Entornos y contenedores

Autores: Guerau Fernandez Isern, Joan Colomer Vila, Maria Begoña Hernández Olasagarre, Enrique Blanco García
El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Josep Jorba Esteve
PID_00298303
Primera edición: septiembre 2023

Introducción

A medida que la cantidad de datos en el área de ciencias de la vida y en otras áreas del conocimiento aumenta exponencialmente, surge la necesidad de estructurar y manejar sistemáticamente la información adquirida. Que los datos que manejamos sigan los principios FAIR (*) es fundamental para la correcta manipulación de los datos y de su trazabilidad a lo largo del ciclo de vida del dato. Además, hay dos procesos fundamentales, los cuales estudiaremos en los próximos temas, que son la escalabilidad y la reproducibilidad.

Tradicionalmente, a fin de minimizar el número de pasos manuales que se llevan a cabo en un análisis, los procesos se han unido programáticamente en los denominados *pipelines*. Esta concatenación de procesos normalmente se estructura en archivos (*scripts*). Este tipo de automatización normalmente conlleva una elevada dependencia en las versiones del programario instaladas y en la arquitectura local del ordenador en el que se procesa.

Como seguramente ya os habréis dado cuenta, lenguajes de programación como Python, *bash*, R, etc. ejecutan lo que se denominan paquetes. Los paquetes pueden contener programas que pueden ser utilizados para poder procesar los datos a analizar. Un paquete por sí solo normalmente no tiene todo el código necesario para realizar sus funciones y necesita de otros a fin de ejecutarlas. De esta forma se reutiliza código, minimizando el tiempo de desarrollo de un paquete y creando herramientas más robustas. Para poder gestionar las dependencias que un paquete tiene sobre otros aparecieron los gestores de paquetes (*apt, yum, Home Brew, pip*...).

En el proceso de utilización y actualización de los paquetes nos podemos encontrar con diversas dificultades. Durante el desarrollo de un paquete es posible que una función se vea modificada, alterando los datos de entrada o salida, o eliminada en una versión más reciente del paquete. En el momento en que distintos paquetes dependen de esta función se pueden producir problemas de compatibilidad y no es suficiente saber que un paquete depende de otro sino también si depende de una versión en específico. La actualización de un paquete puede representar que otro paquete ya no funcione. Eso conllevaría la necesidad de instalar dos versiones distintas del mismo software, hecho que muchas veces no es posible. Otra problemática asociada a las versiones de los paquetes es la reproducibilidad de resultados en sistemas distintos. Dos ordenadores podrían tener instalados los mismos paquetes y sus dependencias, pero con versiones distintas, produciendo resultados diferentes. En proyectos multicéntricos a veces es necesario ejecutar los mismos procesos en cada una de las instituciones implicadas, con la seguridad de que no habrá variabilidad en los resultados debido al procesamiento de los datos.

Así, debemos controlar:

- La utilización de los mismos paquetes y de sus dependencias.
- La implementación en otros entornos computacionales evitando que la configuración local altere el resultado final.

Para mantener una configuración estable y aplicable a diferentes infraestructuras computacionales, fundamentalmente hay dos estrategias a seguir: la creación de entornos fijos y/o la creación de contenedores.

Objetivos

1. Entender los conceptos y la utilidad de entornos y contenedores.

2. Aprender a crear y a manejar entornos Conda.

3. Saber crear y manejar contenedores Docker.

1. Conda

1.1. Introducción

Empezaremos por la creación de entornos, y como ejemplo utilizaremos Conda. Conda es un sistema de gestión de paquetes y de entornos *open source* compatible con los sistemas operativos Linux, Mac OS y Windows. No requiere privilegios de administrador para poderse utilizar.

Aunque inicialmente Conda se creó para la gestión de paquetes en Python, actualmente se pueden gestionar paquetes creados en diferentes lenguajes de programación, como R, Ruby, Scala, Java, JavaScript o C/C++.

Muchas veces hay confusión entre Conda, MiniConda y Anaconda (figura 1). El núcleo del gestor de paquetes y entornos es Conda, mientras que MiniConda es el instalador mínimo de Conda, además de combinar Python y unos paquetes básicos, y Anaconda engloba MiniConda y aumenta el número de paquetes preinstalados.

Figura 1. Esquema de las diferencias entre Conda, MiniConda y Anaconda.
Fuente: elaboración propia.

1. Conda

1.2. ¿Qué son paquetes y canales en Conda?

Un paquete es un archivo comprimido, sea un tar (.tar.bz2) o un .conda, que contiene:

- Librerías de sistema.
- Python y otros módulos.
- Ejecutables y otros componentes.
- Metadatos en el directorio info/.
- Una colección de ficheros que se instalan directamente.

Un paquete no necesariamente ha de contener todos estos elementos; por ejemplo, un paquete que únicamente contiene metadatos se denomina *metapackage*. Podéis encontrar un listado de los paquetes que gestiona Conda en https://anaconda.org/ para su instalación.

Cuando queramos instalar un paquete en un entorno Conda deberemos saber dónde localizarlo, y esa es la función de los canales. Los canales son URLs que nos dirigen a donde podemos encontrar los directorios que contienen los datos de los paquetes. Al instalar un paquete, el comando `conda` busca en un conjunto de canales que tiene por defecto. Si no se especifica lo contrario, Conda instalará los paquetes contenidos en estos canales definidos por defecto. Además de los canales por defecto, hay otro canal que ostenta un estatus diferencial. Este es el canal **conda-forge**. Este canal está administrado por la comunidad y puede tener algunas ventajas respecto a los canales por defecto gestionados por Anaconda Inc., como por ejemplo que los paquetes usualmente están más actualizados o que algunos paquetes no son accesibles desde los canales por defecto. Finalmente, destacar el canal **bioconda**, que se centra en paquetes desarrollados específicamente para el área de la bioinformática.

1. Conda

1.3. Entornos Conda

Un entorno Conda es un directorio que contiene una colección específica de paquetes Conda que has instalado. Si modificas un entorno, los otros entornos no se ven afectados y fácilmente puedes activar y desactivar entornos.

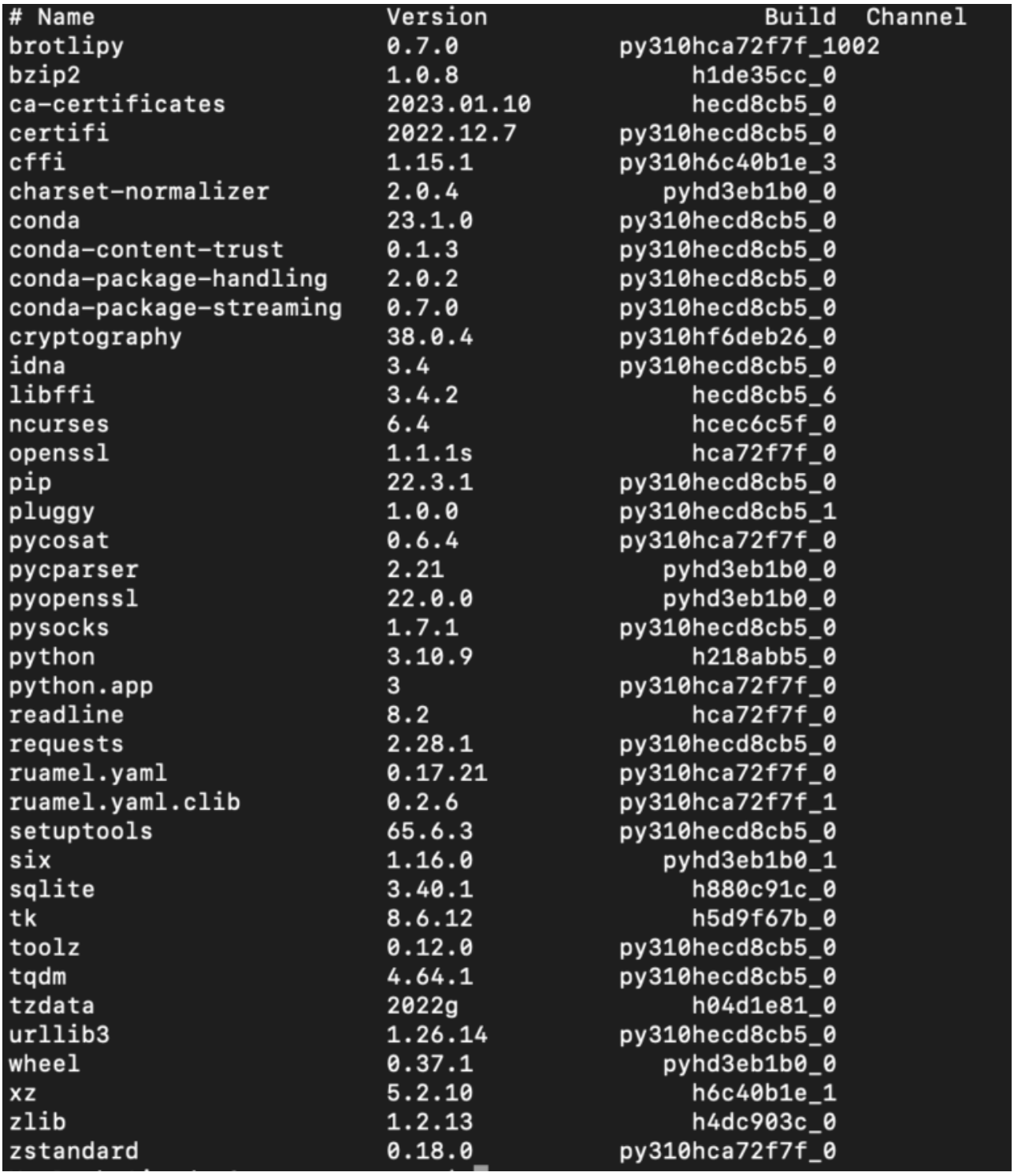
Para poder crear entornos Conda en primer lugar debéis instalar MiniConda o Anaconda, dependiendo de vuestras preferencias. Seguiremos el ejemplo utilizando MiniConda y la instalación en Linux/Mac OS. En este enlace podréis encontrar las instrucciones necesarias dependiendo del sistema operativo que utilicéis (https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html). Debéis descargaros el instalador dependiendo de vuestro sistema y, en el caso de Linux o Mac OS, ejecutar:

```
$ bash ~/miniconda.sh -b -p $HOME/miniconda
```

Para testar si la instalacion ha finalizado correctamente podéis escribir en el terminal

```
$ conda list
```

que devolverá el listado de paquetes instalados (figura 2). Recordad que si no se encuentra el comando conda seguramente es debido a que no está en la ruta especificada de comandos de vuestro terminal, pero lo podréis encontrar en la carpeta donde habéis instalado MiniConda o Anaconda, en la subcarpeta «$HOME/miniconda/bin».



```
# Name                    Version           Build  Channel
brotlipy                  0.7.0           py310hca72f7f_1002
bzip2                     1.0.8                h1de35cc_0
ca-certificates           2023.01.10           hecd8cb5_0
certifi                   2022.12.7       py310hecd8cb5_0
cffi                      1.15.1          py310h6c40b1e_3
charset-normalizer        2.0.4              pyhd3eb1b0_0
conda                     23.1.0          py310hecd8cb5_0
conda-content-trust       0.1.3           py310hecd8cb5_0
conda-package-handling    2.0.2           py310hecd8cb5_0
conda-package-streaming   0.7.0           py310hecd8cb5_0
cryptography              38.0.4          py310hf6deb26_0
idna                      3.4             py310hecd8cb5_0
libffi                    3.4.2                hecd8cb5_6
ncurses                   6.4                  hcec6c5f_0
openssl                   1.1.1s               hca72f7f_0
pip                       22.3.1          py310hecd8cb5_0
pluggy                    1.0.0           py310hecd8cb5_1
pycosat                   0.6.4           py310hca72f7f_0
pycparser                 2.21               pyhd3eb1b0_0
pyopenssl                 22.0.0             pyhd3eb1b0_0
pysocks                   1.7.1           py310hecd8cb5_0
python                    3.10.9               h218abb5_0
python.app                3               py310hca72f7f_0
readline                  8.2                  hca72f7f_0
requests                  2.28.1          py310hecd8cb5_0
ruamel.yaml               0.17.21         py310hca72f7f_0
ruamel.yaml.clib          0.2.6           py310hca72f7f_1
setuptools                65.6.3          py310hecd8cb5_0
six                       1.16.0             pyhd3eb1b0_1
sqlite                    3.40.1               h880c91c_0
tk                        8.6.12               h5d9f67b_0
toolz                     0.12.0          py310hecd8cb5_0
tqdm                      4.64.1          py310hecd8cb5_0
tzdata                    2022g                h04d1e81_0
urllib3                   1.26.14         py310hecd8cb5_0
wheel                     0.37.1             pyhd3eb1b0_0
xz                        5.2.10               h6c40b1e_1
zlib                      1.2.13               h4dc903c_0
zstandard                 0.18.0          py310hca72f7f_0
```

Figura 2. Salida del comando *conda list*.
Fuente: elaboración propia.

Conda tiene un entorno por defecto llamado *base*. No es recomendable instalar paquetes en este entorno. Si se desea iniciar un nuevo proyecto se deben crear nuevos entornos Conda.

1. Conda

1.4. Creación de entornos Conda

Es importante dar un nombre descriptivo al entorno para poder reconocer su contenido. En un *pipeline* estándar, podemos utilizar múltiples programas que pueden ser reutilizables en otros análisis. Debido a esta redundancia de procesos, es recomendable crear un entorno para cada herramienta y no crear entornos con múltiples programas. De este modo, un entorno estará definido por el software que contenga mediante el nombre, y su utilización será más sencilla que si un mismo entorno contiene múltiples paquetes, ya que será difícil determinar dónde se encuentra el programa que necesitas en un momento determinado. A veces también es recomendable no únicamente especificar el programa en el nombre del entorno, sino también la versión del mismo.

En primer lugar, instalaremos un paquete muy utilizado en Python como es `numpy`. Para crear un entorno utilizaremos el comando `create` y especificaremos la versión a instalar. Para saber qué versiones están accesibles podemos utilizar el comando `search`:

```
$ conda search numpy
```

Esto nos devolverá un listado de todas las versiones que están disponibles. Si no especificamos la versión del paquete a instalar, Conda intentará instalar la versión más nueva. Una vez seleccionamos la versión que necesitamos podemos crear nuestro nuevo entorno:

```
$ conda create -n numpy1-23-5 numpy=1.23.5
```

Utilizando la opción *-n* indicamos el nombre que queremos asignar al nuevo entorno.

Si quisiéramos especificar desde qué canal queremos instalar un paquete podemos utilizar la opción `channel`:

```
$ conda create -n numpy1-23-5b numpy=1.23.5 --channel conda-forge
```

Conda no únicamente instala el paquete especificado, sino también sus dependencias. En el caso de *numpy*, por ejemplo, no hemos especificado que instalase Python, pero al ser una dependencia automáticamente se instala en el entorno especificado.

Si siempre utilizamos un set de paquetes podemos instalarlos conjuntamente:

```
$ conda create -n basic-analisis numpy=1.23.5 pandas=1.5.3
matplotlib=3.7.1
```

Una vez creados los entornos podemos saber qué paquetes se han instalado utilizando el comando `list`:

```
$ conda list -n numpy1-23-5
```

1. Conda

1.5. Activar entornos Conda

Una vez hemos creado nuestro primer entorno Conda es hora de utilizarlo. Para obtener un listado de los entornos que tenemos en nuestro sistema utilizaremos el comando `env list`:

```
$ conda env list
```

Observaréis que además de nuestro entorno `numpy1-23-5` y `basic-analisis` tenemos el entorno **base**, que como os había comentado anteriormente está definido por defecto. Es recomendable no instalar software en este entorno.

Para poder utilizar uno de los entornos debemos activarlo:

```
$ conda activate numpy1-23-5
```

Sabremos que el entorno se ha activado, ya que aparecerá su nombre entre paréntesis delante de la línea de comandos.

1. Conda

1.6. Desactivar entornos Conda en uso

Una vez finalizados los procesos requeridos en el entorno debemos cerrar el entorno. Para ello utilizaremos el comando `deactivate`:

```
$ conda deactivate
```

De esta forma volveremos al estado inicial, antes de la activación del entorno.

1. Conda

1.7. Instalar paquetes dentro de un entorno ya creado

Los entornos Conda pueden ser modificados *a posteriori* de su creación. Simplemente debéis activar el entorno y, una vez dentro, instalar un nuevo paquete. El comando `install` por defecto instala un paquete en el entorno activo.

```
$ conda activate numpy1-23-5
```

```
( numpy1-23-5 ) $ conda install pandas=1.5.3
```

En este caso se instalaría el paquete *pandas* en el entorno `numpy1-23-5`. Si el nuevo paquete no es compatible con la configuración del entorno activo, saltará un error y no podrá ser instalado.

Si queremos especificar el canal podemos hacerlo:

```
( numpy1-23-5 ) $ conda install conda-forge::pandas=1.5.3
```

1. Conda

1.8. Renombrar entornos Conda

Si al instalar nuevos paquetes queremos especificarlo en el nombre del entorno podemos modificar el nombre asignado:

```
$ conda rename -n numpy1-23-5 numpy1-23-5-pandas1-5-3
```

1. Conda

1.9. Eliminar entornos Conda

Si un entorno ya no se utiliza, es conveniente eliminarlo y, para ello, utilizaremos el comando `remove`:

```
$ conda remove -n numpy1-23-5-pandas1-5-3 -all
```

1. Conda

1.10. Compartir entornos

En proyectos colaborativos es frecuente la necesidad de reproducir tareas en los distintos centros. Para ello se deben crear entornos agnósticos de sistema operativo y plenamente compatibles. Conda utiliza YAML (YAML *Ain't Markup Language*) como archivos de entorno que nos permitirán importar y exportar entornos.

Por convención, los archivos de entorno en Conda se llaman *environment.yml.*

Si en nuestro directorio de trabajo ejecutamos:

```
$ conda env create
```

automáticamente, Conda buscará el archivo *environment.yml.* Si no lo encuentra, saltará un error. Si el archivo de entorno tiene otro nombre, lo podemos especificar de la siguiente manera:

```
$ conda env  create --file prueba.yaml
```

Podéis observar la estructura de un archivo de entorno en la figura 3, donde se pueden apreciar tres apartados:

- `name`: nombre del entorno que se creará si no se especifica lo contrario.
- `Channels`: canales a utilizar.
- `dependencies`: paquetes a instalar con relación `canal:paquete:versión`

```yaml
name: nf-core-clipseq-1.0.0
channels:
  - conda-forge
  - bioconda
  - defaults
dependencies:
  - conda-forge::python=3.7.3
  - conda-forge::markdown=3.1.1
  - conda-forge::pymdown-extensions=6.0
  - conda-forge::pygments=2.5.2
  - conda-forge::pigz=2.3.4
  - conda-forge::perl=5.26.2

  # bioconda packages
  - bioconda::fastqc=0.11.9
  - bioconda::multiqc=1.9
  - bioconda::cutadapt=3.0
  - bioconda::bowtie2=2.4.2
  - bioconda::star=2.6.1d # Needs to be 2.6 to work with iGenomes indices
  - bioconda::samtools=1.11
  - bioconda::umi_tools=1.1.1
  - bioconda::bedtools=2.29.2
  - bioconda::subread=2.0.1
  - bioconda::preseq=2.0.3
  - bioconda::rseqc=4.0.0

  # peak calling packages - may need to switch to pip for latest versions
  - bioconda::icount=2.0.0
  - bioconda::paraclu=9
  - bioconda::pureclip=1.3.1
  - bioconda::piranha=1.2.1

  # motif calling packages
  - bioconda::meme=5.1.1
```

Figura 3. Ejemplo de archivo de entorno YAML.
Fuente: elaboración propia.

Para poder generar un archivo YAML de un entorno que nosotros hemos creado, ejecutaremos:

```
$ conda env export -n basic-analisis --file basic.yaml
```

Especificamos el nombre del archivo con la opción `--file`

Para asegurar que el entorno puede ser reproducible independientemente del sistema operativo, hace falta especificar la opción `--from-history`:

```
$ conda env export -n basic-analisis --from-history --file basic.yaml
```

2. Docker

2.1. Introducción

Una vez hemos visto los entornos Conda, nos introduciremos en otra metodología para controlar los procesos que utilizamos: los contenedores (*).

Los contenedores son sistemas de virtualización que contienen todas las herramientas necesarias para ejecutar un software. Muy a menudo se comparan las máquinas virtuales con los contenedores. La diferencia más importante es que las máquinas virtuales virtualizan toda una máquina hasta las capas de *hardware*, mientras que los contenedores únicamente virtualizan la capa de software encima del sistema operativo. Esta característica los hace más ligeros y fáciles de modificar.

Aunque los contenedores no son una tecnología nueva, su aplicación de manera extensa empezó con la aparición de Docker en 2013. La popularización de estas aplicaciones introdujo la complejidad de administrar cientos o miles de contenedores, y por ello apareció lo que se conoce como la orquestación de contenedores. Aunque a lo largo del tiempo han aparecido distintas plataformas de orquestación, inclusive una del mismo Docker, como es Docker Swarm, Google creo en 2014 Kubernetes, de código abierto, que se ha convertido en el software preferido de muchas empresas y se ha consolidado como un estándar. Las plataformas de orquestación se encargan de reiniciar las aplicaciones si fallan, de equilibrar la carga de trabajo, de escalar automáticamente, de implementar sin tiempo de inactividad, etc.

Aunque en sistemas HPC (High Performance Computing) se utiliza más Singularity, en este apartado de contenedores nos centraremos en cómo utilizar Docker.

2. Docker

2.2. Contenedores Docker

En primer lugar, necesitaremos instalar Docker en nuestro ordenador. La forma más sencilla es utilizando Docker Desktop (https://www.docker.com/products/docker-desktop/).

Para saber que tienes Docker instalado correctamente en el terminal puedes interrogar su versión:

```
$ docker -v
```

El *output* te indicará la versión y el *build* que tienes instalados.

Y también:

```
$ docker system info
```

Si saltase algún error se debería comprobar que el proceso de instalación estuviera finalizado correctamente y que Docker Desktop estuviera abierto.

Para crear un contenedor necesitas lo que se denominan imágenes. Las imágenes son los moldes de los contenedores, son las recetas/instrucciones para crear los contenedores.

2. Docker

2.3. Descargar contenedores creados

En primer lugar, miraremos si tenemos alguna imagen en nuestro sistema. En teoría, si es la primera vez que utilizas Docker, te debería aparecer una lista en blanco al utilizar:

```
$ docker image ls
```

Empezaremos por el contenedor más sencillo y lo bajaremos directamente:

```
$ docker image pull hello-world
```

Hay el proceso de descarga y si todo ha funcionado correctamente al repetir el comando de listado de imágenes os debería aparecer la imagen `hello-world`.

La imagen de «hello-world» procede de Docker Hub (*), un repositorio de imágenes.

Seguidamente, ejecutaremos el contenedor que se cree a partir de la imagen `hello-world` mediante:

```
$ docker container run hello-world
```

Una vez ejecutado recibiréis un mensaje de parte del equipo de Docker (figura 4).

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figura 4. Mensaje de bienvenida del equipo de Docker al ejecutar «hello-world».
Fuente: elaboración propia.

Cuando se ejecuta un contenedor suceden tres procesos:

- Inicializa el contenedor a partir de la imagen.
- Se ejecuta la acción preestablecida del contenedor si esta existe.
- Una vez la acción ha finalizado el contenedor se para.

En el caso del ejemplo anterior la acción preestablecida era imprimir el mensaje de bienvenida, pero la acción realizada puede ser mucho más compleja.

Además de poder ejecutar los comandos predeterminados por el contenedor también le podemos pasar comandos o entrar en modo interactivo. Para probar estas opciones ejecutaremos el contenedor Alpine que contiene una distribución de Ubuntu muy simple.

```
$ docker container run -it alpine sh
```

En este comando utilizamos la opción `it` para ser interactivo y `sh` nos especifica que el terminal que queremos utilizar es *bash*.

Veréis que la línea de comandos cambia a:

```
/ #
```

Podréis comprobar que ahora estáis dentro de Alpine y al ejecutar

```
/ #  cat /etc/os-release
```

os mostrará la versión de Alpine que os habéis bajado.

De esta manera, una vez ejecutado el contenedor no se ha finalizado como habíamos observado anteriormente, sino que se mantiene activo y responde a los comandos que introduzcamos.

Para poder salir del contenedor y finalizar su ejecución podéis introducir el comando `exit`.

2. Docker

2.4. Eliminar imágenes y contenedores Docker

Siempre es recomendable mantener una buena gestión de imágenes y contenedores, ya que van ocupando espacio innecesariamente si no se utilizan. Así aprenderemos a eliminar imágenes y contenedores.

Para poder eliminar imágenes, necesitaremos saber la Image ID. Para ello utilizaremos el siguiente comando:

```
$ docker image ls
```

Nos aparecerá un *output* similar al siguiente:

Tabla 1.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
alpine	latest	9ed4aefc74f6	10 days ago	7.05MB
hello-world	latest	feb5d9fea6a5	18 months ago	13.3kB

Fuente: elaboración propia.

Así para eliminar una imagen simplemente se debe especificar el Image ID tal que así:

```
$ docker image rm feb5d9fea6a5
```

O utilizando su nombre:

```
$ docker image rm hello-world
```

Muchas veces nos podemos encontrar con un mensaje de error:

```
Error response from daemon: conflict: unable to delete feb5d9fea6a5 (must
be forced) - image is being used by stopped container 06dda9650983
```

Esto nos está indicando que hay un contenedor que aún está activo o que aún no ha sido limpiado que está o ha utilizado esta imagen. Para ello primero debemos eliminar los contenedores que están impidiendo la eliminación de esta imagen.

Primero listamos los contenedores activos:

```
$ docker container ls
```

O, de manera similar, podemos utilizar:

```
$ docker ps
```

Es posible que no nos devuelva ningún contenedor activo. En realidad, el mensaje de error previo nos indicaba que el contenedor había parado; por lo tanto, tiene sentido no encontrar el contenedor en este listado. Así necesitaremos ver los contenedores tanto activos como los que recientemente se han parado:

```
$ docker container ls -all
```

Aquí sí que podemos ver el contenedor anteriormente especificado como enlazado a «hello-world»:

Tabla 2.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
8a149fe84874	alpine	«sh»	About an hour ago	Exited		xenodochial_thompson
06dda9650983	hello-world	«/hello»	12 hours ago	Exited		upbeat_satoshi

Fuente: elaboración propia.

Para evitar que se vayan acumulando los contenedores se puede añadir al comando `run` una opción de eliminar automáticamente una vez finalizado:

```
$ docker container run --rm hello-world
```

Si no hemos especificado la opción `rm`, debemos eliminarlo manualmente utilizando como referencia el Container ID:

```
$ docker container rm 06dda9650983
```

Si se elimina correctamente nos devolverá el Container ID en el terminal. Se puede borrar más de un Container ID simultáneamente.

Si tienes varios contenedores asociados a una imagen y quieres borrarlos todos utilizando un patrón puedes:

```
$ docker container ps -a | grep "world" | awk '{print $1}' | xargs docker rm
```

En este caso utilizamos el comando `xargs` para controlar una lista de argumentos para poder ser eliminados mediante `rm` de Docker.

Ahora veréis que al listar los contenedores ya no aparece el contenedor, y si probamos de eliminar la imagen como previamente habíamos intentado lo hará sin problemas.

```
$ docker image rm hello-world
```

```
$ docker image ls
```

Si se quiere eliminar todos los contenedores existentes podemos utilizar:

```
$ docker container prune
```

De esta manera todos los contenedores existentes serán eliminados y si queremos volver a reconectarlos deberemos iniciarlos de nuevo.

2. Docker

2.5. Parar contenedores en Docker

Iniciar o parar un contenedor no es lo mismo que iniciar o parar un proceso. Para finalizar un contenedor, Docker proporciona los comandos `stop` y `kill`. Aunque parezca que ambos comandos hagan lo mismo, internamente su ejecución es distinta. Para parar un proceso, tanto podemos utilizar el ContainerId como el nombre del contenedor.

El comando `stop` para el contenedor de una manera menos agresiva que `kill`. Esto es debido al tipo de señal que se envía al contenedor. stop envía una señal SIGTERM, la cual se puede bloquear o parar, mientras que `kill` envía una señal SIGKILL que no se puede gestionar. Si en un tiempo prudencial el comando stop no ha parado el contenedor, Docker automáticamente envía una señal SIGKILL. Por defecto ese tiempo son 10 segundos, pero si queremos modificarlo podemos utilizar la opción *-t* expresada en segundos. Así:

```
$ docker container stop -t 77 hello-world
```

Docker pararía el proceso mediante una SIGKILL pasador 77 segundos.

Como hemos visto en el apartado anterior también podríamos utilizar el comando `rm` para finalizar un contenedor. La diferencia reside en que el comando `rm` elimina el proceso y no lo podemos visualizar en la lista `docker ps -a`, mientras que parando el contenedor lo podemos mantener y reutilizar posteriormente.

Otra opción interesante es pausar el contenedor. Mientras que si paramos un contenedor liberamos los recursos de memoria y CPU, al pausarlo únicamente liberamos CPU.

```
$ docker container pause hello-world
```

2. Docker

2.6. Buscar contenedores en Docker Hub

Un recurso que ya hemos utilizado para el contenedor «hello-world» es Docker Hub. En este repositorio podemos encontrar muchos contenedores ya creados. Muchos de ellos ya han sido construidos y testeados por los mismos desarrolladores del software que estáis buscando. Como ejemplo iremos a la página del *variant caller* GATK (https://hub.docker.com/r/broadinstitute/gatk). Aquí encontraréis las instrucciones para podéroslo bajar. Si queréis una versión determinada al bajar la imagen necesitáis especificarla. Para ello se utilizan los *tags* igual que como vimos en el apartado de Conda. En la página de GATK tenéis una pestaña donde tenéis las distintas versiones indicadas por *tags*.

Para indicar qué versión queremos utilizar lo indicaremos con los «`:`»:

```
$ docker image pull broadinstitute/gatk:4.4.0.0
```

Si no especificamos la versión se bajará la imagen más reciente denominada *`latest`*.

Como podéis apreciar en el comando `pull` que hemos utilizado para bajar GATK delante del nombre de `gatk` tenemos el nombre de la institución que lo ha creado y hecho público. En este caso el Broad Institute. Si este nombre previo no aparece, indica que los desarrolladores son el propio equipo de Docker.

Es importante tener en cuenta que cualquier persona puede crear una cuenta en Docker Hub, por lo tanto, es importante mantenerse cauteloso al bajar software de fuentes no contrastadas. Debéis procurar bajar imágenes directamente de los desarrolladores o de comunidades establecidas. Docker mantiene una serie de imágenes referenciadas como Docker Official Images, las cuales han sido analizadas y proporcionan repositorios básicos para la comunidad, como por ejemplo Ubuntu o Centos.

Otra opción para buscar imágenes es el comando `search`. En este caso, al utilizar

```
$ docker search GATK
```

obtendremos un listado de imágenes donde encontraríamos repositorios con GATK (figura 5).

```
NAME                          DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
broadinstitute/gatk           Official release repository for GATK version…   91
bitnami/keycloak-gatekeeper   Bitnami Keycloak Gatkeeper Docker Image         8
broadinstitute/gatk3          Official release repository for GATK version…   4
kfdrc/gatk                    alpine based gatk                               2                    [OK]
gatksv/sv-pipeline                                                            1
broadinstitute/gatk-nightly   Official repository for nightly development …   1                    [OK]
mgibio/gatk-cwl               Image containing gatk, for use in cwl workfl…   1                    [OK]
gatksv/sv-pipeline-base                                                       0
gatksv/sv-pipeline-rdtest                                                     0
gatksv/gatk                                                                   0
gatksv/cnmops                                                                 0
gatksv/samtools-cloud                                                         0
gatksv/sv-base-mini                                                           0
gatksv/wham                                                                   0
gatkworkflows/mtdnaserver                                                     0
gatksv/manta                                                                  0
dukegcb/gatk-base             Dependencies (Java, R) for running GATK         0                    [OK]
gatksv/sv-base                                                                0
gatksv/delly                                                                  0
gatksv/sv-pipeline-qc                                                         0
biocontainers/gatk                                                            0
pegi3s/gatk-3                 GATK 3 (https://gatk.broadinstitute.org/hc/e…   0
pegi3s/gatk-4                 GATK 4 (https://gatk.broadinstitute.org/hc/e…   0
jsotobroad/gatk                                                               0
agrf/gatk                     https://software.broadinstitute.org/gatk/       0
```

Figura 5. *Output* de la búsqueda de imágenes que contengan GATK.
Fuente: elaboración propia.

2. Docker

2.7. Crear tu propia imagen Docker

Si tu búsqueda en Docker Hub ha sido infructuosa o requieres una imagen específica para tus necesidades, la solución es crearla tú mismo.

En primer lugar, instalaremos software en una sesión interactiva. Para ello utilizaremos la imagen de Alpine que previamente habíamos creado e intentaremos instalar en paquete de *python numpy*. Alpine no es la distribución en la que querríais basaros para desarrollar vuestros proyectos; seguramente Ubuntu o Debian sean unas elecciones más apropiadas. Antes de empezar una imagen es importante que sepáis qué necesitaréis instalar, cuáles son vuestros requerimientos y escoger la distribución más apropiada. También es una buena práctica no instalar muchos programas en cada imagen, ya que serán más difíciles de crear y mantener. Es la misma filosofía que aplicábamos a los entornos Conda.

```
$ docker container run -it alpine sh
```

Seguidamente comprobaremos si tenemos Python instalado:

```
/ # python --version
```

Y podemos observar que no está instalado.

Alpine tiene un gestor de paquetes llamado *Alpine Package Keeper* (apk) para instalar software. Dependiendo de la distribución de Linux que os hayáis instalado podríais utilizar `apt-get, yum, zypper`...

En primer lugar, instalaremos Python y otros paquetes necesarios, y posteriormente el paquete *numpy*.

```
/ # apk --no-cache --update-cache add gcc gfortran python3 py3-pip
python3-dev build-base wget freetype-dev libpng-dev openblas-dev

/ # pip install numpy
```

Si ahora entramos en Python podemos comprobar que se ha instalado correctamente el paquete *numpy*.

Una vez salgas de este contenedor los cambios no se habrán guardado. Para crear tu propia imagen lo más recomendable es utilizar un Dockerfile.

Un Dockerfile es un archivo de texto con la siguiente estructura mínima:

- FROM <Imagen preexistente>
- RUN <Comandos para instalar desde la línea de comandos>
- CMD <Comandos que deben correrse por defecto>

Las instrucciones que se lanzan por defecto (CMD) tienen una estructura definida. Solo puede haber una línea de CMD en el Dockerfile; si hubiera más de una se ejecutaría la última.

Así si quisiéramos reproducir el contenedor anterior deberíamos crear un archivo nombrado *Dockerfile* de la siguiente manera:

```
FROM alpine

RUN apk --no-cache --update-cache add gcc gfortran python3 py3-pip
python3-dev build-base wget freetype-dev libpng-dev openblas-dev

RUN pip install numpy

CMD python3 --version
```

Una vez tenemos el Dockerfile definido crearemos la imagen:

```
$ docker image build -t uoc/numpy .
```

La opción `-t` nos indica el nombre que le queremos dar a nuestra imagen. Siguiendo la nomenclatura anteriormente mencionada indicamos el autor y el paquete. El «.» nos indica que Dockerfile está en la misma carpeta donde estamos ejecutando los comandos; debemos especificar la ruta al directorio del Dockerfile.

Ahora podréis ver la nueva imagen creada:

```
$ docker image ls
```

Tabla 3.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
uoc/numpy	latest	af1700de32cb	25 minutes ago	588MB

Si corremos el contenedor Docker nos devolverá la versión de Python que tenemos instalada en la imagen:

```
$ docker container run uoc/numpy
```

Si entrásemos comandos en la línea de ejecución del contenedor, estos son los que se utilizarían en detrimento de los que estuviesen en el Dockerfile. Así:

```
$ docker container run uoc/numpy which python3
```

Te indicará la localización de Python3 en el *filesystem* del contenedor.

2. Docker

2.8. Renombrar imágenes

En realidad, lo que crearemos es una copia de una imagen con un nuevo nombre. Para ello haremos:

```
$ docker image tag uoc/numpy uoc/alpine-numpy
```

Si la imagen es susceptible a evolucionar es importante mantener una numeración por sus *tags*, como hemos visto en Docker Hub:

```
$ docker image tag uoc/alpine-numpy uoc/alpine-numpy:1.0.0
```

Si ahora miráis las imágenes que tenéis creadas podréis ver que en la columna *tag* tendréis la versión que hemos creado ahora y la *latest*, que es por defecto, si no se especifica uno.

Tabla 4.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
uoc/numpy	1.0.0	af1700de32cb	25 minutes ago	588MB
uoc/numpy	latest	af1700de32cb	25 minutes ago	588MB

Fuente: elaboración propia.

2. Docker

2.9. Ejecución de *scripts*

Como hemos visto, la creación de un contenedor es relativamente fácil, pero dependiendo de vuestras necesidades es posible que los ejemplos anteriores se queden cortos. Por ello extenderemos los conocimientos necesarios para crear imágenes más complejas.

En primer lugar, utilizaremos el contenedor con Alpine que hemos creado anteriormente para ejecutar un *script* en Python.

Si directamente pasamos los parámetros al iniciar el contenedor, nos devolverá un error, ya que en su sistema el archivo no se encuentra:

```
$ docker container run --rm uoc/alpine-numpy python3 num.py
```

Siendo `num.py`:

```
import numpy as np

a = np.array([1,2,3,4,5,6,7])

print (a)
```

El error siendo:

```
python3: can't open file '//num.py': [Errno 2] No such file or directory
```

Para que funcione este comando necesitaremos enlazar los dos sistemas, el nuestro propio y el del contenedor Docker. Para ello utilizaremos el comando mount especificando dónde está nuestro archivo y posicionándolo en el sistema del contenedor.

```
$ docker container run --rm --mount type=bind,source=${PWD},target=/temp
uoc/alpine-numpy python3 /temp/num.py
```

Hay distintos tipos de mount. En este caso utilizaremos el modo *bind*, que es el que nos interesa para este ejemplo. Con el *source* especificamos el directorio donde se ubica el *script*, podemos utilizar la variable ${PWD} si lanzamos el contenedor desde el mismo directorio donde está el *script*, y *target*, que especifica dónde lo localizaremos en el contenedor. Es importante que cuando ejecutamos Python3 especifiquemos la carpeta donde localizamos el *script*, en este caso /temp.

De esta manera nos retornará:

```
[1 2 3 4 5 6 7]
```

2. Docker

2.10 Volúmenes Docker

En el ejemplo anterior hemos utilizado `mount` para unir los dos sistemas. `Mount` depende del sistema en el cual se ejecuta, mientras que los **volúmenes** son nativos de Docker. Los volúmenes pueden compartirse entre contenedores y persisten más que los contenedores.

En el ejemplo anterior, podríamos pasar un volumen al contenedor con la opción `-v`:

```
$ docker container run --rm -v $(PWD):/temp uoc/alpine-numpy python3
/temp/num.py
```

Definimos la carpeta local $(PWD) y dónde se localiza en el contenedor /temp.

Para crear un volumen con nombre `data_set` y asignarle una carpeta local:

```
$ docker volume create --driver local --opt
device=/Path/al/directorio/local --opt type=none --opt o=bind data_set
```

En este caso podremos lanzar:

```
$ docker container run --rm -v data_set:/temp uoc/alpine-numpy python3
/temp/num.py
```

Podemos visualizar los distintos volúmenes:

```
$ docker volume ls
```

... inspeccionarlos

```
$ docker volume inspect data_set
```

... y eliminarlos:

```
$ docker volume rm data_set
```

2. Docker

2.11. Integrar datos en el contenedor Docker

Muchas veces necesitaremos utilizar un *input* de manera sistemática en un contenedor. Por ejemplo, si queremos hacer un *variant call* utilizando GATK, los genomas de referencia siempre serán los mismos y tal vez es interesante guardarlos dentro del contenedor para asegurarnos la reproducibilidad de los resultados a fin de que no dependa de la referencia utilizada. Para un caso más simple introduciremos nuestro *script num.py* en el contenedor. Para ello crearemos una nueva imagen modificando el Dockerfile.

Añadiremos una nueva línea:

```
COPY num.py /home
```

Estaremos haciendo una copia del *script* en el home del contenedor. Hacemos otra imagen:

```
$ docker image build -t uoc/alpine-num .
```

Si ahora entramos de forma interactiva dentro del contenedor y listáis los archivos dentro de /home encontrareis el archivo *num.py*:

```
$ docker container run --rm -it uoc/alpine-num sh
```

El orden de los comandos en el Dockerfile es importante. Es recomendable hacer los comandos COPY después de los RUN, ya que al hacer el *build*, Docker va por orden, y si en algún momento queremos añadir otro *script* en lugar de *num.py*, si el COPY está al final del proceso, Docker utiliza los RUNs que tiene en memoria y no ha de construir la imagen de cero, y el proceso es mucho más rápido. Docker va línea a línea, y si esa capa o conjunto de capas ya la tiene en memoria *caché* agiliza el proceso no reinstalándolas.

Si los datos que queremos introducir dentro de nuestra imagen están en internet, directamente podemos copiarlos utilizando RUN. Podríamos añadir estas líneas en el Dockerfile como ejemplo:

```
RUN wget
https://ftp.ncbi.nlm.nih.gov/refseq/H_sapiens/annotation/GRCh38_latest/refse
```

Por defecto, el archivo se copia en el *root* del sistema del contenedor. Si quisiéramos moverlo a otra localización podríamos especificarla:

```
RUN mv GRCh38_latest_clinvar.vcf.gz /home
```

y la copiaría en nuestra carpeta /home.

Ahora que sabemos cómo introducir el *script* dentro del contenedor, podemos crear un nuevo contenedor que corra el *script* automáticamente. Debemos substituir CMD del Dockerfile a:

```
CMD ["python3","/home/num.py"]
```

De esta manera creamos una nueva imagen:

```
$ docker image build -t uoc/alpine-numpy-ex .
```

y ejecutamos directamente:

```
$ docker container run --rm uoc/alpine-numpy-ex
```

y nos devuelve el resultado del archivo *num.py*.

Si queremos introducir un nuevo *script* de Python (*atcg.py*) donde su resultado dependa de un argumento,

```
import sys

filename = sys.argv[1]

filenumb = len(filename)

print (filenumb)
```

lo copiaremos mediante Dockerfile:

```
COPY atcg.py /home
```

y crearemos una nueva imagen:

```
$ docker image build -t uoc/alpine-atcg .
```

Si lo corremos directamente, tendremos el resultado del CMD (en nuestro caso, la versión de Python); mientras que si añadimos argumentos al comando `run`, sobrescribe CMD y obtenemos el resultado del nuevo *script*:

```
$ docker container run --rm uoc/alpine-atcg python3 /home/atcg.py ATG
```

Si quisiéramos tener este nuevo *script* como los comandos por defecto deberíamos cambiar en el Dockerfile la línea de CMD por los valores por defecto y crear una nueva línea nombrada ENTRYPOINT para localizar el *script*:

```
ENTRYPOINT ["python3","atcg.py"]

CMD ["ACTG"]
```

También podemos añadir antes que ENTRYPOINT y CMD una entrada para definir el directorio de trabajo:

```
WORKDIR /home
```

De esta manera, si creamos una nueva imagen `uoc/alpine-entry` y ejecutamos el contenedor directamente,

```
$ docker container run  --rm uoc/alpine-entry
```

nos devolverá «4» la longitud de la línea «ACTG» ubicada por defecto en el Dockerfile. Si ahora al final del comando `run` añadimos otro *input*, este sustituirá al de CMD:

```
$ docker container run  --rm uoc/alpine-entry Genetica
```

Devolverá «8».

La relación entre ENTRYPOINT y CMD la podemos observar en la siguiente tabla.

Tabla 5.

	No ENTRYPOINT	**ENTRYPOINT** exec_entry p1_entry	**ENTRYPOINT** [«exec_entry», «p1_entry»]
No CMD	error, not allowed	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [«exec_cmd», «p1_cmd»]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd

CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

Fuente: elaboración propia.

Resumen

En este apartado habéis aprendido cómo crear y utilizar entornos Conda y contenedores Docker. Utilizar este tipo de herramientas es muy importante en la reproducibilidad de procesos y resultados. Tanto si has de compartir tu código con otras personas como si debes repetir el mismo procedimiento con otras muestras en un futuro, es crucial poder tener un seguimiento de las herramientas utilizadas. Estos procedimientos mejoran la calidad de la investigación publicada y facilitan el proceso de revisión por parte de investigadores externos. Actualmente prácticamente todos los programas dependen de otros paquetes de programas, los cuales evolucionan independientemente. La modificación de una dependencia puede conllevar a diferentes resultados de un programa o inclusive al mal funcionamiento de este. Por ello es importante mantener los programas y sus dependencias tal y como se utilizaron para poder asegurar la correcta reproducibilidad. La utilización de pequeños entornos y contenedores facilita el mantenimiento y el replicado de procesos. Además, tener los programas aislados del sistema general permite la eliminación y actualización de estos de una manera más sencilla y limpia. Algunos programas pueden necesitar ciertas dependencias incompatibles con otras ya instaladas en el sistema, y por ello la creación de entornos o la utilización de contenedores facilita el trabajar en distintas configuraciones en el mismo sistema.

Actividades

Conda

1. Cread un nuevo entorno de nombre «UOC-bioinformatics» que contenga Python 2.7.

2. Instalad en este entorno ya creado R y el paquete Tidyverse en su última versión.

3. Enumerad los paquetes instalados en el entorno de la actividad anterior.

4. Instalad el paquete Scipy en un entorno nuevo a través del canal Bioconda.

5. Cread un archivo «environment.yml» de este último entorno.

Docker

1. Bajaos la imagen de Ubuntu de Docker Hub, entrad en el contenedor de forma interactiva y determinad la versión de Ubuntu descargada.

2. Repetid el procedimiento anterior con la versión previa a la *latest*.

3. Cread un Dockerfile donde añadáis la última versión de R al sistema operativo Centos.

4. Cread un volumen Docker donde le podáis pasar al contenedor un *script* en *bash* que retorne «Hello World».

5. Repetid el procedimiento anterior modificando el mensaje a través del terminal al correr el contenedor y que retorne «Hi, I'm back».

Bibliografía

Björn Grüning y otros (2018). Practical Computational Reproducibility in the Life Sciences. *Cell Systems*, *6* (6), p. 631-635. https://docs.anaconda.com/free/anacondaorg/user-guide/

Carole Goble y otros (2020). FAIR Computational Workflows. *Data Intelligence,* 2, p. 108-121.

Mark D. Wilkinson y otros (2016). The FAIR Guiding Principles for Scientific Data Management and Stewardship. *Scientific Data*, 3. 160018

Serghei Mangul y otros (2019). Challenges and Recommendations to Improve the Installability and Archival Stability of Omics Computational Tools. *PLoS Biology*, 17, e3000333.

Sean P. Kane, Karl Matthias (2023). *Docker: Up & Running* (3rd Edition). O'Reilly Media.

Victoria Stodden y otros (2018). An Empirical Analysis of Journal Policy Effectiveness for Computational Reproducibility. *Proceedings of the National Academy of* Sciences, 115, p. 2.584-2.589.

Wade L. Schulz y otros (2016). Use of Application Containers and Workflows for Genomic Data Analysis. *Journal of Pathology Informatics*, *7* (53). https://doi.org/10.4103/2153-3539.197197 (2016).

Yang-Min Kim y otros (2018). Experimenting with Reproducibility: A Case Study of Robustness in Bioinformatics. *Gigascience*, 7, giv077.

Yuxing Yan, James Yan (2018). *Hands-On Data Science with Anaconda*. Packt Publishing https://docs.docker.com/

Zachary D. Stephens y otros (2015). Big Data: Astronomical or Genomical? *PLoS Biol*ogy, *13*, e1002195.