# Entorns i contenidors

Autors: Guerau Fernandez Isern, Joan Colomer Vila, Maria Begoña Hernández Olasagarre, Enrique Blanco García
L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: Josep Jorba Esteve
PID_00298304
Primera edició: setembre 2023

# Introducció

A mesura que la quantitat de dades en l'àrea de ciències de la vida i en altres àrees del coneixement augmenta exponencialment, sorgeix la necessitat d'estructurar i manejar sistemàticament la informació adquirida. Que les dades que manegem segueixin els principis FAIR (*) és fonamental per a la correcta manipulació de les dades i de la seva traçabilitat al llarg del cicle de vida de la dada. A més, hi ha dos processos fonamentals, els quals estudiarem en els propers temes, que són l'escalabilitat i la reproductibilitat.

Tradicionalment, per tal de minimitzar el nombre de passos manuals que es duen a terme en una anàlisi, els processos s'han unit programàticament en els anomenats *pipelines*. Aquesta concatenació de processos normalment s'estructura en arxius (*scripts*). Aquest tipus d'automatització normalment comporta una elevada dependència en les versions del programari instal·lades i en l'arquitectura local de l'ordinador en el qual es processa.

Com segurament ja heu vist, llenguatges de programació com Python, *bash*, R, etc. executen el que s'anomenen paquets. Els paquets poden contenir programes que poden ser utilitzats per poder processar les dades a analitzar. Un paquet per si sol normalment no té tot el codi necessari per realitzar les seves funcions i en necessita d'altres per tal d'executar-les. D'aquesta manera es reutilitza codi, minimitzant el temps de desenvolupament d'un paquet i creant eines més robustes. Per poder gestionar les dependències que un paquet té sobre d'altres van aparèixer els gestors de paquets (*apt*, *yum*, *Home Brew*, *pip*...).

En el procés d'utilització i actualització dels paquets ens podem trobar amb diverses dificultats. Durant el desenvolupament d'un paquet és possible que una funció es vegi modificada, alterant les dades d'entrada o sortida, o eliminada en una versió més recent del paquet. En el moment en què diferents paquets depenen d'aquesta funció es poden produir problemes de compatibilitat i no és suficient saber que un paquet depèn d'un altre sinó també si depèn d'una versió en específic. L'actualització d'un paquet pot representar que un altre paquet ja no funcioni. Això comportaria la necessitat d'instal·lar dues versions diferents del mateix *software*, fet que moltes vegades no és possible. Una altra problemàtica associada a les versions dels paquets és la reproductibilitat de resultats en sistemes diferents. Dos ordinadors podrien tenir instal·lats els mateixos paquets i les seves dependències, però amb versions diferents, produint resultats diferents. En projectes multicèntrics de vegades cal executar els mateixos processos en cadascuna de les institucions implicades, amb la seguretat que no hi haurà variabilitat en els resultats a causa del processament de les dades.

Així, hem de controlar:

- La utilització dels mateixos paquets i de les seves dependències.
- La implementació en altres entorns computacionals, evitant que la configuració local alteri el resultat final.

Per mantenir una configuració estable i aplicable a diferents infraestructures computacionals, fonamentalment hi ha dues estratègies a seguir: la creació d'entorns fixos i la creació de contenidors.

# Objectius

1. Entendre els conceptes i la utilitat d'entorns i contenidors.
2. Aprendre a crear i a manejar entorns Conda.
3. Saber crear i manejar contenidors Docker

# 1. Conda

## 1.1. Introducció

Començarem per la creació d'entorns, i com a exemple utilitzarem Conda. Conda és un sistema de gestió de paquets i d'entorns *open source* compatible amb els sistemes operatius Linux, Mac OS i Windows. No requereix privilegis d'administrador per poder-se fer servir.

Tot i que inicialment Conda es va crear per a la gestió de paquets a Python, actualment es poden gestionar paquets creats en diferents llenguatges de programació, com R, Ruby, Scala, Java, JavaScript o C/C++.

Moltes vegades hi ha confusió entre Conda, MiniConda i Anaconda (figura 1). El nucli del gestor de paquets i entorns és Conda, mentre que MiniConda és l'instal·lador mínim de Conda, a més de combinar Python i uns paquets bàsics, i Anaconda engloba MiniConda i augmenta el nombre de paquets preinstal·lats.

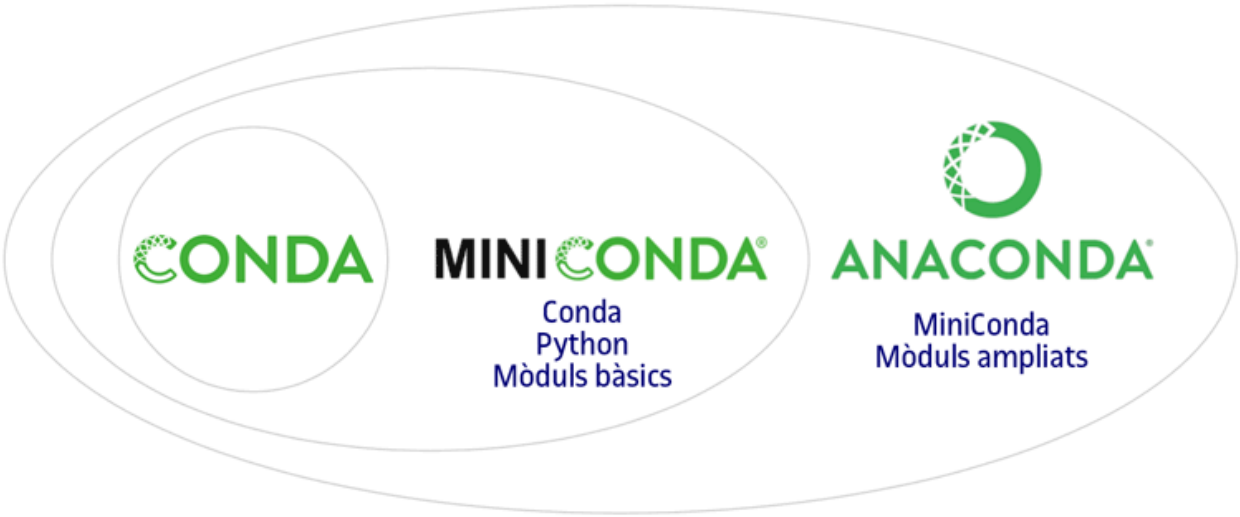



Figura 1. Esquema de les diferències entre Conda, MiniConda i Anaconda.
Font: elaboració pròpia.

# 1. Conda

## 1.2. Què són paquets i canals a Conda?

Un paquet és un arxiu comprimit, sigui un tar (.tar.bz2) o un .conda, que conté:

- Llibreries de sistema.
- Python i altres mòduls.
- Executables i altres components.
- Metadates en el directori info/.
- Una col·lecció de fitxers que s'instal·len directament.

Un paquet no necessàriament ha de contenir tots aquests elements; per exemple, un paquet que únicament conté metadades es denomina *metapackage*. Podeu trobar un llistat dels paquets que gestiona Conda a https://anaconda.org/ per a la seva instal·lació.

Quan vulguem instal·lar un paquet en un entorn Conda haurem de saber on localitzar-lo, i aquesta és la funció dels canals. Els canals són URLs que ens dirigeixen on podem trobar els directoris que contenen les dades dels paquets. En instal·lar un paquet, l'ordre `conda` busca en un conjunt de canals que té per defecte. Si no s'especifica el contrari, Conda instal·larà els paquets continguts en aquests canals definits per defecte. A més dels canals per defecte hi ha un altre canal que ostenta un estatus diferencial. Aquest és el canal **conda-forge**. Aquest canal està administrat per la comunitat i pot tenir alguns avantatges respecte als canals per defecte gestionats per Anaconda Inc., com per exemple que els paquets usualment estan més actualitzats o que alguns paquets no són accessibles des dels canals per defecte. Finalment, cal destacar el canal **bioconda**, que se centra en paquets desenvolupats per a la bioinformàtica.

# 1. Conda

## 1.3. Entorns Conda

Un entorn Conda és un directori que conté una col·lecció específica de paquets Conda que has instal·lat. Si modifiques un entorn, els altres entorns no es veuen afectats i fàcilment pots activar i desactivar entorns.

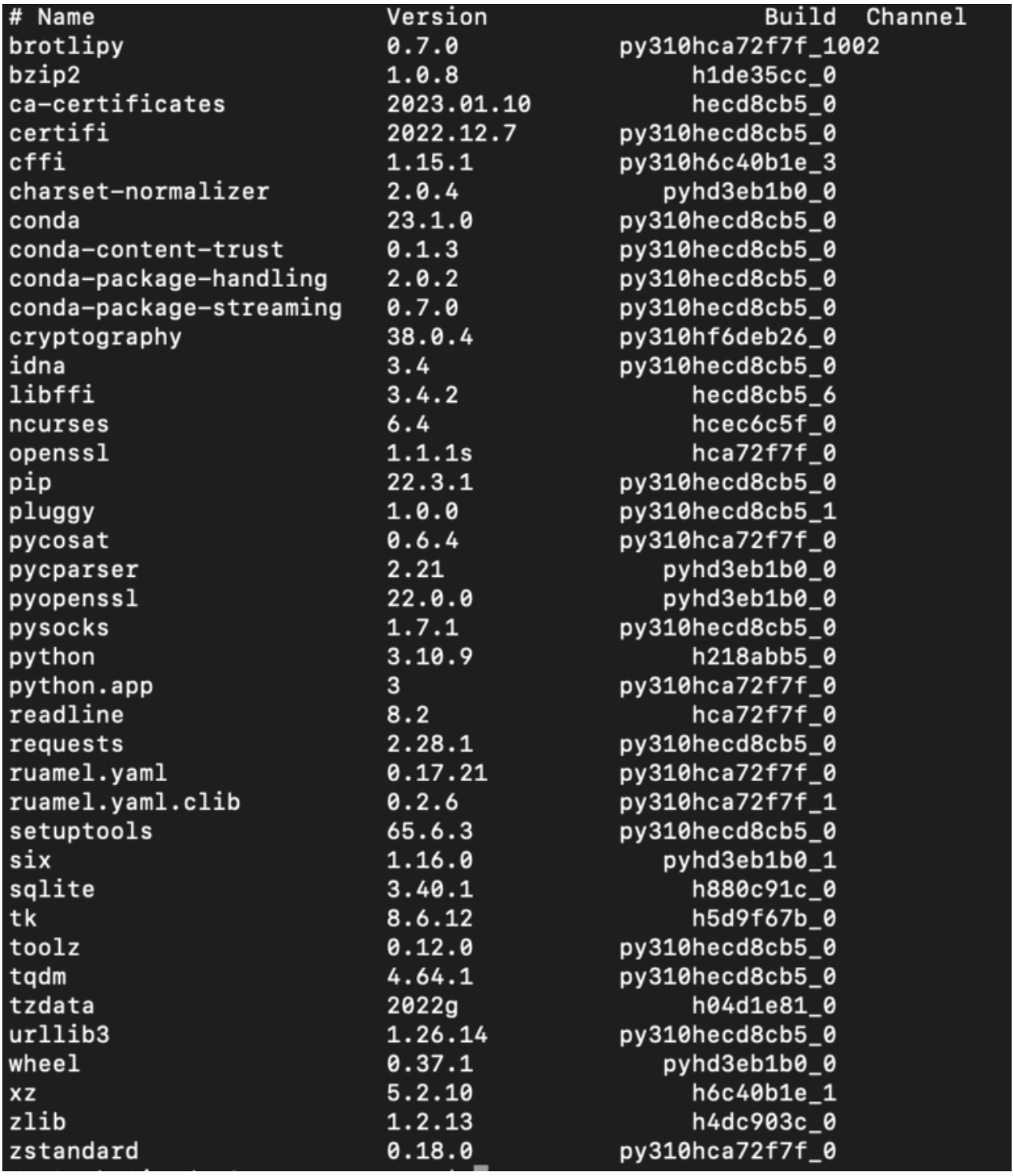
Per poder crear entorns Conda, en primer lloc, heu d'instal·lar MiniConda o Anaconda, depenent de les vostres preferències. Seguirem l'exemple utilitzant MiniConda i la instal·lació a Linux/Mac OS. En aquest enllaç podreu trobar les instruccions necessàries depenent del sistema operatiu que feu servir (https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html). Us heu de descarregar l'instal·lador depenent del vostre sistema operatiu i , en el cas de Linux o Mac OS, executar:

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

Per testar si la instal·lació ha finalitzat correctament podeu escriure al terminal

```
$ conda list
```

que retornarà el llistat de paquets instal·lats (figura 2). Recordeu que, si no es troba l'ordre conda, segurament és pel fet que no està en la ruta especificada d'ordres del vostre terminal, però el podreu trobar a la carpeta on heu instal·lat MiniConda o Anaconda, a 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 | |

FFigura 2. Sortida de l'ordre *conda list*.
Font: elaboració pròpia.

Conda té un entorn per defecte anomenat *base*. No és recomanable instal·lar paquets en aquest entorn. Si es vol iniciar un nou projecte, s'ha de crear nous entorns Conda.

# 1. Conda

## 1.4. Creació d'entorns Conda

És important donar un nom descriptiu a l'entorn per poder reconèixer el contingut. En un *pipeline* estàndard, podem utilitzar múltiples programes que poden ser reutilitzables en altres anàlisis. A causa d'aquesta redundància de processos, és recomanable crear un entorn per a cada eina i no crear entorns amb múltiples programes. D'aquesta manera, un entorn estarà definit pel *software* que compta mitjançant el nom, i la seva utilització serà més senzilla que si un mateix entorn conté múltiples paquets, ja que serà difícil determinar on es troba el programa que necessites en un moment determinat. De vegades també és recomanable no únicament especificar el programa en el nom de l'entorn, sinó també la seva versió.

En primer lloc, instal·larem un paquet molt usat a Python com és `numpy`. Per crear un entorn utilitzarem l'ordre `create` i n'especificarem la versió a instal·lar. Per saber quines versions són accessibles podem utilitzar l'ordre `search`:

```
$ conda search numpy
```

Això ens retornarà un llistat de totes les versions que estan disponibles. Si no especifiquem la versió del paquet a instal·lar, Conda intentarà instal·lar la versió més nova. Un cop seleccionem la versió que necessitem podem crear el nostre nou entorn:

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

Fent servir l'opció *-n* indiquem el nom que volem assignar al nou entorn.

Si voleu especificar des de quin canal volem instal·lar un paquet podem utilitzar l'opció `channel`:

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

Conda no instal·la únicament el paquet especificat, sinó també les seves dependències. En el cas de *numpy*, per exemple, no hem especificat que instal·lés Python, però com que és una dependència automàticament s'instal·la en l'entorn especificat.

Si sempre utilitzem un set de paquets, podem instal·lar-los conjuntament:

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

Un cop creats els entorns podem saber quins paquets s'han instal·lat utilitzant l'ordre `list`:

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

# 1. Conda

## 1.5. Activar entorns Conda

Un cop hem creat el nostre primer entorn Conda és hora d'utilitzar-lo. Per obtenir un llistat dels entorns que tenim en el nostre sistema utilitzarem l'ordre `env list`:

```
$ conda env list
```

Observareu que a més del nostre entorn `numpy1-23-5` i `basic-analisis` tenim l'entorn **base**, que com us havia comentat anteriorment està definit per defecte. És recomanable no instal·lar *software* en aquest entorn.

Per poder utilitzar un dels entorns l'hem d'activar:

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

Sabrem que l'entorn s'ha activat, ja que apareixerà el seu nom entre parèntesis davant de la línia d'ordres.

# 1. Conda

## 1.6. Desactivar entorns Conda en ús

Un cop finalitzats els processos requerits a l'entorn hem de tancar l'entorn. Per a això utilitzarem l'ordre `deactivate`:

```
$ conda deactivate
```

D'aquesta manera tornarem a l'estat inicial, abans de l'activació de l'entorn.

# 1. Conda

## 1.7. Instal·lar paquets dins d'un entorn ja creat

Els entorns Conda poden ser modificats *a posteriori* de la seva creació. Simplement, haurem d'activar l'entorn i, un cop dins, instal·lar un nou paquet. L'ordre `install` per defecte instal·la un paquet a l'entorn actiu.

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

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

En aquest cas s'instal·laria el paquet *pandas* a l'entorn `numpy1-23-5`. Si el nou paquet no és compatible amb la configuració de l'entorn actiu, saltarà un error i no podrà ser instal·lat.

Si volem especificar el canal, podem fer-ho:

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

# 1. Conda

## 1.8. Reanomenar entorns Conda

Si, quan instal·lem nous paquets, ho volem especificar en el nom de l'entorn, podem modificar el nom assignat:

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

# 1. Conda

## 1.9. Eliminar entorns Conda

Si un entorn ja no s'utilitza, és convenient eliminar-lo i, per fer-ho, utilitzarem l'ordre `remove`:

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

# 1. Conda

## 1.10. Compartir entorns

En projectes col·laboratius és freqüent la necessitat de reproduir tasques en els diferents centres. Per a això s'han de crear entorns agnòstics de sistema operatiu i plenament compatibles. Conda utilitza YAML (YAML *Ain't Markup Language*) com a arxius d'entorn que ens permetran importar i exportar entorns.

Per convenció, els arxius d'entorn de Conda s'anomenen *environment.yml.*

Si en el nostre directori de treball executem:

```
$ conda env create
```

automàticament Conda buscarà l'arxiu *environment.yml.* Si no el troba, saltarà un error. Si l'arxiu d'entorn té un altre nom, el podem especificar de la manera següent:

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

Podeu observar l'estructura d'un arxiu d'entorn de la figura 3, on es poden observar tres apartats:

- `name`: nom de l'entorn que es crearà si no s'especifica el contrari.
- `Channels`: canals a usar.
- `dependencies`: paquets a instal·lar amb relació `canal:paquet:versió`

```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. Exemple d'arxiu d'entorn YAML.
Font: elaboració pròpia.

Per poder generar un arxiu YAML d'un entorn que nosaltres hem creat, executarem:

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

Especifiquem el nom de l'arxiu amb l'opció `--file`

Per assegurar que l'entorn pot ser reproduïble independentment del sistema operatiu cal especificar l'opció `--from-history`:

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

# 2. Docker

## 2.1. Introducció

Un cop hem vist els entorns Conda, ens introduirem en una altra metodologia per controlar els processos utilitzats: els contenidors (*).

Els contenidors són sistemes de virtualització que contenen totes les eines necessàries per executar un *software*. Molt sovint es comparen les màquines virtuals amb els contenidors. La diferència més important és que les màquines virtuals virtualitzen tota una màquina fins a les capes de *hardware*, mentre que els contenidors únicament virtualitzen la capa de *software* damunt del sistema operatiu. Aquesta característica els fa més lleugers i fàcils de modificar.

Tot i que els contenidors no són una tecnologia nova, la seva aplicació de manera extensa va començar amb l'aparició de Docker el 2013 (consulta la caixa lateral). La popularització d'aquestes aplicacions va introduir la complexitat d'administrar centenars o milers de contenidors, i per això va aparèixer el que es coneix com la Orquestradors de contenidors. Tot i que al llarg del temps han aparegut diferents plataformes d'orquestració, fins i tot una del mateix Docker, com és Docker Swarm, Google va crear el 2014 Kubernetes, de codi obert, que s'ha convertit en el *software* preferit de moltes empreses i s'ha consolidat com un estàndard. Les plataformes d'orquestració s'encarreguen de reiniciar les aplicacions si fallen, d'equilibrar la càrrega de treball, d'escalar automàticament, d'implementar sense temps d'inactivitat, etc.

Tot i que en sistemes HPC (High Performance Computing) s'utilitza més Singularity, en aquest apartat de contenidors ens centrarem en com utilitzar Docker.

# 2. Docker

## 2.2. Contenidors Docker

En primer lloc, necessitarem instal·lar Docker al nostre ordinador. La forma més senzilla és utilitzant Docker Desktop (https://www.docker.com/products/docker-desktop/).

Per saber que tens Docker instal·lat correctament al terminal pots interrogar la seva versió:

```
$ docker -v
```

*L'output* t'indicarà la versió i el *build* que tens instal·lats.

I també:

```
$ docker system info
```

Si saltés algun error, s'hauria de comprovar si el procés d'instal·lació ha finalitzat correctament i que Docker Desktop estigui obert.

Per crear un contenidor, necessites el que es denominen imatges. Les imatges són els motlles dels contenidors, són les receptes/instruccions per crear els contenidors.

# 2. Docker

## 2.3. Descarregar contenidors creats

En primer lloc, mirarem si tenim alguna imatge en el nostre sistema. En teoria, si és la primera vegada que utilitzem Docker, ens hauria d'aparèixer una llista en blanc quan utilitzem:

```
$ docker image ls
```

Començarem pel contenidor més senzill i el descarregarem directament:

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

Deixarem que s'executi el procés de descàrrega i, si tot ha funcionat correctament, quan repetim l'ordre de llistat d'imatges ens hauria d'aparèixer la imatge `hello-world`.

La imatge de «hello-world» procedeix de Docker Hub (*), un repositori d'imatges.

Seguidament, executarem el contenidor que es generi a partir de la imatge `hello-world` mitjançant:

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

Un cop executat, rebreu un missatge de part de l'equip de Docker (figura 4).

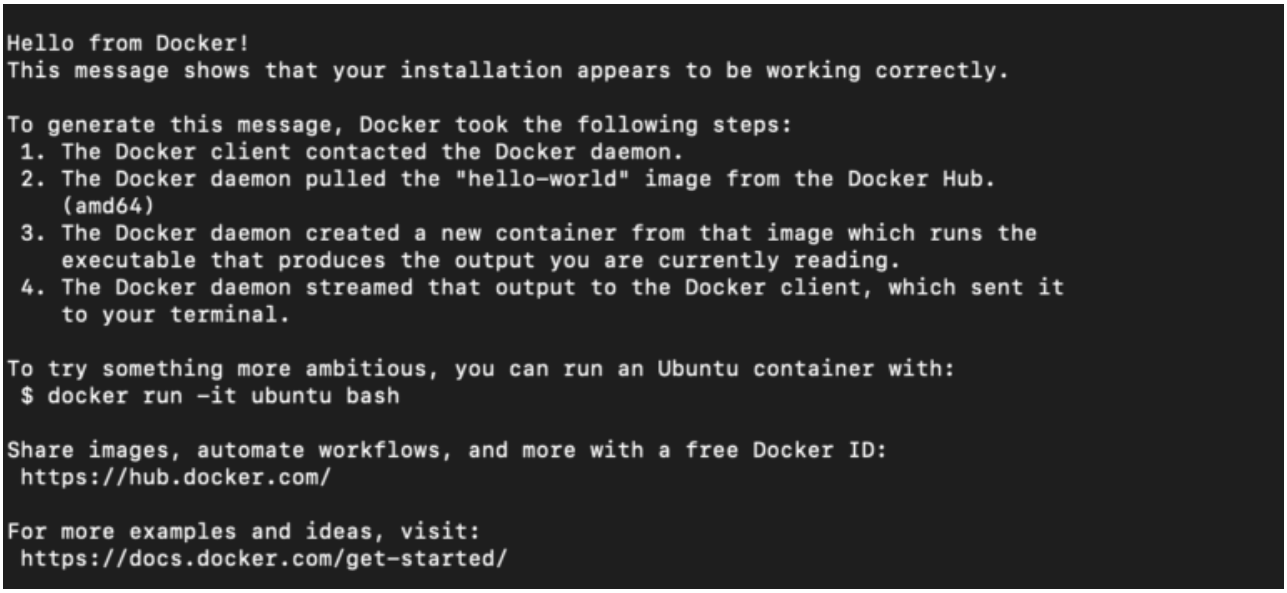



Figura 4. Missatge de benvinguda de l'equip de Docker en executar «hello-world».
Font: elaboració pròpia.

Quan s'executa un contenidor succeeixen tres processos:

- S'inicialitza el contenidor a partir de la imatge.
- S'executa l'acció preestablerta del contenidor, si existeix.
- Un cop l'acció ha finalitzat, el contenidor s'atura.

En el cas de l'exemple anterior, l'acció preestablerta era imprimir el missatge de benvinguda, però l'acció realitzada pot ser molt més complexa.

A més de poder executar les ordres predeterminades pel contenidor, també podem passar ordres o entrar en mode interactiu. Per provar aquestes opcions executarem el contenidor Alpine que conté una distribució d'Ubuntu molt simple:

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

En aquesta ordre utilitzem l'opció `it` per ser interactiu i `sh` ens especifica que el terminal que volem utilitzar és *bash*.

Veurem que la línia d'ordres canvia a:

```
/ #
```

Podrem comprovar que ara ens trobem dins d'Alpine i quan executem:

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

ens mostrarà la versió d'Alpine que ens hem baixat.

D'aquesta manera, un cop executat el contenidor no s'ha finalitzat com havíem observat anteriorment, sinó que es manté actiu i respon a les ordres que hi introduïm.

Per poder sortir del contenidor i finalitzar la seva execució podem introduir-hi l'ordre `exit`.

# 2. Docker

## 2.4. Eliminar imatges i contenidors Docker

Sempre és recomanable mantenir una bona gestió d'imatges i contenidors, ja que, si no s'utilitzen, van ocupant espai de manera innecessària. Així, aprendrem a eliminar imatges i contenidors.

Per poder eliminar imatges, necessitarem saber la Image ID. Per a això, utilitzarem la següent ordre:

```
$ docker image ls
```

Ens apareixerà un *output* similar al següent:

### Taula 1.

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

**Font: elaboració pròpia.**

Així, per eliminar una imatge simplement s'ha d'especificar la Image ID:

```
$ docker image rm feb5d9fea6a5
```

O emprant el seu nom:

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

Moltes vegades ens podem trobar amb un missatge d'error:

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

Això ens està indicant que hi ha un contenidor que encara està actiu, que encara no ha estat netejat o que està o ha fet ús d'aquesta imatge. Per això, primer hem d'eliminar els contenidors que estan impedint l'eliminació d'aquesta imatge.

Primer llistem els contenidors actius:

```
$ docker container ls
```

O, de manera similar, podem utilitzar:

```
$ docker ps
```

És possible que no ens retorni cap contenidor actiu. En realitat, el missatge d'error previ ens indicava que el contenidor s'havia aturat; per tant, té sentit que no trobem el contenidor en aquest llistat. Així, necessitarem veure tant els contenidors actius com els que recentment s'han aturat:

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

Aquí sí que podem veure el contenidor anteriorment especificat com a enllaçat a «hello-world»:

### Taula 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 |

**Font: elaboració pròpia.**

Per evitar que es vagin acumulant els contenidors, es pot afegir a l'ordre `run` una opció d'eliminar automàticament un cop finalitzat:

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

Si no hem especificat l'opció `rm`, l'hem d'eliminar manualment utilitzant com a referència el Container ID:

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

Si s'elimina correctament, ens retornarà el Container ID al terminal. Es pot esborrar més d'un Container ID simultàniament.

Si tenim diversos contenidors associats a una imatge i volem esborrar-los tots utilitzant un patró podem:

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

En aquest cas, utilitzem l'ordre `xargs` per controlar una llista d'arguments per poder ser eliminats mitjançant `rm` de Docker.

Ara veurem que, quan llistem els contenidors, ja no apareix el contenidor i, si provem d'eliminar la imatge com prèviament havíem intentat, ho farà sense problemes.

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

```
$ docker image ls
```

Si es volen eliminar tots els contenidors existents, podem utilitzar:

```
$ docker container prune
```

D'aquesta manera tots els contenidors existents seran eliminats i, si volem tornar a reconnectar-los, haurem d'iniciar-los de nou.

# 2. Docker

## 2.5. Parar contenidors a Docker

Iniciar o aturar un contenidor no és el mateix que iniciar o aturar un procés. Per finalitzar un contenidor, Docker proporciona les ordres `stop` i `kill`. Encara que sembli que totes dues ordres facin el mateix, la seva execució és, internament, diferent. Per aturar un procés, podem utilitzar tant el ContainerId com el nom del contenidor.

L'ordre `stop` atura el contenidor d'una manera menys agressiva que `kill`. Això és degut al tipus de senyal que s'envia al contenidor. L'ordre `stop` envia un senyal SIGTERM, que es pot bloquejar o parar, mentre que `kill` envia un senyal SIGKILL que no es pot gestionar. Si en un temps prudencial l'ordre `stop` no ha aturat el contenidor, Docker envia automàticament un senyal SIGKILL. Per defecte, aquest període són 10 segons, però si volem modificar-lo podem fer ús de l'opció *-t* expressada en segons. Així:

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

Docker pararia el procés mitjançant una SIGKILL després de 77 segons.

Com hem vist en l'apartat anterior també podríem utilitzar l'ordre `rm` per finalitzar un contenidor. La diferència rau en el fet que l'ordre `rm` elimina el procés i no el podem visualitzar en la llista `docker ps -a`, mentre que, parant el contenidor, el podem mantenir i reutilitzar posteriorment.

Una altra opció interessant és pausar el contenidor. Si aturem un contenidor, alliberem els recursos de memòria i CPU; si el pausem, només alliberem CPU.

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

# 2. Docker

## 2.6. Buscar contenidors a Docker Hub

Un recurs que ja hem utilitzat per al contenidor «hello-world» és Docker Hub. En aquest repositori podem trobar molts contenidors ja creats. Molts d'ells ja han estat construïts i testats pels mateixos desenvolupadors del *software* que està buscant. Com a exemple anirem a la pàgina del *variant caller* GATK (https://hub.docker.com/r/broadinstitute/gatk). Aquí trobareu les instruccions per poder-vos-el descarregar. Si en voleu una versió determinada, en baixar la imatge necessiteu especificar-la. Per a això s'utilitzen els *tags* igual que com vam veure en l'apartat de Conda. A la pàgina de GATK teniu una pestanya on teniu les diferents versions indicades per *tags*.

Per indicar quina versió volem utilitzar ho indicarem amb els «`:`»:

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

Si no especifiquem la versió, es baixarà la imatge més recent denominada `latest`.

Com podeu apreciar en l'ordre `pull` que hem utilitzat per baixar GATK davant del nom de `gatk`, tenim el nom de la institució que l'ha creat i l'ha fet públic. En aquest cas, el Broad Institute. Si aquest nom previ no apareix, indica que els desenvolupadors són el mateix equip de Docker.

És important tenir en compte que qualsevol persona pot crear un compte a Docker Hub i, per tant, és important ser prudent quan descarreguem *software* de fonts no contrastades. Haureu de procurar baixar imatges directament dels desenvolupadors o de comunitats establertes. Docker manté una sèrie d'imatges referenciades com Docker Official Images, les quals han estat analitzades i proporcionen repositoris bàsics per a la comunitat, com ara Ubuntu o Centos.

Una altra opció per buscar imatges és l'ordre `search`. En aquest cas, quan utilitzem

```
$ docker search GATK
```

obtenim un llistat d'imatges on trobaríem repositoris amb 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 recerca d'imatges que continguin GATK.
Font: elaboració pròpia.

# 2. Docker

## 2.7. Crear la teva pròpia imatge Docker

Si la nostra cerca a Docker Hub ha estat infructuosa o requerim una imatge específica per a les nostres necessitats, la solució és crear-la nosaltres mateixos.

En primer lloc, instal·larem *software* en una sessió interactiva. Amb aquesta finalitat, utilitzarem la imatge d'Alpine que prèviament havíem creat i intentarem instal·lar el paquet de *python numpy*. Alpine no és la distribució en la qual voldríem basar-nos per desenvolupar els nostres projectes; segurament Ubuntu o Debian siguin unes eleccions més apropiades. Abans de començar una imatge és important que tinguem ben clar què necessitarem instal·lar, quins són els nostres requisits i, finalment, escollir la distribució més apropiada. També és una bona pràctica no instal·lar massa programes en cada imatge perquè seran més difícils de crear i de mantenir. És la mateixa filosofia que aplicàvem als entorns Conda.

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

Seguidament comprovarem si tenim Python instal·lat:

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

I podem observar que no està instal·lat.

Alpine té un gestor de paquets anomenat *Alpine Package Keeper* (apk) per instal·lar *software.* Depenent de la distribució de Linux que tinguem instal·lat, podríem utilitzar `apt-get, yum, zypper`...

En primer lloc, instal·larem Python i altres paquets necessaris. Posteriorment, afegirem el paquet *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 ara entrem a Python podem comprovar que s'ha instal·lat correctament el paquet *numpy*.

Un cop sortim d'aquest contenidor, els canvis no s'hauran guardat. Per crear la nostra pròpia imatge, el més recomanable és utilitzar un Dockerfile.

Un Dockerfile és un arxiu de text amb l'estructura mínima següent:

- FROM <Imatge preexistent>
- RUN <Ordres per instal·lar des de la línia d'ordres>
- CMD <Ordres que s'han de córrer per defecte>

Les instruccions que es llancen per defecte (CMD) tenen una estructura definida. Només hi pot haver una línia de CMD al Dockerfile; si n'hi hagués més d'una, s'executaria l'última.

Així, si volguéssim reproduir el contenidor anterior, hauríem de crear un arxiu anomenat *Dockerfile* de la següent 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
```

Un cop tenim el Dockerfile definit crearem la imatge:

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

L'opció `-t` ens indica el nom que li volem donar a la nostra imatge. Seguint la nomenclatura anteriorment esmentada indiquem l'autor i el paquet. El « `.` » ens indica que Dockerfile és a la mateixa carpeta on estem executant les ordres; hem d'especificar la ruta al directori del Dockerfile.

Ara podreu veure la nova imatge creada:

```
$ docker image ls
```

## Taula 3.

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

Si executem el contenidor Docker, ens retornarà la versió de Python que tenim instal·lada a la imatge:

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

Si entréssim ordres en la línia d'execució del contenidor, aquestes són les que s'utilitzarien en detriment de les que estiguessin al Dockerfile. Així:

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

Ens indicarà la localització de Python3 en el *filesystem* del contenidor.

# 2. Docker

## 2.8. Reanomenar imatges

En realitat, el que crearem és una còpia d'una imatge amb un nom nou. Per això, farem:

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

Si la imatge és susceptible d'evolucionar, és important mantenir una numeració en els seus *tags*, com hem vist a Docker Hub:

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

Si ara mireu les imatges que teniu creades, podreu veure que a la columna *tag* tindreu la versió que hem creat ara i la `latest`, que és per defecte, si no se n'especifica un altre.

## Taula 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 |

**Font: elaboració pròpia.**

# 2. Docker

## 2.9. Execució d'*scripts*

Com hem vist, la creació d'un contenidor és relativament senzill de fer, però, depenent de les nostres necessitats, és possible que els exemples anteriors es quedin curts. Per això hem d'ampliar els coneixements necessaris per crear imatges més complexes.

En primer lloc, utilitzarem el contenidor amb Alpine que hem creat anteriorment per executar un *script* a Python.

Si, directament, passem els paràmetres quan iniciem el contenidor, ens retornarà un error perquè l'arxiu no es troba en el seu sistema:

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

Sent `num.py`:

```
import numpy as np

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

print (a)
```

L'error és:

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

Perquè aquesta ordre funcioni, necessitarem enllaçar els dos sistemes, el nostre propi i el del contenidor Docker. Per això, utilitzarem l'ordre `mount` especificant on és el nostre arxiu i posicionant-lo en el sistema del contenidor.

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

Existeixen diferents tipus de *mount*. En aquest cas, utilitzarem el mode *bind*, que és el que ens interessa per a aquest exemple. Amb el *source*, especifiquem el directori on s'ubica *l'script*, podem utilitzar la variable ${PWD} si llancem el contenidor des del mateix directori on hi ha l'*script* i també podem utilitzar *target*, que especifica on localitzarem el contenidor. És important que, quan executem Python3, especifiquem la carpeta on localitzem l'*script*, en aquest cas /temp.

D'aquesta manera ens retornarà:

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

# 2. Docker

## 2.10. Volums Docker

En l'exemple anterior hem utilitzat `mount` per unir els dos sistemes. `Mount` depèn del sistema en el qual s'executa, mentre que els **volums** són nadius de Docker. Els volums es poden compartir entre contenidors i persisteixen més que els contenidors.

En l'exemple anterior, podríem passar un volum al contenidor amb l'opció `-v`:

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

Definim la carpeta local $(PWD) i on es localitza al contenidor /temp.

Per crear un volum amb nom `data_set` i assignar una carpeta local:

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

En aquest cas, podrem llançar:

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

Podem visualitzar els diferents volums:

```
$ docker volume ls
```

... inspeccionar-los:

```
$ docker volume inspect data_set
```

... i eliminar-los:

```
$ docker volume rm data_set
```

# 2. Docker

## 2.11. Integrar dades al contenidor Docker

Moltes vegades necessitarem utilitzar un *input* de manera sistemàtica en un contenidor. Per exemple, si volem fer una *variant call* utilitzant GATK, els genomes de referència sempre seran els mateixos i, tal vegada, és interessant guardar-los dins del contenidor per assegurar-nos la reproductibilitat dels resultats per tal que no depengui de la referència utilitzada. Per a un cas més simple introduirem el nostre *script num.py* al contenidor. Per això crearem una nova imatge modificant el Dockerfile.

Hi afegirem una nova línia:

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

Estarem fent una còpia de l'*script* al *home* del contenidor. Fem una altra imatge:

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

Si ara entrem de forma interactiva dins del contenidor i llistem els arxius dins el *home,* trobarem l'arxiu *num.py*:

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

És important com ordenem les ordres al Dockerfile. És recomanable fer les ordres COPY després dels RUN, ja que, quan fem el *build*, Docker va per ordre i, si en algun moment volem afegir un altre *script* en lloc de *num.py*, si el COPY es troba al final del procés, Docker utilitzarà els RUN que té a la memòria no haurà de construir la imatge de zero, i el procés serà molt més ràpid. Docker va línia a línia i, si aquesta capa o conjunt de capes ja la té en memòria *caché,* agilitza el procés perquè no ha de reinstal·lar-les.

Si les dades que volem introduir en la nostra imatge són a internet, podem copiar-les directament fent servir RUN. Podríem afegir aquestes línies al Dockerfile com a exemple:

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

Per defecte, l'arxiu es copia en el *root* del sistema del contenidor. Si el volguéssim moure a una altra localització, podríem especificar-la:

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

... i la copiaria a la nostra carpeta /home.

Ara que sabem com introduir l'*script* dins del contenidor, podem crear un nou contenidor que corri *l'script* automàticament. Hem de substituir CMD del Dockerfile a:

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

D'aquesta manera creem una nova imatge:

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

... i executem directament:

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

... i ens retorna el resultat de l'arxiu *num.py*.

Si volem introduir un nou *script* de Python (*atcg.py*) on el seu resultat depengui d'un argument:

```
import sys
```

```
filename = sys.argv[1]

filenumb = len(filename)

print (filenumb)
```

... i el copiarem mitjançant Dockerfile:

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

... i crearem una nova imatge:

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

Si el fem córrer directament, tindrem el resultat del CMD (en el nostre cas, la versió de Python); mentre que, si afegim arguments a l'ordre `run`, se sobreescriu CMD i obtenim el resultat del nou *script*:

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

Si volguéssim tenir aquest nou *script* com ordres per defecte, al Dockerfile hauríem de canviar la línia de CMD pels valors per defecte i crear una nova línia anomenada ENTRYPOINT per localitzar l'*script*:

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

CMD ["ACTG"]
```

També podem afegir abans de ENTRYPOINT i CMD una entrada per definir el directori de treball:

```
WORKDIR /home
```

D'aquesta manera, si creem una nova imatge `uoc/alpine-entry` i executem el contenidor directament:

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

Ens retornarà «4», la longitud de la línia «ACTG» ubicada per defecte al Dockerfile. Si ara, al final de l'ordre *run,* hi afegim un altre *input*, aquest substituirà el de CMD:

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

Retornarà «8».

Podem observar la relació entre ENTRYPOINT i CMD en la següent taula.

## Taula 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 |

**Font: elaboració pròpia.**

# Resum

En aquest apartat heu après com crear i utilitzar entorns Conda i contenidors Docker. Fer servir aquest tipus d'eines és molt important en la reproductibilitat de processos i resultats. Tant si hem de compartir el nostre codi amb altres persones com si hem de repetir el mateix procediment amb altres mostres en un futur, és crucial poder tenir un seguiment de les eines utilitzades. Aquests procediments milloren la qualitat de la recerca publicada i faciliten el procés de revisió per part d'investigadors externs. Actualment, pràcticament tots els programes depenen d'altres paquets de programes, que evolucionen independentment. La modificació d'una dependència pot derivar en diferents resultats d'un programa o, fins i tot, al seu mal funcionament. Per això, és rellevant mantenir els programes i les seves dependències tal com es van utilitzar per poder assegurar la correcta reproductibilitat. La utilització de petits entorns i contenidors facilita el manteniment i el replicat de processos. A més, tenir els programes aïllats del sistema general permet la seva eliminació i actualització d'una manera més senzilla i neta. Alguns programes poden necessitar certes dependències incompatibles amb altres ja instal·lades en el sistema i, per això, la creació d'entorns o la utilització de contenidors facilita el fet de treballar en diferents configuracions dins el mateix sistema.

# Activitats

## Conda

1. Creeu un nou entorn de nom «UOC-bioinformatics» que contingui Python 2.7.

2. Instal·leu en aquest entorn ja creat R i el paquet Tidyverse en la seva última versió.

3. Enumereu els paquets instal·lats en l'entorn de l'activitat anterior.

4. Instal·leu el paquet Scipy en un entorn nou a través del canal Bioconda.

5. Creeu un arxiu «environment.yml» d'aquest últim entorn.

## Docker

1. Baixeu-vos la imatge d'Ubuntu de Docker Hub, entreu al contenidor de forma interactiva i determineu la versió d'Ubuntu descarregada.

2. Repetiu el procediment anterior amb la versió prèvia a la *latest*.

3. Creeu un Dockerfile on afegiu l'última versió de R al sistema operatiu Centos.

4. Creeu un volum Docker on pugueu passar al contenidor un *script* en *bash* que retorni «Hello World».

5. Repetiu el procediment anterior modificant el missatge a través del terminal en córrer el contenidor i que retorni «Hi, I'm back».

# Bibliografia

**Björn Grüning i altres** (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 i altres** (2020). FAIR Computational Workflows. *Data Intelligence,* 2, p. 108-121.

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

**Serghei Mangul i altres** (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 i altres** (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 i altres** (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 i altres** (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 i altres** (2015). Big Data: Astronomical or Genomical? *PLoS Biol*ogy, *13*, e1002195.