los operadores `+`, `-`, `*` corresponden a la suma, resta y multiplicación convencional de matrices. Ambas matrices deben tener la misma dimensión, a menos que una sea un escalar. Un escalar puede ser sumado, restado o multiplicado de una matriz de cualquier dimensión.

`.*` y `./` permiten multiplicar y dividir elemento por elemento. Por ejemplo, si

$$a = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad b = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

entonces

```
c = a.*b
```

$$c = \begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix}$$

```
c = a./b
```

$$c = \begin{pmatrix} 0.2 & 0.3333 \\ 0.42857 & 0.5 \end{pmatrix}$$

Si `b` es un escalar, `a.*b` y `a./b` equivalen a `a*b` y `a/b`.

`a^b` es `a` elevado a `b`, si `b` es un escalar. `a.^b` eleva cada elemento de `a` a `b`.

`a'` es la matriz $a^\dagger$ (traspuesta y conjugada)

`a.'` es la matriz traspuesta de `a`.

D.4.2. Operadores relacionales

Los siguientes operadores están disponibles:

`<  <=  >  >=  ==  ~=`

El resultado de estas operaciones es 1 (verdadero) ó 0 (falso). Si uno de los operandos es una matriz y el otro un escalar, se compara el escalar con cada elemento de la matriz. Si ambos operandos son matrices, el test se realiza elemento por elemento; en este caso, las matrices deben ser de igual dimensión. Por ejemplo,

```
a = [1 2 3];
b = [4 2 1];
c = (a<3);
d = (a>=b);
```

$c = (1, 1, 0)$
$d = (0, 1, 1)$

D.4.3. Operadores lógicos

Los siguientes símbolos corresponden a los operadores `AND`, `OR` y `NOT`:

```
&  |  ~
```

El resultado de estas operaciones es 1 (verdadero) ó 0 (falso).

D.4.4. El operador :

Es uno de los operadores fundamentales. Permite crear vectores y extraer submatrices.

: crea vectores de acuerdo a las siguientes reglas:

`j:k`	es lo mismo que `[j,j+1,...,k]`, si `j<=k`.
`j:i:k`	es lo mismo que `[j,j+i,j+2*i,...,k]`, si `i>0` y `j<k`, o si `i<0` y `j>k`.

: extrae submatrices de acuerdo a las siguientes reglas:

`A(:,j)`	es la `j`-ésima columna de `A`.
`A(i,:)`	es la `i`-ésima fila de `A`.
`A(:,:)`	es `A`.
`A(:,j:k)`	es `A(:,j)`, `A(:,j+1)`, ..., `A(:,k)`.
`A(:)`	son todos los elementos de `A`, agrupados en una única columna.

D.4.5. Operadores de aparición preferente en scripts

Los siguientes operadores es más probable que aparezcan durante la escritura de un *script* que en modo interactivo.

`%` : Comentario. El resto de la línea es ignorado.

`...` : Continuación de línea. Si una línea es muy larga y no cabe en la pantalla, o por alguna otra razón se desea dividir una línea, se puede usar el operador `...` . Por ejemplo,

```
m = [1 2 3 ...
4 5 6];
```

es equivalente a

```
m = [1 2 3 4 5 6];
```

D.5. Comandos matriciales básicos

Antes de revisar una a una diversas familias de comandos disponibles, y puesto que las matrices son el elemento fundamental en Octave/Matlab, en esta sección reuniremos algunas de las funciones más frecuentes sobre matrices, y cómo se realizan en Octave/Matlab.

Op. aritmética	`+, -, *, .*, ./`	(ver subsección D.4.1)
Conjugar	`conj(a)`	
Trasponer	`a.'`	
Trasponer y conjugar	`a'`	
Invertir	`inv(a)`	
Autovalores, autovectores	`[v,d]=eig(a)`	(ver subsección D.6.5)
Determinante	`det(a)`	
Extraer elementos	`:`	(ver subsección D.4.4)
Traza	`trace(a)`	
Dimensiones	`size(a)`	
Exponencial	`exp(a)`	(elemento por elemento)
	`expm(a)`	(exponencial matricial)

D.6. Comandos

En esta sección revisaremos diversos comandos de uso frecuente en Octave/Matlab. Esta lista no pretende ser exhaustiva (se puede consultar la documentación para mayores detalles), y está determinada por mi propio uso del programa y lo que yo considero más frecuente debido a esa experiencia. Insistimos en que ni la lista de comandos es exhaustiva, ni la lista de ejemplos o usos de cada comando lo es. Esto pretende ser sólo una descripción de los aspectos que me parecen más importantes o de uso más recurrente.

D.6.1. Comandos generales

`clear` Borra variables y funciones de la memoria

`clear`	Borra todas las variables en memoria
`clear a`	Borra la variable `a`

`disp` Presenta matrices o texto

`disp(a)` presenta en pantalla los contenidos de una matriz, sin imprimir el nombre de la matriz. `a` puede ser una string.

```
disp('     c1       c2');
disp([.3 .4]);
```

$c1$	$c2$
0.30000	0.40000

`load, save` Carga/Guarda variables desde el disco

`save fname a b`	Guarda las variables `a` y `b` en el archivo `fname`
`load fname`	Lee el archivo `fname`, cargando las definiciones de variables en él definidas.

`size,length` Dimensiones de una matriz/largo de un vector

Si a es una matrix de $n \times m$:

`d = size(a)`	`d` = `[m,n]`
`[m,n] = size(a)`	Aloja en `m` el número de filas, y en `n` el de columnas

Si b es un vector de n elementos, `length(b)` es n.

`who` Lista de variables en memoria

`quit` Termina Octave/Matlab

D.6.2. Como lenguaje de programación

Control de flujo

`for`

```
n=3;                                    a=[1 4 9]
for i=1:n
    a(i)=i^2;
end
```

Para Octave el vector resultante es columna en vez de fila.

Observar el uso del operador : para generar el vector `[1 2 3]`. Cualquier vector se puede utilizar en su lugar: `for i=[2 8 9 -3]`, `for i=10:-2:1` (equivalente a `[10 8 6 4 2]`), etc. son válidas. El ciclo `for` anterior se podría haber escrito en una sola línea así:

```
for i=1:n, a(i)=i^2; end
```

`if, elseif, else`

Ejemplos:

a)
```
if a~=b, disp(a); end
```

b)
```
if a==[3 8 9 10]
   b = a(1:3);
end
```

c)
```
if a>3
   clear a;
elseif a<0
   save a;
else
   disp('Valor de a no considerado');
end
```

Naturalmente, `elseif` y `else` son opcionales. En vez de las expresiones condicionales indicadas en el ejemplo pueden aparecer cualquier función que dé valores `1` (verdadero) ó `0` (falso).

`while`

```
while s
    comandos
end
```

Mientras `s` es `1`, se ejecutan los `comandos` entre `while` y `end`. `s` puede ser cualquier expresión que dé por resultado `1` (verdadero) ó `0` (falso).

`break`

Interrumpe ejecución de ciclos `for` o `while`. En *loops* anidados, `break` sale del más interno solamente.

Funciones lógicas

Además de expresiones construidas con los operadores relacionales `==`, `<=`, etc., y los operadores lógicos `&`, `|` y `~`, los comandos de control de flujo anteriores admiten cualquier función cuyo resultado sea `1` (verdadero) ó `0` (falso). Particularmente útiles son funciones como las siguientes:

`all(a)`	`1` si todos los elementos de `a` son no nulos, y `0` si alguno es cero
`any(a)`	`1` si alguno de los elementos de `a` es no nulo
`isempty(a)`	`1` si `a` es matriz vacía (`a=[]`)

Otras funciones entregan matrices de la misma dimensión que el argumento, con unos o ceros en los lugares en que la condición es verdadera o falsa, respectivamente:

`finite(a)`	`1` donde `a` es finito (no `inf` ni `NaN`)
`isinf(a)`	`1` donde `a` es infinito
`isnan(a)`	`1` donde `a` es un `NaN`

Por ejemplo, luego de ejecutar las líneas

```
x = [-2 -1 0 1 2];
y = 1./x;
a = finite(y);
b = isinf(y);
c = isnan(y);
```

se tiene

```
a = [1 1 0 1 1]
b = [0 0 1 0 0]
c = [0 0 0 0 0]
```

Otra función lógica muy importante es `find`:

`find(a)` Encuentra los índices de los elementos no nulos de `a`.

Por ejemplo, si ejecutamos las líneas

```
x=[11 0 33 0 55];
z1=find(x);
z2=find(x>0 & x<40);
```

obtendremos

```
z1 = [1 3 5]
z2 = [1 3]
```

`find` también puede dar dos resultados de salida simultáneamente (más sobre esta posibilidad en la sección D.6.2), en cuyo caso el resultado son los pares de índices (índices de fila y columna) para cada elemento no nulo de una matriz

```
y=[1 2 3 4 5;6 7 8 9 10];
[z3,z4]=find(y>8);
```

da como resultado

```
z3 = [2;2];
z4 = [4;5];
```

`z3` contiene los índice de fila y `z4` los de columna para los elementos no nulos de la matriz `y>8`. Esto permite construir, por ejemplo, la matriz `z5=[z3 z4]` $= \begin{pmatrix} 2 & 4 \\ 2 & 5 \end{pmatrix}$, en la cual cada fila es la posición de `y` tal que la condición `y>8` es verdadera (en este caso, es verdadera para los elementos `y(2,4)` e `y(2,5)`).

Funciones definidas por el usuario

Octave/Matlab puede ser fácilmente extendido por el usuario definiendo nuevas funciones que le acomoden a sus propósitos. Esto se hace a través del comando `function`.

Podemos definir (en modo interactivo o dentro de un *script*), una función en la forma

```
function nombre (argumentos)
   comandos
endfunction
```

`argumentos` es una lista de argumentos separados por comas, y `comandos` es la sucesión de comandos que serán ejecutados al llamar a `nombre`. La lista de argumentos es opcional, en cuyo caso los paréntesis redondos se pueden omitir.

A mediano y largo plazo, puede ser mucho más conveniente definir las funciones en archivos especiales, listos para ser llamados en el futuro desde modo interactivo o desde cualquier *script*. Esto se hace escribiendo la definición de una función en un *script* con extensión `.m`. Cuando Octave/Matlab debe ejecutar un comando o función que no conoce, por ejemplo, `suma(x,y)`,busca en los archivos accesibles en su path de búsqueda un archivo llamado `suma.m`, lo carga y ejecuta la definición contenida en ese archivo.

Por ejemplo, si escribimos en el *script* `suma.m` las líneas

```
function s=suma(x,y)
s = x+y;
```

el resultado de `suma(2,3)` será `5`.

Las funciones así definidas pueden entregar más de un argumento si es necesario (ya hemos visto algunos ejemplos con `find` y `size`). Por ejemplo, definimos una función que efectúe un análisis estadístico básico en `stat.m`:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/n));
```

Al llamarla en la forma `[m,s] = stat(x)`, si `x` es un vector fila o columna, en `m` quedará el promedio de los elementos de `x`, y en `s` la desviación estándard.

Todas las variables dentro de un *script* que define una función son locales, a menos que se indique lo contrario con `global`. Por ejemplo, si un *script* `x.m` llama a una función `f`, y dentro de `f.m` se usa una variable `a` que queremos sea global, ella se debe declarar en la forma `global a` tanto en `f.m` como en el *script* que la llamó, `x.m`, y en todo otro *script* que pretenda usar esa variable global.

D.6.3. Matrices y variables elementales

Matrices constantes importantes

Las siguientes son matrices que se emplean habitualmente en distintos contextos, y que es útil tener muy presente:

`eye(n)`	Matriz identidad de $n \times n$
`ones(m,n)`	Matriz de $m \times n$, con todos los elementos igual a 1.
`rand(m,n)`	Matriz de $m \times n$ de números al azar, distribuidos uniformemente.
`randn(m,n)`	Igual que `rand`, pero con distribución normal (Gaussiana).
`zeros(m,n)`	Igual que `ones`, pero con todos los elementos 0.

Matrices útiles para construir ejes o mallas para graficar

Las siguientes son matrices se emplean habitualmente en la construcci'on de gráficos:

`v = linspace(min,max,n)`	Vector cuyo primer elemento es `min`, su último elemento es `max`, y tiene `n` elementos equiespaciados.
`v = logspace(min,max,n)`	Análogo a `linspace`, pero los `n` elementos están espaciados logarítmicamente.
`[X,Y] = meshgrid(x,y)`	Construye una malla del plano x-y. Las filas de `X` son copias del vector `x`, y las columnas de `Y` son copias del vector `y`.

Por ejemplo:

```
x = [1 2 3];
y = [4 5];
[X,Y] = meshgrid(x,y);
```

da

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \qquad Y = \begin{pmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \end{pmatrix}.$$

Notemos que al tomar sucesivamente los distintos pares ordenados (`X(1,1),Y(1,1)`), (`X(1,2),Y(1,2)`), (`X(1,3),Y(1,3)`), etc., se obtienen todos los pares ordenados posibles tales que el primer elemento está en `x` y el segundo está en `y`. Esta característica hace particularmente útil el comando `meshgrid` en el contexto de gráficos de funciones de dos variables (ver secciones D.6.7, D.6.7).

Constantes especiales

Octave/Matlab proporciona algunos números especiales, algunos de los cuales ya mencionamos en la sección D.3.1.

`i`, `j`	Unidad imaginaria ($\sqrt{-1}$)
`inf`	Infinito
`NaN`	Not-A-Number
`pi`	El número π ($= 3.1415926535897\ldots$)

Funciones elementales

Desde luego, Octave/Matlab proporciona todas las funciones matemáticas básicas. Por ejemplo:

a) Funciones sobre números reales/complejos

`abs`	Valor absoluto de números reales, o módulo de números imaginarios
`angle`	Ángulo de fase de un número imaginario
`conj`	Complejo conjugado
`real`	Parte real
`imag`	Parte imaginaria
`sign`	Signo
`sqrt`	Raíz cuadrada

b) Exponencial y funciones asociadas

`cos`, `sin`, etc.	Funciones trigonométricas
`cosh`, `sinh`, etc.	Funciones hiperbólicas
`exp`	Exponencial
`log`	Logaritmo

c) Redondeo

`ceil`	Redondear hacia $+\infty$
`fix`	Redondear hacia cero
`floor`	Redondear hacia $-\infty$
`round`	Redondear hacia el entero más cercano

Funciones especiales

Además, Octave/Matlab proporciona diversas funciones matemáticas especiales. Algunos ejemplos:

`bessel`	Función de Bessel
`besselh`	Función de Hankel
`beta`	Función beta
`ellipke`	Función elíptica
`erf`	Función error
`gamma`	Función gamma

Así, por ejemplo, `bessel(alpha,X)` evalúa la función de Bessel de orden `alpha`, $J_\alpha(x)$, para cada elemento de la matriz X.

D.6.4. Polinomios

Octave/Matlab representa los polinomios como vectores fila. El polinomio

$$p = c_n x^n + \cdots + c_1 x + c_0$$

es representado en Octave/Matlab en la forma

```
p = [c_n, ..., c1, c0]
```

Podemos efectuar una serie de operaciones con los polinomios así representados.

`poly(x)`	Polinomio cuyas raíces son los elementos de `x`.
`polyval(p,x)`	Evalúa el polinomio `p` en `x` (en los elementos de `x` si éste es un vector)
`roots(p)`	Raíces del polinomio `p`

D.6.5. Álgebra lineal (matrices cuadradas)

Unos pocos ejemplos, entre los comandos de uso más habitual:

`det`	Determinante
`rank`	Número de filas o columnas linealmente independientes
`trace`	Traza
`inv`	Matriz inversa
`eig`	Autovalores y autovectores
`poly`	Polinomio característico

Notar que `poly` es la misma función de la sección D.6.4 que construye un polinomio de raíces dadas. En el fondo, construir el polinomio característico de una matriz es lo mismo, y por tanto tiene sentido asignarles la misma función. Y no hay confusión, pues una opera sobre vectores y la otra sobre matrices cuadradas.

El uso de todos estos comandos son autoexplicativos, salvo `eig`, que se puede emplear de dos modos:

```
d = eig(a)
[V,D] = eig(a)
```

La primera forma deja en `d` un vector con los autovalores de `a`. La segunda, deja en `D` una matriz diagonal con los autovalores, y en `V` una matiz cuyas columnas son los autovalores, de modo que `A*V = V*D`. Por ejemplo, si `a =[1 2; 3 4]`, entonces

$$d = \begin{pmatrix} 5.37228 \\ -0.37228 \end{pmatrix}$$

y

$$D = \begin{pmatrix} 5.37228\ldots & 0 \\ 0 & -0.37228\ldots \end{pmatrix}, \qquad V = \begin{pmatrix} 0.41597\ldots & -0.82456\ldots \\ 0.90938\ldots & 0.56577\ldots \end{pmatrix}.$$

La primera columna de `V` es el autovector de `a` asociado al primer autovalor, $5.37228\ldots$.

D.6.6. Análisis de datos y transformada de Fourier

En Octave/Matlab están disponibles diversas herramientas para el análisis de series de datos (estadística, correlaciones, convolución, etc.). Algunas de las operaciones básicas son:

a) Máximos y mínimos
Si `a` es un vector, `max(a)` es el mayor elemento de `a`. Si es una matriz, `max(a)` es un vector fila, que contiene el máximo elemento para cada columna.

```
a = [1 6 7; 2 8 3; 0 4 1]
b = max(a)
```

$b = (2 \quad 8 \quad 7)$

Se sigue que el mayor elemento de la matriz se obtiene con `max(max(a))`.

`min` opera de modo análogo, entregando los mínimos.

b) Estadística básica
Las siguientes funciones, como `min` y `max`, operan sobre vectores del modo usual, y sobre matrices entregando vectores fila, con cada elemento representando a cada columna de la matriz.

`mean`	Valor promedio
`median`	Mediana
`std`	Desviación standard
`prod`	Producto de los elementos
`sum`	Suma de los elementos

c) Orden
`sort(a)` ordena los elementos de `a` en orden ascendente si `a` es un vector. Si es una matriz, ordena cada columna.

```
b = sort([1 3 9; 8 2 1; 4 -3 0]);
```

$$b = \begin{pmatrix} 1 & -3 & 0 \\ 4 & 2 & 1 \\ 8 & 3 & 9 \end{pmatrix}$$

d) Transformada de Fourier

Por último, es posible efectuar transformadas de Fourier directas e inversas, en una o dos dimensiones. Por ejemplo, `fft` y `ifft` dan la transformada de Fourier y la transformada

inversa de `x`, usando un algoritmo de *fast Fourier transform* (FFT). Específicamente, si `X=fft(x)` y `x=ifft(X)`, y los vectores son de largo `N`:

$$X(k) = \sum_{j=1}^{N} x(j)\omega_N^{(j-1)(k-1)} \ ,$$

$$x(j) = \frac{1}{N}\sum_{k=1}^{N} X(k)\omega_N^{-(j-1)(k-1)} \ ,$$

donde $\omega_N = e^{-2\pi i/N}$.

D.6.7. Gráficos

Una de las características más importantes de Matlab son sus amplias posibilidades gráficas. Algunas de esas características se encuentran también en Octave. En esta sección revisaremos el caso de gráficos en dos dimensiones, en la siguiente el caso de tres dimensiones, y luego examinaremos algunas posibilidades de manipulación de gráficos.

Gráficos bidimensionales

Para graficar en dos dimensiones se usa el comando `plot`. `plot(x,y)` grafica la ordenada `y` versus la abscisa `x`. `plot(y)` asume abscisa `[1,2,...n]`, donde `n` es la longitud de `y`.

Ejemplo: Si `x=[2 8 9]`, `y=[6 3 2]`, entonces
`plot(x,y)`

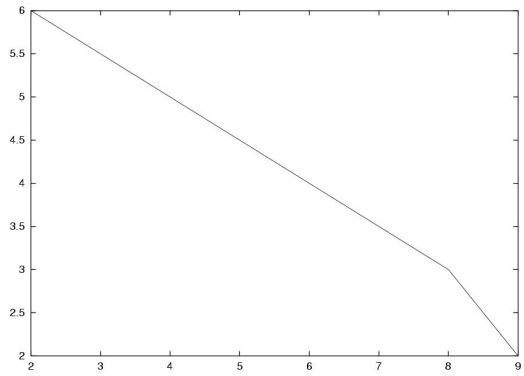


Figura D.1: Gráfico simple.

Por *default*, Octave utiliza `gnuplot` para los gráficos. Por *default*, los puntos se conectan con una línea roja en este caso. El aspecto de la línea o de los puntos puede ser modificado.

Por ejemplo, `plot(x,y,'ob')` hace que los puntos sean indicados con círculos (`'o'`) azules (`'b'`, *blue*). Otros modificadores posibles son:

`-`	línea (*default*)	`r`	red
`.`	puntos	`g`	green
`@`	otro estilo de puntos	`b`	blue
`+`	signo más	`m`	magenta
`*`	asteriscos	`c`	cyan
`o`	círculos	`w`	white
`x`	cruces		

Dos o más gráficos se pueden incluir en el mismo output agregando más argumentos a `plot`. Por ejemplo: `plot(x1,y1,'x',x2,y2,'og',x3,y3,'.c')`.

Los mapas de contorno son un tipo especial de gráfico. Dada una función $z = f(x, y)$, nos interesa graficar los puntos (x, y) tales que $f = c$, con c alguna constante. Por ejemplo, consideremos

$$z = xe^{-x^2-y^2} \ , \quad x \in [-2, 2], \ y \in [-2, 3] \ .$$

Para obtener el gráfico de contorno de z, mostrando los niveles $z = -.3$, $z = -.1$, $z = 0$, $z = .1$ y $z = .3$, podemos usar las instrucciones:

```
x = -2:.2:2;
y = -2:.2:3;
[X,Y] = meshgrid(x,y);
Z = X.*exp(-X.^2-Y.^2);
contour(Z.',[-.3 -.1 0 .1 .3],x,y);   # Octave por default (gnuplot)
contour(x, y, Z.',[-.3 -.1 0 .1 .3]); # Octave con plplot y Matlab
```

Las dos primeras líneas definen los puntos sobre los ejes x e y en los cuales la función será evaluada. En este caso, escojimos una grilla en que puntos contiguos están separados por `.2`. Para un mapa de contorno, necesitamos evaluar la función en todos los pares ordenados (x, y) posibles al escoger x en `x` e y en `y`. Para eso usamos `meshgrid` (introducida sin mayores explicaciones en la sección D.6.3). Luego evaluamos la función [`Z` es una matriz, donde cada elemento es el valor de la función en un par ordenado (x, y)], y finalmente construimos el mapa de contorno para los niveles deseados.

Gráficos tridimensionales

También es posible realizar gráficos tridimensionales. Por ejemplo, la misma doble gaussiana de la sección anterior se puede graficar en tres dimensiones, para mostrarla como una superficie $z(x, y)$. Basta reemplazar la última instrucción, que llama a `contour`, por la siguiente:

```
mesh(X,Y,Z)
```

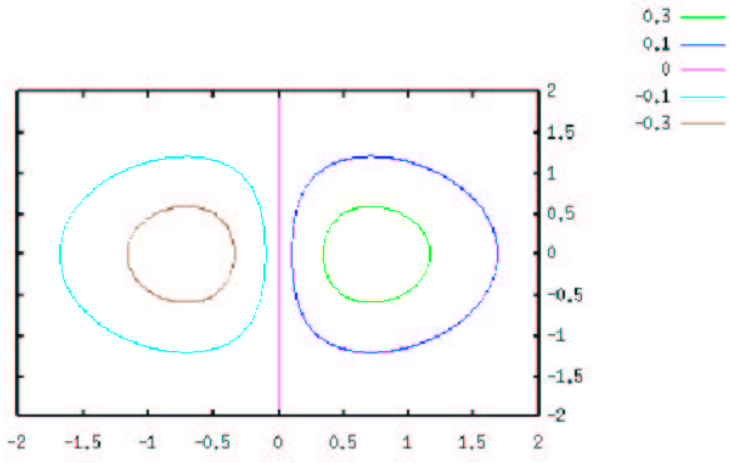


Figura D.2: Curvas de contorno.

Observar que, mientras `contour` acepta argumentos dos de los cuales son vectores, y el tercero una matriz, en `mesh` los tres argumentos son matrices de la misma dimensión (usamos `X`, `Y`, en vez de `x`, `y`).

Nota importante: Otro modo de hacer gráficos bi y tridimensionales es con `gplot` y `gsplot` (instrucciones asociadas realmente no a Octave sino a `gnuplot`, y por tanto no equivalentes a instrucciones en Matlab). Se recomienda consultar la documentación de Octave para los detalles.

Manipulación de gráficos

Los siguientes comandos están disponibles para modificar gráficos construidos con Octave/Matlab:

a) Ejes

`axis([x1 y1 x2 y2])` Cambia el eje x al rango $(x1, x2)$, y el eje y al rango $(y1, y2)$.

b) Títulos

`title(s)` Título (`s` es un string)
`xlabel(s)` Título del eje x, y, z.
`ylabel(s)`
`zlabel(s)`

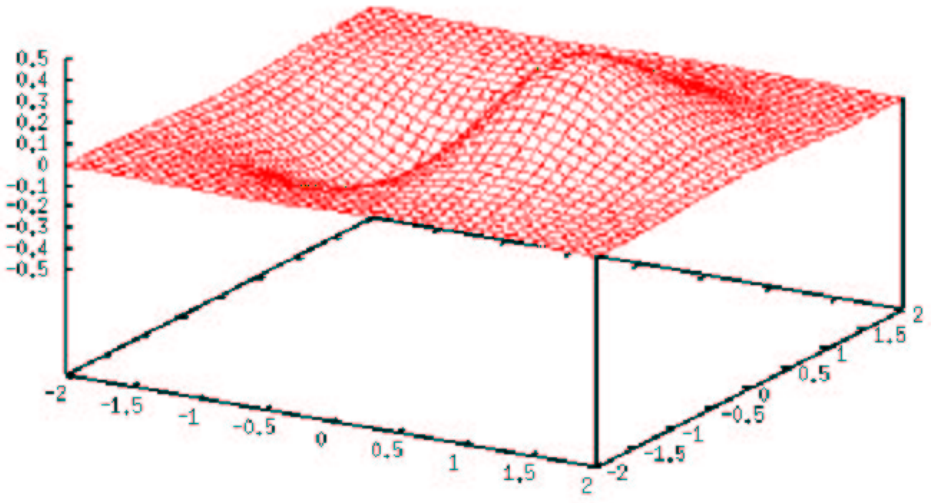


Figura D.3: Curvas de contorno.

c) Grillas

`grid`	Incluye o borra una grilla de referencia en un gráfico bidimensional. `grid ''on''` coloca la grilla y `grid ''off''` la saca. `grid` equivale a `grid ''on''`.

Al usar `gnuplot`, el gráfico mostrado en pantalla no es actualizado automáticamente. Para actualizarlo y ver las modificaciones efectuadas, hay que dar la instrucción `replot`.

Los siguientes comandos permiten manipular las ventanas gráficas:

`hold`	Permite "congelar" la figura actual, de modo que sucesivos comandos gráficos se superponen sobre dicha figura (normalmente la figura anterior es reemplazada por la nueva). `hold on` activa este "congelamiento", y `hold off` lo desactiva. `hold` cambia alternativamente entre el estado `on` y `off`.
`closeplot`	Cierra la ventana actual.

Finalmente, si se desea guardar un gráfico en un archivo, se puede proceder del siguiente modo si Octave está generando los gráficos con `gnuplot` y se trabaja en un terminal con XWindows. Si se desea guardar un gráfico de la función $y = x^3$, por ejemplo:

```
x = linspace(1,10,30);
y = x.^3;
plot(x,y);
```

```
gset term postscript color
gset output ``xcubo.ps''
replot
gset term x11
```

Las tres primeras líneas son los comandos de Octave/Matlab convencionales para graficar. Luego se resetea el terminal a un terminal postscript en colores (`gset term postscript` si no deseamos los colores), para que el output sucesivo vaya en formato postscript y no a la pantalla. La siguiente línea indica que la salida es al archivo `xcubo.ps`. Finalmente, se redibuja el gráfico (con lo cual el archivo `xcubo.ps` es realmente generado), y se vuelve al terminal XWindows para continuar trabajando con salida a la pantalla.

Debemos hacer notar que no necesariamente el gráfico exportado a *Postscript* se verá igual al resultado que `gnuplot` muestra en pantalla. Durante la preparación de este manual, nos dimos cuenta de ello al intentar cambiar los estilos de línea de `plot`. Queda entonces advertido el lector.

D.6.8. Strings

Para manipular una cadena de texto, disponemos de los siguientes comandos:

`lower`	Convierte a minúsculas
`upper`	Convierte a mayúsculas

Así, `lower('Texto')` da `'texto'`, y `upper('Texto')` da `'TEXTO'`.

Para comparar dos matrices entre sí, usamos `strcmp`:

`strcmp(a,b)`	1 si `a` y `b` son idénticas, 0 en caso contrario

Podemos convertir números enteros o reales en strings, y strings en números, con los comandos:

`int2str`	Convierte entero en string
`num2str`	Convierte número en string
`str2num`	Convierte string en número

Por ejemplo, podemos usar esto para construir un título para un gráfico:

```
s = ['Intensidad transmitida vs. frecuencia, n = ', num2str(1.5)];
title(s);
```

Esto pondrá un título en el gráfico con el texto:
`Intensidad transmitida vs. frecuencia, n = 1.5`.

D.6.9. Manejo de archivos

Ocasionalmente nos interesará grabar el resultado de nuestros cálculos en archivos, o utilizar datos de archivos para nuevos cálculos. El primer paso es abrir un archivo:

```
archivo = fopen('archivo.dat','w');
```

Esto abre el archivo `archivo.dat` para escritura (`'w'`), y le asigna a este archivo un número que queda alojado en la variable `archivo` para futura referencia.

Los modos de apertura posibles son:

`r`	Abre para lectura
`w`	Abre para escritura, descartando contenidos anteriores si los hay
`a`	Abre o crea archivo para escritura, agregando datos al final del archivo si ya existe
`r+`	Abre para lectura y escritura
`w+`	Crea archivo para lectura y escritura
`a+`	Abre o crea archivo para lectura y escritura, agregando datos al final del archivo si ya existe

En un archivo se puede escribir en modo binario:

`fread`	Lee datos binarios
`fwrite`	Escribe datos binarios

o en modo texto

`fgetl`	Lee una línea del archivo, descarta cambio de línea
`fgets`	Lee una línea del archivo, preserva cambio de línea
`fprintf`	Escribe datos siguiendo un formato
`fscanf`	Lee datos siguiendo un formato

Referimos al lector a la ayuda que proporciona Octave/Matlab para interiorizarse del uso de estos comandos. Sólo expondremos el uso de `fprintf`, pues el formato es algo que habitualmente se necesita tanto para escribir en archivos como en pantalla, y `fprintf` se puede usar en ambos casos.

La instrucción

```
fprintf(archivo,'formato',A,B,...)
```

imprime en el archivo asociado con el identificador `archivo` (asociado al mismo al usar `fopen`, ver más arriba), las variables `A`, `B`, etc., usando el formato `'formato'`. `archivo=1` corresponde a la pantalla; si `archivo` se omite, el *default* es `1`, es decir, `fprintf` imprime en pantalla si `archivo=1` o si se omite el primer argumento.

`'formato'` es una string, que puede contener caracters normales, caracteres de escape o especificadores de conversión. Los caracteres de escape son:

`\n`	New line
`\t`	Horizontal tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\\`	Backslash
`\'`	Single quote

Por ejemplo, la línea

```
fprintf('Una tabulacion\t y un \'\'original\'\' cambio de linea\n aqui\n')
```

da como resultado

```
Una tabulacion       y un ''original'' cambio de linea
aqui
```

Es importante notar que por *default*, el cambio de línea al final de un `fprintf` no existe, de modo que, si queremos evitar salidas a pantalla o a archivo poco estéticas, siempre hay que terminar con un `\n`.

Los especificadores de conversión permiten dar formato adecuado a las variables numéricas `A`, `B`, etc. que se desean imprimir. Constan del caracter `%`, seguido de indicadores de ancho (opcionales), y caracteres de conversión. Por ejemplo, si deseamos imprimir el número π con 5 decimales, la instrucción es:

```
fprintf('Numero pi = %.5f\n',pi)
```

El resultado:

```
Numero pi = 3.14159
```

Los caracteres de conversión pueden ser

`%e`	Notación exponencial (Ej.: `2.4e-5`)
`%f`	Notación con punto decimal fijo (Ej.: `0.000024`)
`%g`	`%e` o `%f`, dependiendo de cuál sea más corto (los ceros no significativos no se imprimen)

Entre `%` y `e`, `f`, o `g` según corresponda, se pueden agregar uno o más de los siguientes caracteres, en este orden:

- Un signo menos (`-`), para especificar alineamiento a la izquierda (a la derecha es el *default*).

- Un número entero para especificar un ancho mínimo del campo.
- Un punto para separar el número anterior del siguiente número.
- Un número indicando la precisión (número de dígitos a la derecha del punto decimal).

En el siguiente ejemplo veremos distintos casos posibles. El output fue generado con las siguientes instrucciones, contenidas en un *script*:

```
a = .04395;
fprintf('123456789012345\n');
fprintf('a = %.3f.\n',a);
fprintf('a = %10.2f.\n',a);
fprintf('a = %-10.2f.\n',a);
fprintf('a = %4f.\n',a);
fprintf('a = %5.3e.\n',a);
fprintf('a = %f.\n',a);
fprintf('a = %e.\n',a);
fprintf('a = %g.\n',a);
```

El resultado:

```
12345678901234567890
a = 0.044.
a =       0.04.
a = 0.04      .
a = 0.043950.
a = 4.395e-02.
a = 0.043950.
a = 4.395000e-02.
a = 0.04395.
```

En la primera línea, se imprimen tres decimales. En la segunda, dos, pero el ancho mínimo es 10 caracteres, de modo que se alínea a la derecha el output y se completa con blancos. En la tercera línea es lo mismo, pero alineado a la izquierda. En la cuarta línea se ha especificado un ancho mínimo de 4 caracteres; como el tamaño del número es mayor, esto no tiene efecto y se imprime el número completo. En la quinta línea se usa notación exponencial, con tres decimal (nuevamente, el ancho mínimo especificado, 5, es menor que el ancho del output, luego no tiene efecto). Las últimas tres líneas comparan el output de `%f`, `%e` y `%g`, sin otras especificaciones.

Si se desean imprimir más de un número, basta agregar las conversiones adecuadas y los argumentos en `fprintf`. Así, la línea

```
fprintf('Dos numeros arbitrarios: %g y %g.\n',pi,exp(4));
```

da por resultado

```
Dos numeros arbitrarios: 3.14159 y 54.5982.
```

Si los argumentos numéricos de `fprintf` son matrices, el formato es aplicado a cada columna hasta terminar la matriz. Por ejemplo, el *script*

```
x = 1:5;
y1 = exp(x);
y2 = log(x);
a = [x; y1; y2];
fprintf = ('%g  %8g %8.3f\n',a);
```

da el output

```
1   2.71828    0.000
2   7.38906    0.693
3   20.0855    1.099
4   54.5982    1.386
5   148.413    1.609
```

Apéndice E

Herramientas básicas en el uso de L.A.M.P.

L.A.M.P. es la sigla popular para referirse al uso conjunto: Linux+Apache+Mysql+PHP (también ésta ultima se reemplaza por Python o Perl) .

E.1. Objetivo.

Se espera que tras leer este apéndice el lector interesado sea capaz de:

- Elaborar sus propias paginas web utilizando *PHP*, a partir de las cuales se pueda intercambiar información de manera segura mediante el sistema gestor de base de datos *MySql*.

E.2. Prerequisitos

- Conocimiento básico sobre qué es un navegador, es decir, tener cierto tiempo utilizándolo.

- Tener instalados y operando conjuntamente los siguientes programas:

  - *Apache* 1.3 o superior.
  - *MySql* 4 o superior.
  - *PHP* 4 o superior.

E.3. Breve referencia sobre paginas web.

Un navegador es un *software* que mediante la conexión a Internet interpreta *scripts* presentes en el servidor web, desplegando el resultado en pantalla. Dichos *scripts* normalmente se encuentran escritos en lenguaje *html*.

La gran ventaja que ofrecen las páginas *web* son los llamados *hipervínculos* o *links*: objetos que permiten saltar entre diferentes páginas web de manera fácil. Gracias a los *links* existen los "árboles de páginas": diferentes *scripts* interconectados mediante *links* correspondientes a un mismo dominio[1]. Una página como la anteriormente descrita es capaz de desplegar información en un navegador o recibirla a través de formularios. Este tipo de páginas reciben el nombre de páginas *estáticas*, es decir, si se quiere cambiar la información desplegada, se está obligado a modificar el *script* en *html*.

En contraste, existen páginas que cambian su información dependiendo de cuándo o cómo son ejecutadas, recibiendo el nombre de páginas *dinámicas*. Para lograr el dinamismo, el servidor, tras leer una serie de archivos, genera un nuevo código en *html* a tiempo de ejecución. Ejemplos de dichas páginas son aquellas visitadas a diario por cualquier persona: buscadores, bancos en línea, correos electrónicos revisados desde un navegador, etc. Para generar este tipo de páginas existen muchos lenguajes de programación, sin embargo, el más utilizado y sobre la que se introducirá aquí es el lenguaje *PHP*.

E.3.1. Ejemplos

Página Estática en *html*.

- Archivo hola.html

```
<html>
<title> :::::El Clasico Hola mundo::: </title>
<body>  Hola mundo! </body>
</html>
```

El ejemplo anterior corresponde a lo más simple en una página web en *html*; notar que el *script* está estructurado mediante el uso de *etiquetas* del tipo: `<algo> .... </algo>`.

Página Dinámica en *PHP*

- Archivo hola.php

```
<html>
<?php $h="Hola mundo!"; ?>
<title> :::::El Clasico  <?php echo $h; ?>::: </title>
<body> <?php echo $h; ?>  </body>
```

[1] Nombre del sitio Internet.

```
</html>
```

Si se procede a ejecutar ambos *scripts* probablemente no se aprecie ninguna diferencia; sin embargo, el proceso que se ha llevado a cabo por el servidor establece fuertes diferencias entre ambos códigos. Notar que el código en *PHP* se encuentra inserto por completo en el código *html*. Por lo anterior, es necesario tener, al menos, un conocimiento básico de *html* para comenzar a programar en *PHP*.

E.4. Administrador de Bases de datos.

Una base de datos es una manera ordenada de guardar cualquier tipo de información para, de este modo, facilitar su búsqueda posterior. El encargado de buscar la información de manera efectiva es el *administrador de la base de datos*; en nuestro caso, el administrador corresponderá al *software MySql*. La información de la base de datos se almacena en matrices llamadas *tablas*, conformadas por columnas definidas. Las diferentes filas de una tabla se van constituyendo conforme se agregan nuevos registros; la información contenida en los registros corresponde a un conjunto de *strings* o números.

E.5. Servidor Web.

Un servidor Web es un *software* que opera en la máquina remota. El servidor posee la información contenida en la página y su función es proporcionar al internauta el contenido de ésta. Para efectos de este apéndice, esto se reduce a interpretar el código en *PHP* y generar en tiempo de ejecución el nuevo *script* en *html*.

Creando *scripts*.

En las secciones posteriores se listarán los comandos que permiten generar *scripts*, el modo de chequearlos y, de esta forma, aprender mediante ensayo y error es:

- crear la carpeta `~/public_html`.
- Escribir un *script* en el directorio antes citado utilizando algún editor[2].
- abrir el navegador e ir a la URL `http://nombredelhost/~nombredelusuario/pathdelscript`.

Luego de seguir estos pasos, el navegador desplegará en la pantalla la ejecución del *script*. Es importante señalar que el navegador no acusa errores de manera tan implacable como compiladores u otros intérpretes (p.ej. *Python*), por lo cual se recomienda ser especialmente riguroso.

[2]Existen varios editores especializados en lenguajes web sobre los cuales conviene averiguar un poco.

E.6. Páginas Básicas en *html.*

Prácticamente todo el código en *html* corresponde a aspectos estéticos de la página, sobre lo cual no se profundizará. Entre los objetivos que se buscan alcanzar en este apéndice, presentan especial relevancia los formularios, pues permiten introducir información proporcionada por un internauta a la base de datos.

E.6.1. Estructura de una página en *html.*

Si bien el navegador es capaz de interpretar correctamente código en *html* escrito sin seguir las reglas, es importante al menos saber algunas.

Todo el diseño de la página se encuentra entre etiquetas del tipo `<algo> .... </algo>`. Las más relevantes y que le dan estructura son:

- `<html>`: Esta etiqueta delimita en qué parte del *script* comienza y termina el código en *html.*
- `<title>`: Lo que se escriba dentro de esta etiqueta conformará el título de la página, es decir, el nombre que aparecerá en el título de la ventana del navegador.
- `<head>`: Contiene etiquetas y contenidos del encabezado. Principalmente datos que no aparecen en la página, pero que son relevantes.
- `<body>`: Contiene la información que será desplegada en la pantalla, ya sea texto imágenes, sonido, etc.

Cabe destacar que ninguna de las etiquetas mencionadas es obligatoria; puede precindirse de ellas si tan sólo se quiere escribir texto sin ninguna estructura.

E.6.2. Algo de estilo.

Las etiquetas utilizadas para dar estilo al texto dentro del cuerpo de la página (*i.e.* etiqueta `<body>`) son:

Propiedades del texto.

- `<p>`: Delimita un párrafo que finalizará al cerrarse la etiqueta. Esta etiqueta admite opciones especiales de alineación tales como: `<p align="center">`, la cual centra el párrafo. Las variantes obvias de las otras alineaciones son dejadas al lector.
- `<h1>`:Delimita un título de porte variable en dependencia del número que se ponga acompañando a la letra `h`, dicho número debe estar entre 1 y 6.

- `<br>`: Introduce un salto de línea. A diferencia de las etiquetas anteriores, ésta no tiene una etiqueta de cerrado.
- `<hr>`:Introduce una línea horizontal. Al igual que en la etiqueta anterior, ésta es desapareada.
- `<b>`: Todo lo escrito dentro de esta etiqueta quedará en negritas.
- `<em>`: Convierte en itálica todo el texto dentro de esta etiqueta.
- `<u>`: Subraya el texto dentro de la etiqueta.
- `<sub>`: Convierte en subíndice los caracteres dentro de esta etiqueta.
- `<sup>`: Convierte en superíndice los caracteres delimitados por la etiqueta.
- `<font>`: Etiqueta que permite definir atributos sobre el texto, tales como el porte o el color. Por ejemplo, si se requiere texto en rojo: `<font color="red">`.

Propiedades globales de la Página.

Con las etiquetas anteriormente explicadas es posible crear una página con información de manera relativamente ordenada, mas no estética. Un primer paso en esta última dirección es lo que se tratará a continuación.

Todos los atributos globales corresponden a opciones de la etiqueta `<body>`. Lo que se hará es definir los colores de: el texto, los *links*, *links* ya usados y el fondo. En *html* los colores se especifican mediante un código (para conocer el código correspondiente a cada color puede consultarse la tabla de colores (E.2) al final de este apéndice[3]).Lo anterior puede apreciarse en el siguiente ejemplo:

```
<body bgcolor="#000000" text="#ffffff" link="#ffff33" alink="#ffffcc">
```

El campo *bgcolor* corresponde al color de fondo; *text* al color del texto; *link* y *alink*, a los colores de los *links* por visitar y visitados respectivamente.

Alternativamente, es posible poner una foto de fondo de página, simplemente hay que suplir *bgcolor* por:

```
<body background="fondo.jpg">
```

Se recomienda poner fotos pequeñas que sean visualmente agradables como mosaicos, de lo contrario, puede variar el cómo se vean dependiendo del navegador, además de hacer más pesada la página. El siguiente ejemplo utiliza las herramientas desarrolladas.

[3]La cual por razones obvias debe ser vista a color.

Ejemplo

- Archivo ejemplo2.html

```
<html>
<title> :::Herramientas de estilo::: </title>
<body bgcolor="#336699" text="#000033" link="#660000" alink="#33ff00">
<h1 align="center" > <font color="red">
Sobre lo que se hablará en esta página estática
</font> </h1>
<p align="right">
<em> ...Aquí por ejemplo una cita para comenzar</em>
</p>
<br>
<p algin="center">
Escribiendo la parte medular de la página.....<br>
Es posible escribir una formula sin caracteres especiales como la siguiente:
<p align="center">(a<sub>11</sub>+a<sub>22</sub>+....)
<sup>2</sup>=(traza)<sup>2</sup></p>
<p align="left">  Finalmente, se espera un manejo
<b> básico </b> de <em> html </em> si se ha logrado comprender
<u> este ejemplo por completo </u></p>
</body>
</html>
```

Otros recursos.

Como se señaló en la introducción, la característica más relevante de una página *web* es su capacidad de interconexión con otras mediante *links*. Asimismo, existen otros recursos ampliamente usados que mejoran el aspecto y orden de una página en *html*.

Insertando una imagen

Para insertar una imagen existe una etiqueta desapareada[4], en ésta debe darse el *path* relativo del archivo gráfico de la siguiente forma:

```
<img src="path">
```

También se pueden especificar atributos adicionales, tales como: la alineación, el espacio vertical y horizontal utilizado por la foto:

```
<img src="path" align="left" hspace="20" vspace="30">
```

[4]No tiene otra etiqueta de cierre

En la orden anterior, la imagen (dentro del espacio que puede utilizar) se encuentra alineada a la izquierda y tiene un marco de `"20"` horizontal por `"30"` vertical.

Links.

El enlace, es decir, el espacio de la página donde el cursor del *mouse* cambia y permite acceder a la página siguiente, puede corresponder tanto a texto como a una imagen:

- **Enlace en el texto:** Para esto existe la etiqueta:

  ```
  <a href="path_a_la_página_en_cuestion">texto clickeable del link</a>
  ```

  Los *path* pueden ser relativos[5] si se trata de material presente en la misma máquina; de tratarse de un enlace externo a la página, debe especificarse la `URL` completa.

- **Enlace sobre una imagen:** Opera exactamente de la misma manera que un enlace de texto y solo cambia el argumento dentro de la etiqueta.

  ```
  <a href="path_a_la_página_en_cuestion"><img src="path"></a>
  ```

Tablas.

Una tabla permite administrar el espacio en una página de manera eficiente y es especialmente útil cuando se quiere una página ordenada y sin muchas características gráficas. La tabla es delimitada por la etiqueta `<table>`, dentro de ésta, las filas quedan delimitadas por la etiqueta `<tr>`. A su vez, el elemento de cada columna queda atrapado en la etiqueta `<td>`. Dentro de los elementos de la tabla es posible utilizar prácticamente cualquier etiqueta. A continuación, un esquema sobre cómo se programa una tabla:

<table>
<tr><td>

<tr>	<td> "a11"</td>	<td>"a12"</td>	<td>"a13"</td>	<td>"a14"</td> </tr>
<tr>	<td> "a21"</td>	<td> "a22"</td>	<td> "a23"</td>	<td> "a24"</td> </tr>
<tr>	<td> "a31"</td>	<td> "a32"</td>	<td> "a33"</td>	<td> "a34"</td> </tr>
<tr>	<td> "a41"</td>	<td> "a42"</td>	<td> "a43"</td>	<td> "a44"</td> </tr>

</td></tr>
</table>

Figura E.1: Esquema de una tabla en *html*, utilizando los elementos de una matriz.

[5] Respecto al código que se está ejecutando

E.6.3. Formularios.

Toda la preparación previa que se ha llevado a cabo tiene como fin el proveer de interfaz gráfica a la página dinámica. El instrumento que permitirá recibir información desde el visitante son los formularios, el qué se hace con dicha información escapa de las posibilidades de *html*. Para procesar la información debe recurrirse a otros lenguajes.

Manejo de la información utilizando el formulario.

Todo formulario se encuentra definido dentro de la etiqueta `<form>`, la cual contiene algunos atributos que especifican qué hacer con la información. Sin entrar en complicaciones, sólo se señalará que para poder recuperar la información contenida en un formulario deben definirse los siguientes atributos:

```
<form method="post" action="path_del_archivo_que_recupera_las_variables.php">
```

El atributo `method="post"` determina de qué forma es almacenada la información; el atributo `action="archivo.php"`, indica el nombre del archivo al cual es exportada dicha información.

Diseño de formularios.

Existen una serie de formularios que deben escogerse según lo requiera la clase de página que se éste programando. A continuación se listan las etiquetas para implementar los de mayor uso, recuérdese que se está dentro de un formulario y por lo tanto dentro de una etiqueta `<form>`.

- **Cajas de texto:** Para crear una dentro del formulario basta escribir:

  ```
  <input type="text" name="Nombre_de_la_información">
  ```

  Si bien son válidas otras etiquetas adicionales, cabe destacar que el atributo `type` también admite otras alternativas de rellenado tales como *password*, el cual oculta el texto introducido. Usualmente, la información recaudada en los formularios es procesada de manera algorítmica, por lo que conviene dar menos libertad sobre qué información es ingresada por el usuario. Para ello se le hace optar, como se muestra en los 3 siguientes diseños de formularios.

- **Listado de opciones:** La sintaxis es la siguiente:

```
<select name="escoga">
<option value="op1">nombrealt1</option>
<option value="op2">nombrealt2</option>
<option value="op3">nombrealt3</option>
<option value="op4">nombrealt4</option>
</select>
```

Recordar que el listado de opciones debe estar dentro de la etiqueta de formulario, de lo contrario, la información obtenida de éste no irá a ningún lado. El atributo `value` corresponde al nombre que se ha asignado a esa alternativa al procesar la información, es decir, si por ejemplo la `nombrealt1` es escogida, se registrará `op1` como valor de la variable `escoja` cuando se procese el formulario.

- **Botones de radio:** La gran ventaja de éste sistema es que se obliga al internauta a optar, la sintaxis es:

```
<input type="radio" name="escoja" value="op1">nombrealt1
<br>
<input type="radio" name="escoja" value="op2">nombrealt2
```

Sobra decir que un formulario puede también ser mixto, es decir, contener listados de opciones cajas de textos y/o botones de radio.

Envió de la información.

Una vez que se ha finalizado la definición del formulario, debe agregarse un botón que envíe la información, el cual corresponde a otro *input type*. Adicionalmente, si el formulario es muy largo es posible agregar un botón que ponga el formulario en blanco. Las sintaxis son las siguientes:

- **Botón de envió de la información:** `<input type="submit" value="boton_de_envio">`
- **Botón de *reseteo*:** `<input type="reset" value="resetear">`

Ejemplos

Con las nuevas herramientas es posible construir páginas como las expuestas en los siguientes ejemplos: usando el primer ejemplo de esta sección como un archivo de nombre `ejemplo2.html`, se puede construir un pequeño árbol de páginas con los siguientes 2 *scripts*. El primer *script* requiere una foto llamada `inicio.jpg` en el directorio local.

- Archivo ejemplo3.html

```
<html>
<title> :::Utilizando los nuevos recursos::: </title>
<body bgcolor="#666666" text="#660000" link="#66FF00" alink="#660066">
<hr>
<table align="center"><tr><td><a href="ejemplo2.html"> ejemplo<br>
anterior </a></td><td> <a href="ejemplo4.html">
<img src="inicio.jpg" align="center" hspace="30" vspace="20"></a>
</td> <td> <a href="http://www.uchile.cl">enlace externo</a>
</td>
</tr></table> <hr>
</body>
</html>
```

- Archivo ejemplo4.html

```
<html>
<title>.::Ejemplo del uso de formularios tipo caja de texto::.</title>
<body bgcolor="#000000" text="#FFFFFF" link="#66FF00" alink="#00FF99">
<br>
<h2 align="center"> Complete con sus datos y gane!<h2>
<form " method="post" action="procesa.php">
<table align="center">
<tr><td> Nombre:</td><td><input type="text" name="Nombre"></td></tr>
<tr> <td>e-mail:</td><td> <input type="text" name="email"></td></tr>
<tr><td>Telefono:</td><td> <input type="number" name="fono"></td></tr>
<tr><td>Dirección:</td><td> <input type="text" name="direc"></td></tr>
<tr><td></td><td><input type="submit" value="enviar"></td></tr>
</form>
</table>
</body>
</html>
```

Ejemplo de formularios, tipo listado de alternativas:

- Archivo ejemplo4b.html

```
<html><title>::Listado de opciones:::</title>
<body bgcolor="#000000" text="#FFFFFF" link="#66FF99" alink="#660033">
<h1 algin="left"><u>Particulas subatomicas</u></h1>
<form method="post" action="psa.php">
<table><tr><td>Particula subatomica:</td>
<td><select name="psa"><option value="proton">protón</option>
<option value="neutron">neutrón</option><option value="electron">electrón</option>
```

```
</td></tr><tr><td></td><td>
<input type="submit" value="ver"></td></table></form></body></html>
```

A partir de estas herramientas básicas en la programación de páginas *html*, es posible comenzar a introducir elementos que cambien dentro de la página a tiempo de ejecución. Esto es, la programación en *PHP*. Sin embargo, primero debe explorarse brevemente el lenguaje que permite comunicarse con la base de datos.

E.7. *MySql.*

E.7.1. Iniciando sesión.

MySql es un administrador de base de datos que tiene una estructura de usuarios similar a la de **UNIX**: existe un superusuario llamado *root* y los usuarios ordinarios, cada uno con su propia *cuenta*. Se supondrá en los siguientes ejemplos la posesión de una cuenta cuyo nombre de usuario es *lamp* y de palabra clave *bd*. Para comenzar a interactuar con el administrador de bases de datos, hay que iniciar una sesión en éste desde la consola del siguiente modo:

```
usa@host:$mysql -u lamp -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6 to server version: 4.0.24_Debian-10sarge2-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

La orden que se da desde la consola es: `usa@host:$mysql -u lamp -p`, que quiere decir: "comienza una sesión de *MySql* con el nombre de usuario *lamp* (`-u lamp`) y pide el *password* (`-p`)". La última línea corresponde al *prompt* de *MySql*.

E.7.2. Creando una base de datos.

A partir de aquí puede interactuarse con *MySql*; para hacerlo debe conocerse el lenguaje de interacción llamado *Sql*, del cual tan sólo se enseñarán las ordenes básicas. Cabe destacar que todo lo que se escriba en la consola de *MySql* debe finalizar con " ;". El primer paso es crear una base de datos, de nombre *base*, para ello debe introducirse la orden:

```
mysql> create database base;
```

Una vez creada la base de datos se puede comenzar a trabajar. Un usuario de *MySql* puede tener varias bases de datos dentro de su cuenta, es por ello que cada vez que éste se conecte a *MySql* (*i.e.* se *loguee*) debe escoger la base de datos que utilizará; desde la consola esto se hace escribiendo:

```
mysql> connect base;
```

E.7.3. Creando tablas.

La base de datos será completamente inútil si no se han creado tablas. Probablemente esto constituye el paso más complicado en el abordaje de un problema a solucionar con una base de datos, pues es en la estructura de las tablas que quedará plasmado el esquema bajo el cual operará lo que se programe. Por ello se recomienda fuertemente pensar este punto antes que cualquier otro. Para crear una tabla debe especificarse (Al menos): el nombre de la tabla, el nombre de cada campo y el tipo de cada campo. Supóngase el caso simple de una tabla con teléfonos llamada "agenda", esta tabla debe contener al menos dos campos: **nombres**, que contenga texto, y **teléfono**, que contenga números. Para crear tal tabla debe introducirse:

```
mysql> create table agenda(nombre text, telefono int);
```

O en general:

```
mysql> create table nombretabla(campo1 tipo, campo2 tipo,.......,campo_i tipo);
```

Podría juzgarse el ejemplo demasiado simple, pues no queda claro de cuántos tipos pueden ser los campos en una tabla; como la intención no es extenderse, se recomienda instalar el *script* en *PHP* llamado *phpmyadmin*[6], el cual permite una administración de la base de datos desde el navegador de una forma bastante más intuitiva y didáctica que simplemente interactuando con la consola. No se debe abusar del uso de esta interfase, pues es fundamental conocer la sintaxis para interactuar con la base de datos ya que justamente son éstas órdenes las que se incluyen en los *scripts*.

E.7.4. Interactuando con la Tabla.

Teniendo la capacidad para crear bases de datos y, dentro de éstas, crear tablas, lo que resta es aprender a: **insertar**, **buscar** y **remover** registros de una tabla. Para ilustrar estas acciones se continuará con el ejemplo de la agenda telefónica.

- *Escribiendo nuevos registros en la tabla: sentencia*[7] `INSERT`.
  Supóngase que se quiere ingresar un nuevo usuario llamado "Pedro" de número telefónico "5437896". Para ello debe escribirse en la consola:

```
mysql> insert into agenda(nombre,telefono) values ("Pedro",5437896);
```

[6]Disponible para la distribución **Debian GNU/Linux**.
[7]Tanto *MySql* como *html* no son sensibles a las mayúsculas al escribir *scripts*.

o en general:

```
mysql> insert into nombre_tabla(campo1,campo2,.....,campoj,..)
values (valcampo1,valcampo2,....,valcampoj,...);
```

Notar que los *string* deben ir entre comillas dobles o simples, no así los números. Si se escribe dos veces el mismo registro, la base de datos guardará dos registros diferentes con exactamente la misma información y los diferenciará por su fecha de ingreso.

- *Removiendo registros: sentencia* `DELETE`.
  Supóngase que se quiere borrar exclusivamente a Pedro; para ello debe escribirse:

  ```
  mysql> delete from agenda where nombre='Pedro';
  ```

  Notar la aparición de la condición `WHERE`, la cual especifica a quién borrar; de truncar la sentencia antes de esta condición, *MySql* borrará todos los registros de la tabla. Como es de esperarse, se podria haber identificado a quién se quería eliminar mediante cualquier otro campo. El administrador se podría preguntar cómo borrar la cantidad de usuarios que quiera de manera efectiva haciendo algo similar a lo que permite `bash`, esto es posible y fácilmente deducible de la sentencia `select`.

- *Buscando registros en la base de datos: sentencia* `SELECT`.
  La sintaxis de la sentencia `SELECT` es la siguiente:

  ```
  mysql> select nomcampo from nomtabla where condiciones;
  ```

  Un par de ejemplos para ilustrar lo anterior:

  - Seleccionar todos los elementos: `mysql> select * from agenda;`
    Listará todos los elementos de la agenda.
  - Seleccionar todos los nombres de la agenda: `mysql> select nombre from agenda;`
    Listará todos los nombres de la agenda (ídem con los teléfonos).
  - Seleccionar el par (teléfono, nombre) del usuario "Andrea":
    `mysql> select telefono,nombre from agenda where nombre='Andrea';`
  - Operadores lógicos: también es posible ser más específico mediante operadores lógicos como `OR` o `AND`, los cuales funcionan como se esperaría. Esto sirve para, por ejemplo, corroborar la existencia de un registro. Supóngase que se conoce el teléfono "5534126" y se tiene la sospecha de que pertenece a "Andrea". Para ello puede digitarse:

    ```
    mysql> select * from agenda where nombre='Andrea' and telefono= 5534126;
    ```

Si no retorna ningún registro quiere decir que Andrea no tiene ese número. Alternativamente, para obtener más información se puede escribir:

```
mysql> select * from agenda where nombre='Andrea' or telefono= 5534126;
```

Listará todos los registros que cumplan alguna o ambas condiciones.

- Mostrar registros que contengan información diferente:

```
mysql> select ncampos DISTINCT FROM ntabla where condiciones
```

- Orden:
  La mayoría de las veces es necesario saber parcialmente de qué forma vendrán los datos listados, para ello es necesario introducir un orden y una tolerancia, de esta forma es posible controlar exactamente qué hacer aparecer en el resultado de la búsqueda. Supóngase que se quieren mostrar todos los elementos de la agenda ordenados alfabéticamente por nombre, para esto hay que escribir:

```
mysql> select * from agenda order by nombre;
```

  Queda claro que hace la condición `ORDER BY`. Supóngase que se desea invertir el orden y poner los últimos 3 registros, para ello debe escribirse:

```
mysql> select * from agenda order by nombre desc limit 3;
```

  La condición `DESC` exige un orden descendente, mientras que `LIMIT`, acompañado de un entero, da la cantidad máxima de registros que pueden aparecer como resultado de la búsqueda. Resulta especialmente importante hacer órdenes con números, pues permiten utilizar comparadores aritméticos. Supóngase que se tiene una tabla llamada *usuarios* que contiene los campos *edad* y *nombre* de personas. Para encontrar los mayores de edad debería escribirse:

```
mysql> select  * from usuarios where edad >= 18;
```

- *Actualización de datos, sentencia:* `UPDATE`.

Esta sentencia permite la actualización de los datos, es decir, toma un registro viejo y le modifica algún campo. Supóngase que en el ejemplo de la agenda telefónica se quiere cambiar el teléfono a un usuario llamado "Juan", quien tiene un nuevo teléfono "8571646". Para hacerlo debe introducirse la orden:

```
mysql> update agenda set telefono=8571646 where nombre='Juan';
```

Es evidente la función de `SET` en la sintaxis; de no especificarse `WHERE` y truncarse la frase, se cambiaría el teléfono de todos los registros por el nuevo número.

Funciones sobre los campos.

Existen, además, funciones que pueden aplicarse sobre los campos, las cuales pueden investigarse utilizando *phpmyadmin*. Se verá un ejemplo simple: en presencia de una tabla con información importante como contraseñas, sería necesario algún sistema de seguridad sobre ellas. Para esto existe la función `PASSWORD`, la cual encripta el argumento. La tabla se llama *registrados* y contiene 2 campos: *Nombre* tipo texto y *clave* tipo `VARCHAR`. Se sabe como crear una tabla e insertar registros. La siguiente sintaxis muestra la diferencia hecha por la función `PASSWORD`:

```
mysql> create table registrados (nombre text, clave VARCHAR(20));
mysql> insert into registrados (nombre, clave) values ('andres','te45');
mysql> select * from registrados;

+--------+-------+
| nombre | clave |
+--------+-------+
| andres | te45  |
+--------+-------+
1 row in set (0.00 sec)
```

Se procede a encriptar la clave.

```
mysql> update registrados set clave=PASSWORD('te45') where nombre='andres';
mysql> select * from registrados;
+--------+------------------+
| nombre | clave            |
+--------+------------------+
| andres | 37d3b95821add054 |
+--------+------------------+
```

Se ha expuesto lo más básico para poder interactuar con la base de datos. El conocimiento de esta sintaxis volverá más rápidos, seguros y eficientes lo programado; sin embargo, el alcance del apéndice utilizará tan sólo las herramientas aquí expuestas.

E.8. Programación en *PHP*.

Los 2 lenguajes antes vistos no tienen relación entre sí y, de hecho, no podrán funcionar en conjunto de no ser por el lenguaje *PHP*. Este lenguaje es interpretado por un módulo del servidor *web*, por ello, el código en *PHP* nunca abandona el servidor, es decir, el internauta no puede ver la fuente en *PHP* visitando la página.

Para el estudiante de este apéndice se espera un aprendizaje muy rápido de *PHP* por su similitud con `bash` y `C++`. En lo sucesivo, todo el trabajo de *PHP* será comunicarse con la base de datos y manipular los recursos de *html* de manera dinámica.

E.8.1. Lenguaje *PHP*.

Como ya fue señalado en la introducción, el lenguaje en *PHP* se encuentra inserto dentro del *html*. Esto significa que el *script* está escrito principalmente en *html* con trozos en *PHP*. Todo el código en *PHP* queda delimitado por las etiquetas:

`<?php... aqui todo el codigo en php .... ?>`.

Al igual que en `bash`, las variables van antecedidas por un símbolo $, por otra parte, al igual que en `C++` cada línea de código debe ir finalizada por un ";".

Si se regresa con esta nueva visión sobre el primer ejemplo de página dinámica expuesto en este apéndice, se tendrá:

Página Dinámica en *PHP*

- *Archivo hola.php*

```
<html>
<?php $h="Hola mundo!";?>
<title> :::::El Clasico  <?php echo $h; ?>::: </title>
<body> <?php echo $h; ?>  </body>
</html>
```

Lo que ocurre al escribir esta página en el navegador es lo siguiente: el servidor web procesa el *script* interpretando el código delimitado en *PHP*, generando a tiempo de ejecución el siguiente nuevo *script* en *html*, el que es interpretado por el navegador.

```
<html>
<title> :::::El clasico Hola Mundo!:::</title>
<body> Hola Mundo! </body>
</html>
```

Debe quedar completamente claro que el código en *PHP* jamás abandona el servidor *web*, por lo tanto, el código en *PHP* se encuentra inserto dentro del *html* y no viceversa. Dentro de un *script* en *html* es posible, las veces que sea necesario y donde sea necesario, escribir código en *PHP*.

E.8.2. Variables.

Las variables en *PHP* pueden ser *string*, números o arreglos. Para desplegar en pantalla una variable se utiliza la instrucción `echo`. El ejemplo anterior constituye un caso simple de esto.

E.8.3. Recuperando variables desde un formulario.

En el formulario del *ejemplo 4*, éste enviaba vía *post* las variables al archivo `procesa.php`. Ahora se procederá a crear dicho archivo operando sobre las variables.

Ejemplo

- Archivo procesa.php

```
<html><title> recuperando las variables </title>
<body bgcolor="#000000" text="#FFFFFF" link="#66FF00" alink="#00FF99">
<h2 align="center"> La información tirada a pantalla<h2>
<p align="center"> Tu nombre es <b><?php echo $_POST['Nombre'];?></b>
,vives en <b> <?php echo $_POST['direc']; ?> </b>.
Tu e-mail es <b> <?php echo $_POST['correo']; ?> </b>
, además tu teléfono es <?php echo $_POST['fono'];?></p></body> </html>
```

Este archivo simplemente toma las variables y las despliega en pantalla. Se debe notar que la información contenida en un formulario queda contenida en el grupo de variables `$_POST['nombredelavariable']`. Con este conocimiento es posible rehacer todo ejercicio propuesto en el capítulo de `C++` donde se pidan variables y opere con ellas en *PHP*. Cabe destacar todo lo que es posible mezclar *html* y *PHP*.

E.8.4. Control de flujo.

Las sintaxis de los controles de flujo más usuales en *PHP* son exactamente iguales a los ya conocidos de `C++`. A continuación se dan ejemplos de los bucles `while` y `if` respectivamente. Cabe destacar que el uso de " . " pega los *string*.

ejemplo *while*

```
<?php $i=1;
$j="Ejemplo de un bucle haciendo iteraciones, iteracion n°:";?>
<html><body bgcolor="#336699" text="000033" link="660000" alink="#33ff00">
<h2 align='center'> LOOP <em> while </em>. </h2>
<p align="right">
<?php while($i<10)//comentario en php
{echo $j . $i; $i++; ?> <br> <?php } ?>
</p></body></html>
```

ejemplo *if*

El condicional *if* logra que el *script* haga diferentes cosas según el valor de alguna variable. A fin de economizar código, a continuación se ejemplificará el uso del control de flujo `if` procesando la información introducida en el formulario del *ejemplo4b.html*. La idea es tener la información contenida en variables diferentes y que ésta sea desplegada en la pantalla según se elija. La sintaxis del `if` es exactamente igual que en `C++`. • Archivo *psa.php*

```
<?php
$opcion=$_POST['psa'];
//proton
$pmasa="1,672 * 10^27 kg";
$pcarga="1,60217653(14)*10^(-19)C";
$ps="-";
//neutron
$nmasa="1,672 * 10^27 kg";
$ncarga="0";
$ns="no tiene carga";
//electron
$emasa="9,10 * 10^(-31) kg";
$ecarga="1,60217653(14)*10^(-19)C";
$es="-";
//general
$masa;
$carga;
$signo;

if ($opcion=="proton")
{$masa=$pmasa;
$carga=$pcarga;
$signo=$ps;}
else if ($opcion=="electron")
 {$masa=$emasa;
$carga=$ecarga;
$signo=$es;}
else
{$masa=$nmasa;
$carga=$ncarga;
$signo=$ns;}
?>
<html><title> informacion</title>
<body bgcolor="#000000" text="#FFFFFF" link="#66FF99" alink="#660033">
<table><tr><td>La particula: </td><td> <?php echo $opcion; ?> </td></tr>
<tr><td>tiene una masa de :</td><td> <?php echo $masa; ?> </td></tr>
<tr><td>tiene una carga de signo :</td><td> <?php echo $signo; ?> </td></tr>
<tr><td>tiene una cantidad de carga :</td><td> <?php echo $carga; ?> </td></tr>
</table></html>
```

E.8.5. Función *require.*

Al igual que en todos los demás lenguajes estudiados en este curso, *PHP* posee funciones intrínsecas a él o que pueden crearse. En este apéndice tan sólo se hará uso de funciones que vienen ya incluidas en el lenguaje, pues son las primeras que deben conocerse. La función *require* pide como argumento algún archivo cuando se ejecuta el código. La función se encarga de incluir el archivo y evaluarlo dentro del código. Si el archivo en cuestión resulta ser más código, éste será ejecutado. Usualmente, esto es utilizado para pedir formularios; el uso de esta función se ejemplifica ampliamente en el ejemplo final.

E.8.6. Sesión.

PHP tiene la capacidad de definir variables globales sobre un conjunto de páginas a elección; para ello debe realizarse una *sesion*. Éstas son usualmente utilizadas cuando se posee una estructura de usuarios. Para poder crear una sesión debe contarse con: un árbol de páginas ya armado, una página donde se inicie la sesión, y una página donde se termine la sesión. Para ilustrar el uso de las sesiones se considerarán 4 páginas: el formulario del `ejemplo4.html`; el ejemplo anterior `procesa.php`(con un par de modificaciones) como página de inicio de sesión; una nueva página `ejemplo5.php`, que gracias a la sesión es capaz de recuperar las variables; y una página de cierre de sesión `salir.php`, la cual vuelve al `ejemplo4.html`. Es importante adquirir el hábito de generar árboles de páginas lo más intuitivos posible, pues éstas suelen ser usadas por personas con poco o nulo conocimiento de su construcción y gran parte de su éxito radicará en su simpleza. Sin más preámbulos, los ejemplos son los siguientes:

Ejemplos.

- Nuevo archivo `procesa.php`

```
<?php
session_start();//inicio de la sesión
header("Cache-control: private");
//esta línea se escribe para no borrar los formularios como lo hace i.e.
?>
<html><title> recuperando las variables </title>
<body bgcolor="#000000" text="#FFFFFF" link="#66FF00" alink="#00FF99">
<h2 align="center"> La información tirada a pantalla<h2>
<p align="center"> Tu nombre es <b><?php echo $_POST['Nombre'];?>
</b>, vives en <b> <?php echo $_POST['direc']; ?> </b>. Tu e-mail es <b>
<?php echo $_POST['correo']; ?>
</b>, además tu teléfono es <?php echo $_POST['fono'];?> </p>
<h1 align="center"> <a href="ejemplo5.php"> <font color="red">
Aqui para seguir </font></a></h1>
```

```
</body>
</html>
<?php
$_SESSION['nombre']= $_POST['Nombre'];
$_SESSION['mail']= $_POST['correo'];
//Definiendo las variables globales de session.?>
```

• Archivo `ejemplo5.php`

```
<?php
session_start();
?>
<html>
<body bgcolor="#000000" text="#FFFFFF" link="66FF00" alink="00FF99">
<table align="center">
<tr><td> <b>
Las variables aun se recuperan y son:
</b></td></tr>
<tr><td> <b>
El nombre era: <em>
<?php echo $_SESSION['nombre']; ?>
</em> </b></td> </tr>
<tr><td> <b>
El correo era: <em>
<?php echo $_SESSION['mail']; ?> </em>
</b></td></tr>
<h1 align="center">
<a href="salir.php">
<font color="red">
finalizar sesión.
</font></a></h1>
</table>
</body>
</html>
```

• archivo `salir.php`

```
<?php
session_start();
session_unset();
header("Location:ejemplo4.html");
echo"<html></html>";
exit;
```

```
?>
```

Un par de comentarios sobre los ejemplos: la función `session_start()` debe ser lo primero que aparezca en el *script*, es decir, lo primero registrado por el servidor. Para definir una variable de sesión basta asignarla como `$_SESSION['nombre']` la cual existe como variable global para todas las páginas que integren la sesión a partir desde donde fue definida la variable. Para salir se utiliza la función `session_uset()`, la cual destruye todas las variables de la sesión. En el ejemplo `salir.php` se ha utilizado un método que permite redireccionar la página de manera automática sobre el cual no se profundizará. Usualmente, por razones de seguridad, se requiere un árbol de páginas que sea cerrado, para ello simplemente basta definir una variable de sesión en la página de entrada y luego, en cada nueva página que integre la sesión, anteponer un condicional `if` que chequee que esa variable exista, es decir, chequea que se haya pasado por la primera de las páginas. Por ejemplo, si se define la variable de sesión `$_SESSION['estado']="conectado";` en la primera página, una forma de definir el condicional es:

```
if ( $_SESSION['estado']!= 'conectado' )
{die( "Ud no esta logueado!.Click aqui para <a href='ejemplo4.html'>volver</a>");}
```

Esto quiere decir que si la variable de sesión *'estado'* es diferente de conectado, niegue la entrada a la página y despliegue este mensaje. De lo contrario, se continuará procesando el código.

E.8.7. *PHP* interactuando con *MySql*.

Si se ha seguido el apéndice hasta aquí con éxito, el camino está casi completo. Para lograr el objetivo final tan sólo deben introducirse un par de funciones de *PHP* que permitirán la conexión a *MySql*. Para esto se esquematizarán los pasos que todo *script* que se conecta a la base de datos debe seguir:

1. Conectarse a *MySql*.
2. Escoger la base.
3. Determinar si se escribe o se lee desde la base de datos.
4. Escribir la petición como variable en un *string*.
5. Enviarla (si había que escribir en la base de datos, con esto es suficiente).
6. Si se está leyendo desde la base de datos, convertir el resultado en un arreglo y operar sobre la parte de él que se necesite.

Siguiendo la numeración respectiva, utilizando el nombre de usuario y contraseña de *MySql* antes citado, las funciones son:

1. `$conexion=mysql_connect('localhost','lamp', 'bd')or die('No es posible conectar'.mysql_error());`

   Se conecta al servidor de *MySql* local bajo el usuario "lamp". Podría parecer en principio un poco inseguro que aparezca la clave del usuario explícitamente en el *script*; sin embargo, recuérdese que esta parte del *script* está escrita en lenguaje *PHP* y por lo tanto jamás abandona la máquina donde se encuentra el *script*.

2. `mysql_select_db('nombredelabase')or die ('Base no encontrada');`
   Notar que, a diferencia del ítem anterior, aquí la selección de base no se almacena como variable.

3. Para leer o escribir en la base de datos, basta crear una variable de *string* con la sintaxis de lo requerido en lenguaje *sql*. Por ejemplo, supóngase que se quiere escribir en la agenda un nombre guardado en la variable `$nombre` y un teléfono almacenado en la variable `$telefono`; la sintaxis es:
   `$p="insert into agenda(nombre, telefono) values ('$nombre','$telefono')";`

4. Por otra parte, si quiere escogerse un registro particular, por ejemplo el número telefónico del usuario `$usuario` , la sintaxis es:
   `$u="select nombre,telefono from agenda where nombre='$usuario'";`

5. Independiente de si se quiera leer o escribir, si la petición está en una variable `$p`, esta se ejecuta en *MySql* mediante la orden:
   `$pedido=mysql_query($p) or die ('no se pudo');`

6. Supóngase que se pidió un registro, lo que se obtendrá de vuelta en la variable `$pedido` no es un número ni un arreglo. Para poder operar con la información, primero debe convertirse en un arreglo. Dos maneras diferentes de hacerlo son:

   - `$fila=mysql_fetch_row($pedido);`
   - `$arreglo=mysql_fetch_array($pedido);`

   En ambos casos, el arreglo se recorre de la misma manera que en `C++`. Por ejemplo, en el primer caso, si quiere obtenerse el primer elemento, éste corresponderá a la variable `$fila[0]`.En contraste, el segundo caso, permite seleccionar los elementos del arreglo por su nombre dentro de la tabla de *MySql*. Por ejemplo, si tenemos una tabla con los campos `nombre`, `dirección`, entonces los elementos del arreglo corresponden a `$arreglo['nombre']` y `$arreglo['dirección']` respectivamente. En la práctica es mucho más utilizada esta representación que la de *row*, pues es más fácil identificar los elementos. Finalmente, cabe señalar que en ambos casos, los arreglos respectivos, contienen un sólo registro, pese a que el *query*, contenga más de uno. Para obtener todos los registros arrojados por el *query*, basta recorrerlo con un *while*, de la siguiente forma.

```
//suponiendo conectado a la base,
//y con la sentencia sql escrita en un string.
$pedido=mysql_query($sentencia_mysql);
//supongáse que el query de arriba devuelve más de un registro,
// para obtener cada uno basta hacer
while($miarreglo=mysql_fetch_array($pedido){echo $miarreglo['campo'];}
?>
```

Lo anterior desplegará en pantalla el contenido de los sucesivos registros en el campo `campo`. Lo anterior, también funciona para `mysql_fetch_row`.

Con esta sección se da por finalizado el apéndice; sin embargo, se ha escrito un ejemplo final para ver puestas en práctica las herramientas aquí expuestas. Se invita al lector a tomarlo como una prueba final de si logró un aprendizaje real.

E.9. Ejemplo Final.

Se espera haber recorrido con éxito todos los tópicos de una manera superficial. A fin de aclarar cualquier concepto que para el lector haya quedado poco claro, se llevará a cabo un ejemplo simple, el cual debe cumplir el siguiente objetivo: un sistema en línea donde estudiantes que cursen un ramo puedan averiguar sus notas y promedio de manera personalizada mediante una clave. Se hubiera esperado un ejemplo más científico, sin embargo, todas las herramientas de este apéndice apuntan a tareas administrativas más que a otra cosa.

E.9.1. Paso I: Estructura de las tablas.

Para todo el ejemplo se tiene el mismo nombre y clave de usuario en *MySql* que en los ejemplos anteriores. El sistema debe contener una base de datos llamada "ramo" y dentro de ésta, por lo menos, dos tablas: una con los nombres y contraseñas de los diferentes alumnos, es decir, dos campos y otra con el nombre de cada estudiante y las notas de las respectivas evaluaciones. Para poner una cota superior se crearán 8 campos de evaluación. Adicionalmente, se creará una tabla que contenga tanto el nombre como la clave del administrador de este sistema. La creación de la base de datos y las respectivas tablas en la consola de *MySql* son por lo tanto:

```
mysql>create database ramo;
mysql>connect ramo;
mysql>create table alumnos(Nombre text,clave varchar(20));
mysql>create table notas(Nombre text ,nota1 float, nota2 float, nota3 float,
nota4 float, nota5 float, nota6 float, nota7 float, nota8 float);
mysql> create table administrador(Nombre text, clave varchar(20));
mysql> insert into administrador(Nombre,clave)
```

```
values ("administrador", password("admin"));
```

Ésta es toda la sintaxis que es necesario hacer desde la consola de *MySql*. Ahora toca "pensar el árbol de páginas" y la comunicación que tendrán éstas con la base de datos.

E.9.2. Paso II: árbol de páginas.

El árbol de páginas debe surgir de manera natural a partir de los objetivos que debe cumplir lo que se está programando. Éste se reducirá a formularios que permitan introducir información, la cual será procesada por otro *script* en *PHP*. Si bien existen 2 clases de usuarios (el administrador y los usuarios corrientes), todos deben pasar por una página que les permita "entrar" en el sistema, es decir, *'loguearse'*. El archivo expuesto a continuación es simplemente un formulario que envía la información a `logueo.php` para ser procesada.

• Archivo `log.html`

```
<html><title> Bienvenido </title><body bgcolor="#66FFFF" text="#660000">
<h1 align="left"> Proceda a identificarse </h1><br><br><br><hr>
<form method="post" action="logueo.php">
<table align="left">
<tr><td><b>Nombre de Usuario:</b></td><td>
<input type="text" name="nombre"></td></tr>
<tr><td><b>Contraseña:</b></td><td><input type="password" name="clave">
</td></tr>
<tr><td><b>tipo de usuario:</b></td>
<td><select name="usrcls"><option value="1">usuario</option>
<option value="2">administrador</option></td></tr>
<tr><td><input type="submit" value="iniciar sesion"></td></tr>
</table></form><br><br><br><br><br><hr></body></html>
```

El formulario recién expuesto pide 3 datos: el nombre, la contraseña y el tipo de usuario que está intentando *'loguearse'*. El *script* que procesa esta información debe crear una sesión, pues recuérdese que esto debe ser lo primero que aparezca en el *script*. Luego debe chequear que la clave ingresada corresponda a un usuario existente, sin embargo, también debe realizar diferentes acciones si se trata de un usuario común o del administrador, por lo anterior, este *script* funcionará en base al condicional `if`. No debe sorprender la cantidad de código, pues, la idea es una sola, cambiando los nombres de las variables:

• Archivo `logueo.php`

```
<?php session_start();
$nombre=$_POST['nombre'];
$clave=$_POST['clave'];
$tipo=$_POST['usrcls'];
```

```
$conexion=mysql_connect('localhost','lamp', 'bd')or
die('No es posible conectar'.mysql_error());
mysql_select_db('ramo')or die ('error al conectar a la base');
if($tipo==1)
{$p="select Nombre,clave from alumnos where Nombre='$nombre' and
clave=PASSWORD('$clave')";
$q=mysql_query($p) or die('no se pudo hacer el pedido');
$b=mysql_fetch_row($q);
if($b==false)
{echo "usted no es reconocido por el sistema";
?> <a href="log.html"><font color="red"> Volver</font></a>
<?php}
else
{$_SESSION['estado']="conectado"; require('notas.php');}}
else if($tipo==2)
{$p="select Nombre,clave from administrador where Nombre='$nombre'
and clave=PASSWORD('$clave')";
$q=mysql_query($p) or die('no se pudo hacer el pedido');
$b=mysql_fetch_row($q);
if($b==false)
{echo "usted no es reconocido por el sistema";
?> <a href="log.html"><font color="red"> Volver</font></a><?php}
else{$_SESSION['estado']="conectado"; require('acciones.php');}}
else{require("log.html");}?>
```

La función del *script* es la siguiente: primero recupera las variables introducidas en el formulario y se conecta a la base de datos *ramo*; después, en función del valor de la variable *'usrcls'*, decide dónde buscar la información del nombre de usuario y claves respectivas. Finalmente, la última línea contempla la posibilidad de que se haya intentado acceder a esta página sin haber pasado por el formulario, requiriéndoló. A partir de este *script*, el árbol de páginas se bifurca en 2 : la "rama del usuario" y la "rama del administrador". Los nombres de las páginas ya fueron nombrados en el último *script*. Se lista a continuación el *script* `acciones.php`, el cual permite al administrador ingresar información de manera intuitiva.
Archivo • `acciones.php`

```
<?php session_start();if ( $_SESSION['estado'] != 'conectado' )
{die( "Ud no esta logueado!.Click aqui para <a href='log.html'>volver</a>");}?>
<html><title> Administrando las notas</title>
<body bgcolor="#000000" text="#66FF00" link="#CC0033" alink="#66FF66">
<font color="red"><u><a href="salir.php"><h4 align="right"> Cerrar sesión.</h4>
<h1 align="center"><font color="blue"> acciones.</h1>
<table><tr><td><?php require('nuser.php');?></td><td>
```

```
<?php require('nnota.php');?>
</td></tr></table></body></html>
```

Las primeras líneas chequean que el usuario se haya "logueado" y especifica el estilo de la página. Luego, se utiliza la función *require* para solicitar los 2 formularios que corresponden a las acciones que pueden ser realizadas por el administrador: **crear** nuevos usuarios y **agregar** notas a cada usuario. Ambos formularios son del tipo caja de texto y la información de éstos es enviada a un archivo del mismo nombre con una "p" final (de procesar). Los archivos que procesan la información solo escriben directamente en la base de datos la información obtenida de los formularios. Finalmente, cabe destacar el *link* "cerrar sesión", el cual conduce a un *script* que cual destruye todas las variables de sesión y devuelve a la página de "logueo" `log.html`, lo que se presenta a continuación:
Archivo • salir.php

```
<?php session_start();if ( $_SESSION['estado'] != 'conectado' )
{die( "Ud no esta logueado!.Click aqui para <a href='log.html'>volver</a>");}
session_unset();header("Location:log.html");echo"<html></html>";exit;?>
```

A continuación se presentan los *scripts* que permiten poner notas, crear nuevos usuarios y sus respectivos archivos de proceso: Archivo • nnota.php.

```
<?php session_start();if ( $_SESSION['estado'] != 'conectado' )
{die( "Ud no esta logueado!.Click aqui para <a href='log.html'>volver</a>");}?>
<html><body bgcolor="#000000" text="#66FF00" link="#CC0033" alink="#66FF66">
<br><br><form method="post" action="nnotap.php"><table align="center">
<tr><td><h2>Nueva nota</h2></td></tr>
<tr><td>Nombre:</td><td><input type="text" name="nombre"></td></tr>
<tr><td>Nota N°:</td><td><input type="text" name="nnota"></td></tr>
<tr><td>La Nota es:</td><td><input type="text" name="nota"></td></tr>
<tr><td></td><td><input type="submit" value="ingresar nota">
</table></form></body></html>
```

El archivo que procesa las variables:
• `nnotap.php`

```
<?php session_start();if ( $_SESSION['estado'] != 'conectado' )
{die( "Ud no esta logueado!.Click aqui para <a href='log.html'>volver</a>");}
$alumno=$_POST['nombre']; $nnota=$_POST['nnota'];$nota=$_POST['nota'];
if($nnota>8){echo "el sistema aguanta máximo 8 evaluaciones";
require("acciones.php");}
else{$conexion=mysql_connect('localhost','lamp','bd')
or die ('No conecta'.mysql_error());
mysql_select_db('ramo');
```

```
$escribe="update notas set nota$nnota='$nota' where nombre='$alumno'";
$haz=mysql_query($escribe);
echo "<b>".$alumno. " saco un ".$nota." en la evaluacion n° ".$nnota."</b>";
require("acciones.php");}?>
```

Debiera quedar completamente claro lo que ejecuta este *script*: recupera las variables y revisa que no se haya sobrepasado la cota máxima de las 8 notas por alumno. De ser así, procede a escribir la nueva información en la base de datos. Cabe notar que la información no esta escrita con la sentencia `INSERT`, sino `UPDATE`, por lo cual las notas pueden ser cambiadas. La segunda acción habilitada por el administrador es la de agregar nuevos usuarios, lo cual se hace de una manera totalmente análoga a los *scripts* anteriores:

Archivo • `nuser.php`

```
<?php session_start();if ( $_SESSION['estado'] != 'conectado' )
{die( "Ud no esta logueado!.Click aqui para <a href='log.html'>volver</a>");}?>
<html><body bgcolor="#000000" text="#66FF00" link="#CC0033" alink="#66FF66">
<br><br>
<form method="post" action="nuserp.php"><table align="left">
<tr><td align="center"> <h2>Nuevo registro</h2><br></td></tr>
<tr><td>Nombre:</td><td><input type="text" name="nombre"></td></tr>
<tr><td>Contraseña:</td><td><input type="password" name="clave"></td></tr>
<tr><td></td><td><input type="submit" value="ingresar usuario"></td></tr>
</table></form></body></html>
```

El archivo que procesa las variables.

Archivo • `nuserp.php`

```
<?php session_start();if ( $_SESSION['estado'] != 'conectado' )
{die( "Ud no esta logueado!.Click aqui para <a href='log.html'>volver</a>");}
$conexion=mysql_connect('localhost','lamp','bd')
or die ('No conecta'.mysql_error());
mysql_select_db('ramo');
$alumno=$_POST['nombre'];
$clave=$_POST['clave'];
$nusuario="insert into alumnos(Nombre,clave)
values ('$alumno',PASSWORD('$clave'))";
$nusuariob="insert into notas (nombre) values ('$alumno')";
$escribe=mysql_query($nusuario)or die('no se pudo escribir');
$escribeb=mysql_query($nusuariob) or die('no se pudo escribir');
echo "<b> alumno ".$alumno. " ingresado con exito</b>";
require("acciones.php");?>
```

Con esto se da por finalizado el proceso de introducir información al sistema. Ahora solo resta generar un *script* que muestre la información correspondiente a cada usuario.

Archivo • `notas.php`

```
<?php session_start(); $nombre=$_POST['nombre']; $clave=$_POST['clave'];
$conexion=mysql_connect('localhost','lamp', 'bd')
or die('No es posible conectar'.mysql_error());
mysql_select_db('ramo')or die ('error al contactar base');
$pedir="select nota1, nota2, nota3, nota4, nota5, nota6, nota7, nota8
from notas where nombre='$nombre'";
$pedido=mysql_query($pedir);
$notas=mysql_fetch_row($pedido);
?>
<html>
<title> Notas: </title>
<body bgcolor="#333300" text="#3300FF" link="33FF33" alink="#669900">
<h2 align="right"> <a href="salir.php" >
<font color="red" <u> Cerrar sesion.</u>
</a> </h2>
<h3 align="left"> <font color="green"> Alumno <?php echo $nombre;?></h3>
<table align="left"><tr><td><b> Sus notas son</b></td></tr>
<tr><td>Eval.N°:</td><td>Nota</td><tr>
<?php
$i=1;
$g=0;
$cont=0;
while($i<=8)
{ $k=$i-1;
echo "<tr><td>". $i . "</td><td>" .$notas[$k]. "</td></tr>";
$i++;
if($notas[$k]>=1){$cont++;}
$g=$g+$notas[$k];
}
if($g==0){echo "usted no tiene notas";}
else{$t=$g/$cont;}?>
<br><br>
<tr><td>Su promedio final es:</td><td><b><?php echo $t; ?> </b>  </td></tr>
</table>
</body></html>
```

Sobre el último ejemplo cabe destacar: se piden todas las notas del alumno en cuestión; la base de datos devuelve para esto la información que es convertida en fila (`mysql_fetch_row()`); luego la fila comienza a ser recorrida utilizando un control de flujo *while*, el cual va generando a tiempo de ejecución una tabla en *html*; las variables auxiliares `$i`, `$g` y `$cont` son utilizadas para calcular el promedio del número de notas que existan.

E.10. Conclusiones.

El ejemplo ha sido chequeado siguiendo los pasos dados en este apéndice y funciona de manera básica (pudiendo ser mejorado),pues su fin no es ser operativo, sino explicativo. Podría parecer que se trata de una gran cantidad de código, sin embargo, la mayor parte del tiempo se está repitiendo lo mismo con variaciones obvias. Esperando haber logrado una comprensión aceptable por parte del lector, A partir de las herramientas entregadas, es posible realizar proyectos bastante más ambiciosos tales como:

- Sistema de votaciones en línea.
- Comunicación mediante mensajes entre usuarios (un rudimentario *chat*).
- Sistema de encuestas en línea.

En general, cualquier problema de administración que involucre diferentes usuarios a distancia se torna en algo solucionable mediante las herramientas aquí desarrolladas. Por último, se deja como ejercicio al lector un par de puntos que restan por mejorar del ejemplo final.

E.10.1. Mejoras al Ejemplo final.

Faltó lograr:

- No se repitan usuarios.
- Solo existan notas entre 1 y 7.
- Comunique al administrador cuando éste intente ingresar notas a un usuario inexistente.

Sobra decir para el lector con algo de experiencia en programación que todas estos problemas serán fácilmente solucionables utilizando el condicional `if`, de manera de cubrir todas las posibilidades. Esto no se ha hecho, pues volvía el ejemplo demasiado redundante y por lo tanto se ha priorizado su funcionalidad.

E.11. Tabla de Colores en *html*.

Figura E.2: Los 256 colores posibles de desplegar en una página en *html*, con su respectivo código.