Universidad de Murcia

Facultad de Informática

Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

TÍTULO DE GRADO EN
CIENCIA E INGENIERÍA DE DATOS

Fundamentos de Computadores

Práctica 7: Sistema de compilación en Linux. Dependencias.

Boletines de prácticas

CURSO 2022 / 23

Índice general

Boletines de prácticas

B7.1. Boletín 1: Sistema de compilación. Paquetes, módulos, dependencias

B7.1.1. Objetivos

Esta sesión está dedicada a la compilación de programas desde sus archivos fuentes en el sistema operativo Linux. En este proceso, la resolución de dependencias e instalación de paquetes, bibliotecas o módulos es crucial. El objetivo es introducir al alumno en los aspectos básicos del sistema de compilación y satisfacción de dependencias: clonado, configuración, compilación e instalación de programas por parte de los usuarios de un sistema Linux.

B7.1.2. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura y seguimiento de los pasos expuestos en el ejemplo del boletín (simplemente replicándolos y observando los resultados).

2. Realización de forma individual de los ejercicios propuestos en el boletín. Estos consistirán en ligeras modificaciones sobre los pasos replicados anteriormente, y el estudio de sus efectos.

B7.1.3. Del código en la nube al proceso en el computador

Una de las mayores fortalezas del sistema operativo Linux es su administrador de paquetes (a través de herramientas como `apt`) y los repositorios de software asociados. Como ya hemos visto en otros boletines de prácticas, esto permite descargar e instalar nuevo software en el computador de forma totalmente automatizada.

No obstante, pueden existir casos en los que el software incluido en los repositorios no encaje con las necesidades de un usuario particular, ya que no es posible empaquetar todo el software disponible en el mundo con todas sus posibles configuraciones. Una razón común por la que es necesario compilar algún programa manualmente es cuando se necesita ejecutar una versión muy específica que no está en el repositorio (por ejemplo, porque queremos utilizar una versión más antigua por alguna cuestión de compatibilidad con otro software). Otra razón, para usuarios más avanzados, es cuando se necesita modificar el código fuente de una aplicación mediante el uso de algunas opciones de compilación sofisticadas.

Por lo tanto, pueden existir situaciones en las que es necesario obtener el código fuente de un programa, compilarlo e instalarlo por uno mismo. En este boletín veremos una guía acerca de este proceso, tomando como ejemplo la compilación del editor `nano` a partir de su código fuente, disponible en un repositorio Git, y su instalación en un subdirectorio del directorio personal del usuario. Finalmente, veremos cómo es posible ejecutar una u otra versión.

A una instalación realizada por un usuario dentro de los confines de su directorio personal (su *home*), se le suele denominar **instalación local**, y en principio –salvo que los permisos establecidos así lo dispongan– ese programa sólo estará accesible para ese usuario. En contraposición, la **instalación del sistema** es aquella realizada bien por el administrador de paquetes (como `apt`), bien por el superusuario (tras compilar un programa desde los ficheros fuentes), pero que en todo caso ubica el programa ejecutable y sus bibliotecas en directorios del sistema que sólo pueden ser modificados por el superusuario (tales como `/usr/bin`), lo cual hace que el programa instalado esté disponible para todos los usuarios. El software instalado por el sistema es almacenado bajo el directorio `/usr`. Cuando el administrador del sistema necesita instalar algún software adicional, existe por convenio un directorio específico para esta instalación: `/usr/local/`. En otros casos también encontraremos software instalado en el directorio `/opt`.

Comprobando la instalación del sistema del editor `nano`

Es posible que nuestro sistema operativo ya tenga una instalación del sistema de la herramienta `nano`. El comando `whereis` nos puede indicar en qué directorio se encuentra el fichero binario ejecutable, y la documentación de un comando dado:

```
$ whereis nano
nano: /usr/bin/nano /usr/share/nano /usr/share/man/man1/nano.1.gz /usr/share/info/nano.info.gz
```

Por otra parte, también podemos comprobar la ubicación de los ficheros de un programa instalado por el sistema a través de la herramienta `dpkg`, que es el gestor de paquetes de los sistemas Linux derivados de la distribución Debian (como es el caso de Ubuntu). En realidad, `apt` es simplemente una herramienta que proporciona una interfaz de línea de comandos de más alto nivel para acceder al gestor de paquetes subyacente, ofreciendo un uso más simple e intituivo, al tiempo que utiliza *por debajo* `dpkg`. Si ejecutamos el comando `sudo dpkg -L nano`, podemos comprobar la ubicación de los ficheros instalados por el programa nano:

```
$ sudo dpkg -L nano
/.
/bin
/bin/nano
/etc
/etc/nanorc
/usr
/usr/share
/usr/share/doc
/usr/share/doc/nano
/usr/share/doc/nano/AUTHORS
[...]
```

Como puedes ver, los ficheros instalados están en unos pocos directorios: `/bin` (programas ejecutables), `/etc` (ficheros de configuración) y `/usr/share` (ficheros de texto que son independientes del hardware de la máquina local, tales como documentación, diccionarios y bibliotecas).

B7.1.4. El sistema de compilación en Linux

En el mundo de Unix, el sistema de compilación (*build system*) de nuevo software se basa en el comando `make`. Este programa toma como entrada un fichero `Makefile`, que contiene la "receta" para construir el paquete de software, y genera como salida los ficheros que componen el paquete ya compilado (ficheros ejecutables, etc.).

Cuando un paquete necesita ser compilado en una plataforma diferente a la que fue desarrollado, esto normalmente implica modificar el `Makefile` asociado para su compilación. Por ejemplo, el compilador puede tener otro nombre o requerir más opciones. Sin embargo, puesto que no resulta práctico tener que personalizar el `Makefile` para cada una de las diferentes plataformas para las que se pueda compilar un determinado paquete, el software se suele distribuir con un script de configuración que se encarga de generar el `Makefile` de acuerdo con la configuración del sistema detectada por el propio script. En su escenario más simple, para compilar un paquete bastaría con ejecutar `./configure && make && make install`. Veremos a continuación estos tres pasos con más detenimiento.

Pasos en la instalación de software a partir del código fuente en Linux

1. **Configurar el programa.**

   Junto con el código fuente del programa, se suele distribuir un *shell script* de configuración, llamado por convenio `configure`, el cual es responsable de preparar la construcción del software en su sistema específico. Se asegura de que todas las dependencias requeridas (compilador, bibliotecas, etc.) para el resto del proceso de compilación e instalación estén disponibles en el sistema en su versión adecuada, y averigua dónde se encuentran en el sistema de archivos. Normalmente, las distribuciones de programas en código fuente no incluyen un `Makefile` terminado, sino una plantilla llamada `Makefile.in` o `Makefile.am`. Es el script de configuración `configure` el que produce un `Makefile` adaptado al sistema donde se realiza la compilación. Así pues, el primer paso en un proceso general de compilación es ejecutar `./configure` en el directorio raíz de la distribución de código, lo cual generará el fichero `Makefile` necesario para compilar el programa.

2. **Compilar el programa.**

   Una vez que `configure` haya hecho su trabajo, podemos ejecutar `make` para compilar el software. Este comando lee el fichero `Makefile`, el cual contiene un conjunto de reglas que indican cómo construir los ficheros del paquete. Por ejemplo, un `Makefile` puede indicar que el programa `prog` puede ser construido ejecutando el programa enlazador sobre los archivos `main.o`, `foo.o`, y `bar.o`; el archivo `main.o` puede ser construido ejecutando el compilador en `main.c`; etc. Cada vez que se ejecuta `make`, éste lee el `Makefile`, comprueba la existencia y la fecha de modificación de los archivos mencionados, decide qué archivos necesitan ser compilados (o recompilados), y ejecuta los comandos asociados.

3. **Instalar el software.**

   Cuando el programa está compilado y listo para ejecutarse, los archivos de éste se pueden copiar a sus destinos finales, ya se trate de una instalación local (sólo para el usuario que lo compila) o del sistema (global). El comando `make install` copiará el programa compilado, sus bibliotecas y documentación, en las ubicaciones correctas. El paso de instalación también se define en el archivo `Makefile`, por lo que el lugar donde se instala el software se puede generalmente cambiar según las opciones pasadas al script de configuración. Por ejemplo, se puede especificar la ruta donde se procederá a la instalación mediante la opción `--prefix=RUTA`. Dependiendo de dónde se instale el software, es posible que se necesiten permisos de superusuario para que la instalación pueda copiar archivos en los directorios del sistema (para esto tenemos el comando `sudo`).

Autotools

Como usuarios finales del software, lo que percibimos es que la configuración, compilación e instalación de un paquete funcionan porque el script `configure` examina nuestro sistema, y utiliza la información que encuentra para convertir un fichero a modo de plantilla llamado `Makefile.in` en un `Makefile`, pero ¿de dónde vienen el script `configure` y la plantilla `Makefile.in`? Estos dos ficheros suelen tener miles de líneas, tantas que a veces son incluso más largos que el propio código fuente del programa que instalan. Incluso partiendo de un script `configure` existente, sería muy desalentador construirlo manualmente. La realidad es que estos scripts no se construyen a mano.

Puesto que estos scripts son largos y complejos de generar, se suelen utilizar un conjunto de herramientas conocidas como *Autotools*. Este conjunto de herramientas se encarga de, a partir de plantillas más sencillas, analizar el sistema donde se va a hacer la compilación e instalación, resolver las dependencias del software y generar los ficheros que actúan como entrada a los comandos `configure` y `make`. La Figura I.1 muestra un esquema general de este procedimiento. Los programas `autoconf` y `automake` generan, respectivamente, el script `configure` y el archivo plantilla `Makefile.in`.

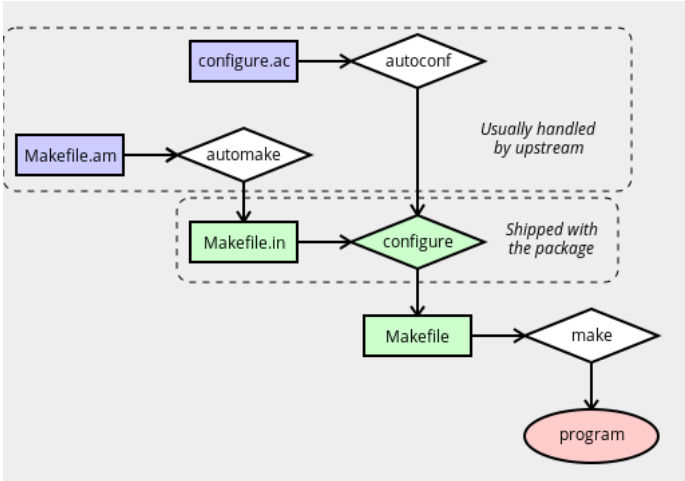


Figura I.1: Diagrama de componentes de *autotools*

B7.1.5. Caso práctico de ejemplo: Instalación local del programa `nano`

Ahora vamos ilustrar con un programa concreto, el editor de texto **nano**, cómo se lleva a cabo el proceso de compilación e instalación de un programa en nuestro sistema Linux, partiendo de su código fuente disponible en un repositorio en la nube.

1. **Clonación del repositorio Git donde reside el código fuente.**

   Vamos empezar por clonar en algún lugar de nuestro directorio personal el repositorio del editor `nano`, que es parte del conjunto de software libre GNU. En nuestro caso, crearemos en un subdirectorio `build` expresamente para este proceso. Dentro de este directorio clonaremos el repositorio y procederemos a su compilación e instalación. Este repositorio se ubica en la siguiente URL: `git://git.savannah.gnu.org/nano.git`. Por tanto, ejecutamos el siguiente comando:

   ```
   $ cd && mkdir build && cd build
   ~/build$ git clone git://git.savannah.gnu.org/nano.git
   Clonando en 'nano'...
   [...]
   Resolviendo deltas: 100% (48727/48727), listo.
   ```

   Como indica la salida de `git`, el directorio que se ha creado se llama **nano**. Si examinamos dicho directorio, lo primero que tenemos que hacer es identificar los componentes principales de la carpeta de la distribución de un programa en código fuente.

   ```
   ~/build$ cd nano
   ~/build/nano$ ls
   AUTHORS              ChangeLog.2007-2015  doc           nano-regress  README.hacking     THANKS
   autogen.sh           configure.ac         IMPROVEMENTS  NEWS          roll-a-release.sh  TODO
   ChangeLog            COPYING              m4            po            src
   ChangeLog.1999-2006  COPYING.DOC          Makefile.am   README        syntax
   ```

   El código fuente del programa se encuentra en la carpeta `src` y su documentación en la carpeta `doc`. Podemos ver que hay dos archivos `README` en el directorio, `README` y `README.hacking`. Un archivo `README` debe contener una guía sobre cómo compilar el código y sus aspectos más importantes:

   - Lenguaje: en qué lenguaje de programación está escrito el código.
   - Dependencias: qué otro software necesita tener instalado en el sistema para que la aplicación se compile y ejecute.
   - Instrucciones: Los pasos literales que se deben seguir para compilar el software.

   Por tanto, antes de compilar e instalar manualmente un programa en código fuente, el primer paso que tenemos que hacer es leer cualquier fichero README de la carpeta de este código.

   Si visualizamos estos dos archivos (con el comando `less`), veremos cuál es diferencia entre ellos: cada uno se corresponde a un tipo de instalación diferente.

2. **Comprobación e instalación de las dependencias requeridas**.

   En nuestro caso, hemos optado por la instalación basada en el repositorio Git, así que seguiremos las instrucciones del archivo `README.hacking`. La primera parte de este archivo nos indica las dependencias del programa `nano`, es decir, el software del que depende y por tanto debe estar previamente instalado en nuestro sistema operativo con la versión correcta:

   ```
   autoconf     (version >= 2.69)
   automake     (version >= 1.14)
   autopoint    (version >= 0.18.3)
   gcc          (version >= 5.0)
   [...]
   You will also need to have the header files for ncurses installed,
from libncurses-dev on Debian, ncurses-devel on Fedora, or similar.
   ```

Para cada una de las dependencias mostradas en el fichero `README.hacking`, ejecuta el comando `apt list` para comprobar si está instalado el paquete (en este caso, la salida del comando mostraría *[instalado]*, indicando la versión del paquete). En el caso de que no estuviera instalado o tuviéramos una versión demasiado antigua, ejecutarías el comando `apt install` o `apt upgrade` para instalarlo o actualizarlo, respectivamente. Mostramos como ejemplo la instalación del primer paquete de la lista, `autoconf`:

```
~/build/nano$ apt list autoconf
Listando... Hecho
~/build/nano$ sudo apt install autoconf
[...]
Se instalarán los siguientes paquetes adicionales:
  automake autotools-dev libsigsegv2 m4
[...]
~/build/nano$ apt list autoconf
Listando... Hecho
autoconf/jammy,jammy,now 2.71-2 all [instalado]
```

Después de instalarlo, comprobamos que la versión que se ha instalado (2.71) es mayor que la requerida (2.69), por lo que esta dependencia se ha resuelto. Seguimos con la instalación de la siguiente dependencia, `automake`. Si observamos la salida de la instalación de `autoconf`, veremos que `automake` es una dependencia de `autoconf`, y se ha instalado dentro del proceso de instalación de este paquete. Es importante comprobar las dependencias en orden y tal como se muestran en tu fichero `README`. Podemos comprobar que la versión instalada de `automake` es correcta:

```
~/build/nano$ apt list automake
Listando... Hecho
automake/jammy,jammy,now 1:1.16.5-1.3 all [instalado, automático]
~/build/nano$ apt list libncurses-dev
Listando... Hecho
libncurses-dev/jammy 6.3-2 amd64
libncurses-dev/jammy 6.3-2 i386
~/build/nano$ sudo apt install libncurses-dev
```

Como vemos, esta version es la 1.16.5 y es mayor que la requerida (1.14). Repite los pasos anteriores para el resto de dependencias, instalando o actualizando los paquetes que sean necesarios.[1]. Por último, no olvides que también se necesitan los ficheros de cabecera (*headers*) de la librería `ncurses`, por lo que en el caso de Ubuntu habría que instalar el paquete `libncurses-dev`, tal como muestra el listado anterior.

3. **Generación de los ficheros `configure` y `Makefile.in` mediante *autotools*.**

   El script `autogen` utiliza las herramientas *autotools* para generar, a partir de los ficheros plantilla `configure.ac` y `Makefile.am`, el script de configuración `configure` y la plantilla de instalación `Makefile.in`. Ahora seremos capaces de generar el archivo `Makefile` final que será usado para la instalación del programa.

```
~/build/nano$ ./autogen.sh
[...]
Autoconf...
Autoheader...
Automake...
Done.
~/build/nano$ ls -t
syntax          install-sh         README.hacking
src             missing            THANKS
m4              config.h.in        TODO
lib             po                 ChangeLog.2007-2015
doc             config.rpath       IMPROVEMENTS
Makefile.in     ABOUT-NLS          Makefile.am
configure       gnulib             ChangeLog.1999-2006
autom4te.cache  roll-a-release.sh  ChangeLog
[...]
```

[1] Si estas usando la máquina virtual proporcionada para este laboratorio, tendrás que instalar autopoint, gettext, groff, pkg-config y texinfo.

4. **Configuración y generación del archivo de compilación** `Makefile`.

   En este punto, ejecutamos el script de configuración que acabamos de generar en el paso anterior, indicando al script mediante la opción `--prefix=RUTA` el directorio en el que queremos instalar el programa. En este caso, vamos a configurarlo para una instalación local, dentro de nuestro directorio personal (indicado por la variable de entorno `$HOME`).

```
~/build/nano$ ./configure --prefix=$HOME/local/nano
checking build system type... x86_64-pc-linux-gnu
[...]
config.status: creating Makefile
[...]
```

   Como podemos ver, se ha generado el archivo `Makefile`.

5. **Compilación e instalación del programa**.

   Ya podemos compilar e instalar a través del comando `make`, que tiene como entrada el fichero `Makefile` con todas las reglas necesarias para la compilación del programa `nano`.

```
~/build/nano$ make
[...]
gcc  -g -O2 -Wall   -o nano browser.o chars.o color.o cut.o files.o  global.o help.o history.o move.o
nano.o prompt.o rcfile.o search.o  text.o utils.o winio.o ../lib/libgnu.a   -lz  -Wl,-Bsymbolic-functions
-lncursesw -ltinfo
[...]
```

   Tras ejecutar `make`, podemos ver que, entre otras cosas, en el subdirectorio `src` han aparecido los ficheros de código objeto (`.o`) resultantes de compilar cada uno de los ficheros de código fuente (`.c`), así como el fichero ejecutable que contiene el programa `nano`. Aunque ya podríamos ejecutar dicho programa en esta ubicación, lo habitual es que el programa ejecutable se copie junto con su documentación y bibliotecas, a otra ubicación en el sistema de ficheros distinta al directorio que contiene el código fuente. Esto es precisamente la tarea acometida en el siguiente paso con `make install`.

```
~/build/nano$ make install
[...]
~/build/nano$ ls ~/local/nano
bin  share
~/build/nano$ ls ~/local/nano/bin
nano  rnano
```

6. **Ejecuta tu programa nano.**

   ¿Que sucede si ejecutamos ahora el comando `nano`? ¿Cuál de los dos programas se ejecuta, el de la instalación del sistema o el que acabamos de instalar localmente? En este punto, la instalación local y la instalación del sistema del programa `nano` conviven en la misma máquina. Para saber qué programa ha iniciado el comando `nano`, tenemos que recordar que la variable de entorno `$PATH` es la que dictamina en qué directorios busca el shell los programas que podemos ejecutar sin necesidad de proporcionar la ruta exacta al fichero ejecutable de dicho programa. Por tanto, como lo más probable es que el directorio donde hemos instaldo `nano` (`$HOME/local/nano`) no esté en el *path*, estaremos aún ejecutando la versión del programa instalada globalmente en el sistema por el administrador de paquetes.

   Para ejecutar la instalación local que acabamos de realizar, tenemos que indicar la ruta completa al fichero ejecutable, o bien modificar el *path* si queremos dar prioridad a las instalaciones locales. Si asignamos un nuevo valor a la variable de entorno `PATH`, anteponiendo a la lista existente el directorio `$HOME/local/nano/bin`, el shell lo encontrará ahí antes de que llegue a buscar en `/usr/bin`, que es donde está la instalación del sistema[2]

[2] Ten en cuenta que este cambio en el *path* es temporal (sólo afecta al shell actual); para hacerlo permanente, debes modificar el fichero `.profile` o bien `.bash_profile`.

```
~/build/nano$ nano --version
 GNU nano, versión 6.2
[...]
~/build/nano$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:[...]
~/build/nano$ whereis nano
nano: /usr/bin/nano /usr/share/nano /usr/share/man/man1/nano.1.gz [...]
~/build/nano$ ~/local/nano/bin/nano --version
 GNU nano from git, v6.3-5-gec0c13af
[...]
~/build/nano$ PATH=$HOME/local/nano/bin:$PATH
~/build/nano$ nano --version
 GNU nano from git, v6.3-5-gec0c13af
[...]
```

B7.1.6. Ejercicios a realizar durante la sesión

NOTA IMPORTANTE:
Antes de realizar los ejercicios, asegúrate de grabar la sesión con `script -a typescript_prac7_bol1`.
Debes empezar cada ejercicio escribiendo el comentario de documentación `### EJERCICIO N ###` . Finalmente, no olvides añadir dicho fichero a tu repositorio-bitácora, respetando la organización de la misma en directorios y subdirectorios.
Recuerda responder a cada pregunta tecleando tu respuesta en el mismo terminal (a continuación de los comandos necesarios para cada apartado) precedida por el carácter #, que indica al *shell* que se trata de un comentario (el shell ignora los siguientes caracteres hasta el siguiente final de línea, cuando pulses INTRO). Puedes introducir tu respuesta en mútiples líneas si así lo deseas, empezando cada línea con #.

1. Lista los directorios de instalación del comando “gcc” y el propósito de cada uno de ellos. ¿Que comando has utilizado?

2. Descarga, compila e instala localmente, en un subdirectorio de `$HOME/local`, el programa `git`, partiendo de su repositorio de código fuente situado en esta URL: `https://github.com/git/git.git`

3. Ejecuta la versión de `git` que has instalado localmente, y compárala con la versión instalada en el sistema.

4. Descarga, compila e instala localmente, en un subdirectorio de `$HOME/local`, el programa `htop`, partiendo de su repositorio de código fuente situado en esta URL: `https://github.com/htop-dev/htop.git`

5. Ejecuta la versión de `htop` que has instalado localmente, y compárala con la versión instalada en el sistema.

B7.2. Boletín 2: Gestión de paquetes y entornos virtuales en Python

B7.2.1. Objetivos

Esta sesión está dedicada a las diferentes alternativas de ejecución de programas Python, la resolución de dependencias e instalación de paquetes Python, así como la utilización de los entornos virtuales.

B7.2.2. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura y seguimiento de los pasos expuestos en el ejemplo del boletín (simplemente replicándolos y observando los resultados).

2. Realización de forma individual de los ejercicios propuestos en el boletín. Estos consistirán en ligeras modificaciones sobre los pasos replicados anteriormente, y el estudio de sus efectos.

B7.2.3. Ejecución de programas Python

En este apartado vamos a repasar cómo podemos ejecutar programas escritos en Python de forma local en nuestra máquina. El intérprete de Python suele venir ya instalado en cualquier distribución de Linux y podemos optar por trabajar localmente por línea de comandos. No obstante, existen muchas plataformas *cloud* que nos proporcionan un intérprete online sin necesidad de instalar ningún software o crear ficheros en nuestra máquina. Podemos también optar por la interfaz de "navegador Web" pero sobre nuestra instalación local. A continuación comentamos estas diferentes opciones de entornos de ejecución de Python.

El intérprete de Python por línea de comandos

Hay dos maneras de usar el intérprete de Python: el **modo de consola** y el **modo de programa**, guion o script.

En el *modo de shell* o consola, podemos interactuar con el intérprete de Python línea a línea. El intérprete utiliza tres signos 'mayor' para indicar que está listo para recibir una sentencia (*statement*, instrucción que el intérprete de Python puede ejecutar). Como vemos a continuación, escribimos la sentencia `print(``Hola mundo!!'')` y el intérprete procesa esta sentencia e imprime la salida por consola. En la siguiente línea da un nuevo aviso de que está listo para más sentencias:

```
$ python3
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>  print("Hola mundo")
Hola mundo
>>>
```

La consola de Python es muy útil ya que podemos probar pequeños fragmentos de código de forma interactiva, comprobar de forma rápida la sintaxis de una sentencia, módulos específicos o saber de forma inmediata la salida que dará cierta ejecución de sentencias. Esto nos permite probar código antes de copiarlo al archivo de código fuente. En ocasiones, esto puede ser más conveniente que trabajar directamente con el archivo de código fuente, ya que para comprobar el correcto funcionamiento de cada sentencia, tenemos que ejecutarlo en modo "depuración" (paso a paso, de sentencia en sentencia), y por cada cambio de código, modificar el archivo y volverlo a ejecutar.

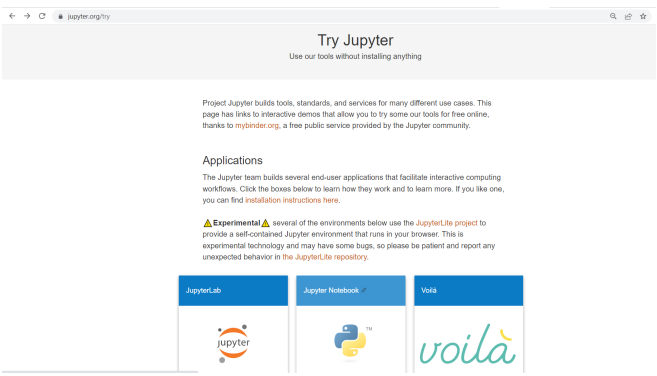
Cuando trabajamos en *modo programa*, escribimos nuestro código en un archivo (que por lo general tendrá extensión '.py') que posteriormente pasamos como parámetro al intérprete de Python de nuestro sistema operativo para que lo ejecute:

```
$ echo "print("Hola mundo")" > hello.py
$ python3 hello.py
Hola mundo
```

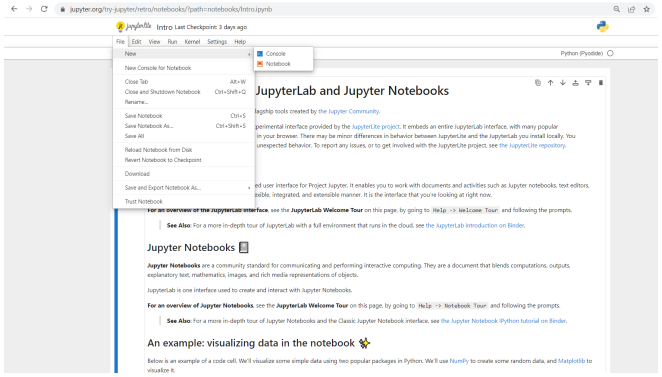
El intérprete de Python desde la Web

Los intérpretes interactivos desde la Web nos dan la facilidad de poder ejecutar código Python y testear sin la necesidad de preocuparnos de instalarlos en la máquina. Podemos acceder a estos intérpretes por medio de nuestro navegador web y empezar a programar directamente. Los intérpretes Web más conocidos son *Google Colab*, *Replit*, *Python Tutor* y *Jupyter*.

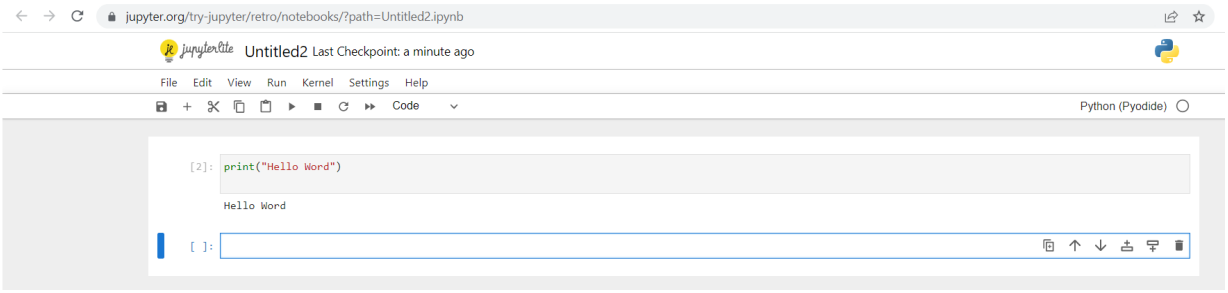
Los *Jupyter Notebooks* son una aplicación web, de código abierto, que nos va a permitir crear y compartir documentos con código en vivo, ecuaciones, visualizaciones y texto explicativo. Además, podemos trabajar de forma local con ellos si lo deseamos, lo cual ofrece una gran flexibilidad. Si elegimos la forma remota, sólo tenemos que acceder a su página Web `https://jupyter.org`, seleccionar "Try" y luego "Jupyter Notebooks", tal y como se muestra en la Figura I.2. Una vez estemos en Jupyter Notebooks, seleccionar en la barra de herramientas de arriba *File->New->Notebook*.



(a) Selección de Jupyter Notebook



(b) Creación de notebook



(c) Ejemplo de notebook

Figura I.2: Web de Jupyter Notebook

B7.2.4. Gestión de paquetes en Python

Una de las formas más habituales de obtener bibliotecas de Python es a través de su gestor de paquetes **pip** (*package installer for Python*). Para instalarlo en nuestro sistema, basta con ejecutar el comando:
`sudo apt install python3-pip`.
Gracias a `pip`, tendremos acceso al repositorio oficial (y más grande) de paquetes de Python, el llamado *Python Package Index* (PyPI). Cualquier programador puede registrarse gratuitamente en PyPI y subir ahí sus propios paquetes. En la web de PyPI podemos realizar búsquedas de paquetes por nombre y filtrar resultados en función de ciertos criterios (como el estado de desarrollo, la licencia, etc). Además, cada paquete tiene su propia página en la web de PyPI, donde podemos obtener información como el comando para instalarlo, el nombre del autor, las versiones de Python con las que se ha testeado el paquete, y mucho más.

Para instalar con `pip` cualquier paquete disponible en PyPI, simplemente ejecutamos el comando `pip install` seguido del nombre del paquete. La herramienta `pip` no sólo instalará el paquete que le indicamos y su dependencias, sino que además los almacenará en una caché local de caras a futuras instalaciones.

Otro comando útil, complementario al de instalar, es el que nos muestra un listado de los paquetes instalados, `pip list`, junto con su versión. Una vez hemos instalado un paquete en nuestro entorno de Python, podemos obtener información adicional sobre el mismo en nuestro terminal. Para ello tenemos que ejecutar el comando `pip show` seguido del nombre del paquete. Es importante remarcar que el comando `pip` no mostrará información sobre los paquetes propios o "built-in" del intérprete que están instalados por defecto, tales como *time* o *math*.

Una consideración a tener en cuenta es que `pip` instala por defecto los paquetes en el entorno global de Python, de forma que los paquetes instalados por un usuario se ubican en un subdirectorio de su directorio personal, por lo general, en `$HOME/.local/lib/pythonX.X/site-packages`. Sin embargo, se recomienda instalar nuestras dependencias para un proyecto de software determinado en un *entorno virtual de Python* expresamente creado para dicho proyecto, ya que de este modo mantenemos las dependencias separadas por proyectos y evitamos posibles conflictos entre diferentes versiones de una misma biblioteca que puedan ser requeridas por dos proyectos distintos.

B7.2.5. Entornos virtuales en Python

Python dispone de una herramienta muy útil para el desarrollo de aplicaciones de forma aislada: los *entornos virtuales*. Estos entornos funcionan como directorios de instalación aislados de otros entornos virtuales y del sistema operativo. Cada entorno virtual tiene su propia copia del intérprete de Python, incluyendo copias de sus utilidades de soporte. Los paquetes o bibliotecas instalados en un entorno virtual no son visibles en otros entornos. Este aislamiento permite localizar las dependencias de un proyecto sin tener que instalarlas en todo el sistema. Esto tiene la ventaja evidente de poder tener varios entornos, con varios conjuntos de paquetes o incluso diferentes versiones del intérprete de Python, sin conflictos entre ellos.

Python tiene varias herramientas nativas para la gestión de entornos virtuales que hacen que todo el proceso sea bastante sencillo. En nuestro caso, nos centraremos en el módulo **venv**, ya que es la forma preferida de crear y gestionar entornos virtuales, y está incluida en la biblioteca estándar de Python. En Ubuntu, para poder utilizar esta herramienta necesitamos tener instalado el paquete `python3.10-venv`. Los comandos básicos que ofrece la funcionalidad **venv** de Python3:

- `python3 -m venv [miruta]`. Este comando crea un entorno virtual en un directorio determinado por la ruta especificada.
- `source [miruta]/bin/activate`. Antes de poder utilizar un entorno virtual, es necesario activarlo explícitamente. La activación hace que el entorno virtual sea el intérprete de Python local por defecto mientras dure la sesión por consola.
- `deactivate`. Para terminar la sesión con el entorno virtual.

A continuación, veremos ejemplos de comandos para crear y activar un entorno virtual, y cómo podemos instalar una versión específica de un determinado paquete de Python, diferente de la versión que pueda existir en la instalación del sistema. En primer lugar, empezamos creando el entorno virtual y mostrando el contenido del directorio creado. Dentro de dicho directorio, se crearán varios subdirectorios, entre los cuales `bin` contendrá la copia del intérprete de Python junto con sus utilidades, así como el *script* `activate` que utilizamos seguidamente para activar el entorno virtual. Como vemos, al activarlo se modifica el *prompt* para que sepamos en todo momento si estamos fuera o dentro de un entorno virtual, y de cuál se trata.

```
$ python3 -m venv mypythonproject
$ ls mypythonproject/
bin  include  lib  lib64  pyvenv.cfg
$ ls mypythonproject/bin/
activate  activate.csh  activate.fish  Activate.ps1  pip  pip3  pip3.10  python  python3  python3.10
$ source mypythonproject/bin/activate
(mypythonproject) $
```

Una vez activado el entorno virtual, podemos ver los paquetes disponibles con el comando `pip list` y comprobar que únicamente tenemos los paquetes `pip` y `setuptools` (que están disponibles por defecto al crear un

entorno virtual. A continuación, procedemos a instalar el paquete *NumPy* (uno de los paquetes fundamentales para computación científica con Python) mediante el comando `pip3 install numpy`.

```
(mypythonproject) $ pip list
Package    Version
---------- -------
pip        22.0.2
setuptools 59.6.0
(mypythonproject) $ pip3 show numpy
WARNING: Package(s) not found: numpy
(mypythonproject) $ pip3 install numpy
Collecting numpy
  Downloading numpy-1.23.4-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (17.1 MB)
[...]
Successfully installed numpy-1.23.4
(mypythonproject) $ pip3 show numpy
Name: numpy
Version: 1.23.4
Summary: NumPy is the fundamental package for array computing with Python.
[...]
Location: /home/alumno/mypythonproject/lib/python3.10/site-packages
[...]
```

Vemos que la instalación obtiene la versión 1.23.4, que se instala dentro del directorio donde hemos creado nuestro entorno virtual (subdirectorio `lib/python3.10/site-packages`). Por último, si desactivamos el entorno virtual, podemos comprobar si el paquete *numpy* se encuentra instalado en el directorio del usuario o bien en el sistema, y de qué versión se trata. Los paquetes de Python instalados en el sistema se ubican por convenio en el directorio `/usr/lib/python3/dist-packages`, mientras que la instalación de usuario utiliza por defecto el subdirectorio `.local/lib/python3.10/site-packages` del directorio personal del usuario:

```
(mypythonproject) $ deactivate
$ pip3 show numpy
Name: numpy
Version: 1.21.5
Summary: NumPy is the fundamental package for array computing with Python.
[...]
Location: /usr/lib/python3/dist-packages
[...]
$ pip3 install numpy==1.22
Defaulting to user installation because normal site-packages is not writeable
Collecting numpy==1.22
Successfully installed numpy-1.22.0
$ pip3 show numpy
Name: numpy
Version: 1.22.0
[...]
Location: /home/alumno/.local/lib/python3.10/site-packages
```

Jupyter Notebooks con entornos virtuales en local

Jupyter Notebooks ofrece una gran flexibilidad que incluye poder ejectuar *notebooks* en local desde nuestro navegador. Para poder utilizarlo en este modo, vamos a instalarlo, mediante la herramienta `pip`, y vamos a usar el entorno virtual descrito anteriormente para instalar y configurar Jupyter de forma aislada. Tras activar nuestro entorno virtual con `source ~/mypythonproject/bin/activate`, procedemos a instalar Jupyter y posteriormente lanzarlo en local mediante el comando `jupyter notebook`:

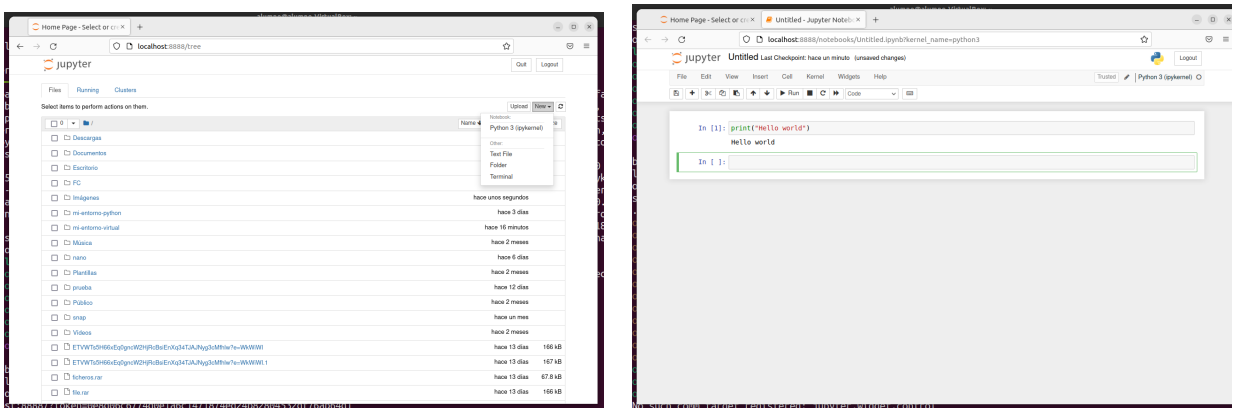
```
(mypythonproject) $ pip install jupyter
[...]
...
Successfully installed Send2Trash-1.8.0 [...] jupyter-1.0.0 jupyter-client-7.3.5 [...]

(mypythonproject) $ jupyter notebook
[I 14:10:32.308 NotebookApp] Writing notebook server cookie secret to ....
```

```
To access the notebook, open this file in a browser:
    file:///home/alumno/.local/share/jupyter/runtime/nbserver-125586-open.html
Or copy and paste one of these URLs:
    http://localhost:8888/?token=0e8d06c6774d0e1a6c1471874ed24b82804532df76ab64d1
 or http://127.0.0.1:8888/?token=0e8d06c6774d0e1a6c1471874ed24b82804532df76ab64d1
```

Se mostrará un registro de las actividades de Jupyter Notebook en el terminal. Cuando se ejecuta Jupyter Notebook, este funciona en un número de puerto específico. Normalmente, el primer notebook que ejecutas usará el puerto 8888 como muestra la salida del comando.

Para acceder al notebook, copiamos una de las tres últimas URLs indicadas por la salida del comando en el navegador de nuestro sistema operativo Ubuntu. Como se muestra en la Figura I.3, Jupyter Notebook mostrará todos los archivos y las carpetas en el directorio desde el que se ejecuta. Para crear un nuevo archivo de Notebook, seleccione "New->Python 3" en el menú desplegable que se encuentra en la parte superior derecha. Con esto se abrirá un Notebook. Ahora podemos escribir código de Python en la celdas y ejecutarlo pulsando el botón de "Run" en la barra de herramientas superior.



(a) Inicio de Jupyter Notebook en local (b) Un notebook en local

Figura I.3: Jupyter Notebook instalado en local

Jupyter Notebook es una herramienta muy poderosa que dispone de muchas características. Estas quedan fuera del alcance de la asignatura, pero os animamos a que las investiguéis por cuenta propia, si no lo habéis hecho ya en otras asignaturas.

B7.2.6. Ejercicios a realizar durante la sesión

NOTA IMPORTANTE:
Antes de realizar los ejercicios, asegúrate de grabar la sesión con `script -a typescript_prac7_bol2`.
Debes empezar cada ejercicio escribiendo el comentario de documentación `### EJERCICIO N ###` . Finalmente, no olvides añadir dicho fichero a tu repositorio-bitácora, respetando la organización de la misma en directorios y subdirectorios.
Recuerda responder a cada pregunta tecleando tu respuesta en el mismo terminal (a continuación de los comandos necesarios para cada apartado) precedida por el carácter #, que indica al *shell* que se trata de un comentario (el shell ignora los siguientes caracteres hasta el siguiente final de línea, cuando pulses INTRO). Puedes introducir tu respuesta en mútiples líneas si así lo deseas, empezando cada línea con #.

1. **Gestión de paquetes en Python.**

   a) Utilizando `pip3`, lista los paquetes disponibles en el entorno global de Python. A continuación, repite el listado pero añade la opción `-v` para obtener más detalles sobre la ubicación de cada paquete instalado.

b) Repite el listado *verboso* anterior de paquetes instalados, pero esta vez combínalo con `grep` para mostrar únicamente los paquetes que están disponibles para todos los usuarios del sistema.

c) Repite el listado *verboso* anterior, pero esta vez combínalo con `grep` para mostrar únicamente los paquetes que están instalados en el entorno global del usuario `alumno`, y por tanto, disponibles sólo para dicho usuario.

d) Muestra la información sobre el paquete `requests` e indica si dicho paquete está disponible para cualquier usuario del sistema.

e) Comprueba si el paquete `pandas` está instalado, y procede a instalarlo con `pip3` si todavía no lo está. Después, muestra la información sobre dicho paquete. Finalmente, utiliza el filtro `grep` para mostrar únicamente los paquetes de los que depende `pandas`.

f) A partir de la salida del último comando del apartado anterior, aplica adicionalmente los filtros `cut` (usando el delimitador apropiado) y `tr` para que el resultado mostrado por pantalla sea únicamente los nombres de los paquetes requeridos por `pandas` (sin ningún separador).

g) Sabiendo que `pip3 show` admite como parámetro uno o más nombres de paquetes, utiliza la salida del comando anterior (colocándolo dentro de un *subshell*, consulta la práctica 5 para más detalles) para escribir un comando que muestre la información sobre todos los paquetes de los que depende el paquete `pandas`. Comprueba que efectivamente dichos paquetes son requeridos por `pandas`.

h) Comprueba mediante `apt list` si el paquete `python3-tk` está instalado en el sistema. En caso de no estarlo, instálalo con `sudo apt install python3-tk`.

i) Después, ejecuta desde la línea de comandos (en modo *programa*) el programa `sample_plot.py` (disponible en el Aula Virtual). ¿Puedes ejecutarlo correctamente? En caso negativo, explica por qué y ejecuta los comandos necesarios para que dicho programa funcione.

j) Ejecuta de nuevo el código anterior en modo programa para comprobar que funciona. Guarda el gráfico generado en formato *png* y añade dicha fichero de imagen a tu repositorio-bitácora.

2. **Entornos virtuales en Python.**

a) Crea un entorno virtual llamado *myvirtenv* y actívalo. Ejecuta la consola de Python en el entorno virtual e importa el paquete `seaborn`. ¿Que ocurre? Realiza los pasos oportunos para que el programa `sample_plot.py` funcione dentro del entorno virtual que has creado.

b) Utiliza `pip` para instalar el programa `glances` dentro del entorno virtual. Después, muestra la información sobre dicho paquete, filtrando de nuevo la salida para mostrar únicamente la ubicación en la que ha sido instalado.

c) Ejecuta `glances --version` (**NOTA IMPORTANTE:** Esta forma de invocar al programa `glances` hace que únicamente se muestre su versión y no se borre la salida del terminal hasta el momento, por lo que no afecta al *log* generado con el programa `script`.).

d) Desactiva el entorno virtual, y repite el comando anterior para comprobar si `glances` está instalado en el sistema. Si no lo está, utiliza `apt` para instalarlo y finalmente compara las dos versiones (la del entorno virtual creado y la instalada en el sistema), tanto con `glances --version` como mediante `pip3 show`.

3. **Jupyter notebooks.**

a) Instala Jupyter dentro del entorno virtual creado en el ejercicio anterior, y ejecuta `jupyter-notebook`. Copia el código del programa `sample_plot.py` y ejecútalo en un *notebook*.

b) En una nueva celda, vuelve a pegar el mismo código salvo las importaciones, y modifica los valores que se representan en la gráfica. Ejecútalo y comprueba el cambio en la representación.

c) Guarda el *notebook*, sal de Jupyter, y finalmente añade el fichero generado a tu repositorio-bitácora.