Workflows

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

Los *pipelines* tradicionales están muy ligados a las infraestructuras de computación locales donde se ejecutan. Estos no tienen la capacidad de resumir un proceso que se haya parado, tienen poca documentación, no cuentan con una trazabilidad de los parámetros y versiones de paquetes utilizados y requieren de instalación manual, lo cual impide una fácil distribución de este. Para poder solucionar estos inconvenientes se han creado los Workflow Managers. Estos permiten la utilización de *pipelines* de análisis complejos en distintos entornos de computación asegurando la máxima reproducibilidad de los procesos ejecutados.

Varios Workflow Managers se han desarrollado específicamente para los campos de investigación y salud integrando entornos, contenedores y computación en la nube.

Hay cinco características que hacen a los Workflow Managers herramientas de gran utilidad:

1. Reproducibilidad. La utilización de entornos y contenedores asegura una apropiada reproducibilidad de los procesos ejecutados.

2. Portabilidad. Es una de las grandes ventajas de la utilización de Workflow Managers, ya que crea los flujos de trabajo necesarios para poderse exportar a cualquier entorno computacional. Muchos de ellos permiten la fácil migración a distintos entornos, inclusive de alta computación y servicios en la nube. Es más, es posible la interacción directa con orquestadores como Kubernetes o DockerSwarm.

3. Escalabilidad. Ser capaz de manejar y analizar datos con una complejidad creciente es cada vez más común. En este sentido hay dos aspectos que deben tenerse en cuenta: el manejo eficiente de los recursos y ser capaz de utilizar datos más complejos y de mayor tamaño. La mayoría de Workflow Managers implementan la paralelización en diversos pasos, sea mediante gestor de colas o *scheduling* estática o adaptativa. La paralelización puede producirse a nivel de datos, procesos o *pipelines*. Una asignación dinámica de los recursos permite que los procesos más intensivos no se vean afectados respecto a los que no requieren tantos. Este balanceo minimiza cuellos de botella y reduce los tiempos de computación. Los recursos pueden asignarse específicamente para cada paso del flujo de trabajo.

4. Robustez. Muchos *pipelines* requieren de procesos complejos y de larga duración. En el posible evento de la interrupción del *pipeline* en algún proceso debido a un error, sea programático o por la ausencia de un *input* requerido, los Workflow Managers son capaces de resumir el proceso desde el lugar donde hubo el último paso correcto, resultando en el ahorro en la utilización de recursos y tiempo. Este proceso se consigue mediante la producción de archivos y resultados intermedios, siendo comparados con los resultados esperados. Este proceso genera un aumento en las necesidades de almacenamiento, pero comporta una ventaja sustancial en el caso de tener la necesidad de una reentrada en el *pipeline*.

5. Modularidad. La compartimentación de los procesos permite un gran dinamismo en la actualización de ciertos pasos del proceso, así como de la introducción de puntos de control para cada etapa. La modularidad también permite la reutilización de un proceso en varios *pipelines* simultáneamente.

Finalmente, indicar que algunos Workflow Managers también tienen recursos para aumentar la seguridad en la ejecución de los procesos, como la validación del origen de los datos o utilizar autenticación de usuarios.

Objetivos

1. Ofrecer una visión general de los Workflow Managers.

2. Introducir a los lectores a Nextflow.

1. Diferentes tipos de *workflows*

La implementación de un *pipeline* en un Workflow Manager requiere de la definición precisa de dónde se extraen los datos iniciales (*input*), cuál es el flujo entre las distintas herramientas y cuál es el resultado final (*output*). Para poder estructurar los distintos parámetros se utilizan lenguajes de dominio específico (DSL). La utilización de DSLs aumenta la portabilidad y la escalabilidad. Nextflow y Snakemake son dos de los ejemplos más populares de *workflows* que utilizan DSLs propios en el ámbito de la bioinformática. La diversidad de DSLs con su propia estructuración y nomenclatura propicia la reducción de interoperabilidad entre los distintos *workflows*. Para mitigar esta disparidad se han creado unas especificaciones que añaden un nivel mayor de abstracción, permitiendo un marco común entre los distintos *workflows*. Ejemplos de especificaciones para *workflows* son el Common Workflow Language (CWL) y Workflow Description Language (WDL). CWL prioriza la portabilidad y la reproducibilidad, mientras que WDL tiene como principal objetivo reducir la curva de aprendizaje mediante un lenguaje más comprensible. CWL define los *pipelines* utilizando ficheros YAML, mientras que WDL utiliza sus propios ficheros descriptivos. Algunos Workflow Managers, como *cwltool* o Cromwell, se han creado basándose en estas especificaciones, mientras que otros como Snakemake implementan opciones de exportación a estas especificaciones.

La utilización de Workflow Managers para la creación de *pipelines* bioinformáticos está creciendo en popularidad. Distintos *workflows* están disponibles para la comunidad variando en su flexibilidad y facilidad de uso (https://github.com/pditommaso/awesome-pipeline). Mientras que algunos *workflows* necesitan de un conocimiento avanzado de programación, otros tal que Galaxy, KNIME o BioWorkflow implementan una interfaz gráfica para facilitar su utilización. Aunque el desarrollo de sistemas de *workflow* se inició a principios de los años noventa, ha sido la capacidad de correr *pipeline*s en clústeres de computación o en la nube lo que ha facilitado su uso más generalizado.

2. Nextflow

2.1. Introducción

Como ejemplo de Workflow Manager en esta sección utilizaremos uno de los programas más extendidos con una comunidad creciente y con mayor soporte como es Nextflow.

En primer lugar, debemos instalar Nextflow siguiendo las instrucciones directamente de su página web (https://www.nextflow.io/).

Nextflow combina tres elementos: un entorno de ejecución, un programa específico para lanzar otros programas y un DSL específico. La estrategia que implementa Nextflow en la creación de *pipelines* es la de crear módulos para cada paso en el *pipeline* y vehicularlos e interconectarlos a través de canales. Una de las características más importantes de Nextflow es la reutilización de estos módulos para distintos estadios de un mismo *pipeline* o entre distintos *pipelines*. La ejecución de los distintos módulos no es lineal, sino que depende de la disponibilidad de los *inputs* para ejecutarse. Si dos módulos independientes requieren los mismos *inputs* o *inputs* independientes se pueden ejecutar simultáneamente para después poder converger en otro posteriormente si se requiere.

2. Nextflow

2.2. DSL2

Nextflow utiliza como DSL una extensión del lenguaje de programación Groovy. A partir de la versión 20.07.1 de Nextflow se actualizó la sintaxis del DSL original creando DSL2. A lo largo de esta sección utilizaremos esta nueva implementación, DSL2. Esta actualización, entre otras muchas características, permite la utilización de más de un *script* para definir el *workflow*, a diferencia de DSL1, en el que se acumulaba todo en un único *script*. En el código de Nextflow se debe especificar la utilización de DSL2, ya que por defecto utiliza DSL1. Para ello se indicará al inicio del *script*:

```
nextflow.enable.dsl=2
```

2. Nextflow

2.3. Estructura de *workflows*

Los *workflows* creados para Nextflow contienen tres partes diferenciadas: procesos, canales y *workflows*. Un proceso ejecuta una tarea. Cada proceso es independiente del otro y puede tener más de un canal de entrada y de salida. Un canal es un sistema de colas asíncrono que permite el flujo de información entre procesos (figura 1). Para juntar los distintos procesos y su flujo de ejecución (canales) se resume en un apartado específico en el *script* que se denomina *workflow*.

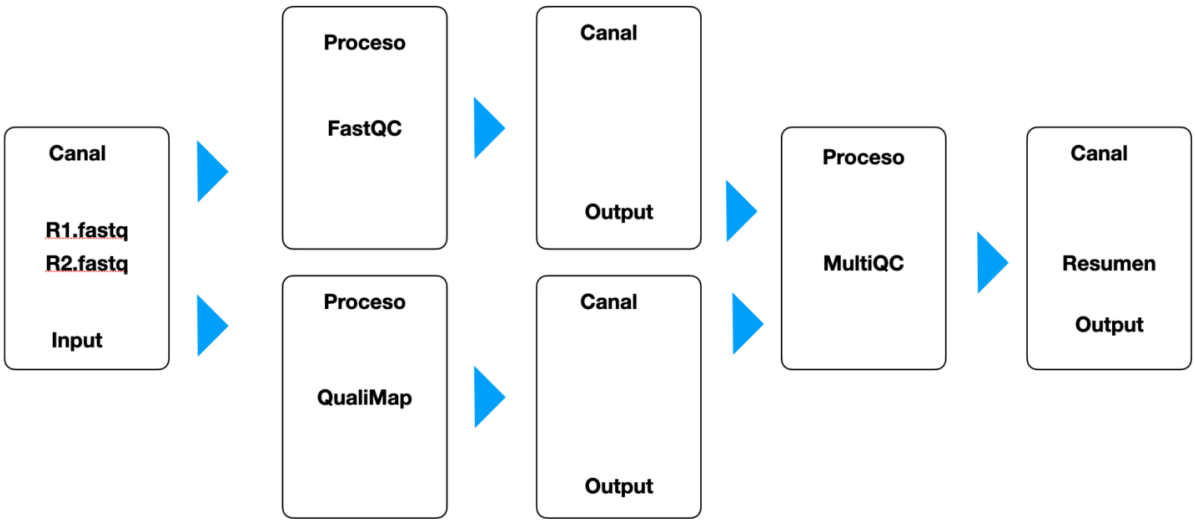

Figura 1. Ejemplo de esquema de *workflow* en Nextflow.
Fuente: elaboración propia.

Nextflow diferencia los comandos que se ejecutaran dentro de un proceso y quién será el encargado de ejecutarlos. Esto permite tener un marco general que describa qué se quiere hacer independientemente de las herramientas que se utilizaran para ejecutarlo. Esta estructura permite lanzar un proceso en distintos entornos computacionales variando simplemente un fichero de configuración, el cual define los ejecutores específicos del entorno donde uno se encuentre.

2. Nextflow

2.4. «Hello world»

Empezaremos por un *script* sencillo (helloworld.nf) donde imprimimos una frase, por ejemplo «Hello world».

```
nextflow.enable.dsl=2


params.str = 'Hello world!'


process printStr {

 output:

    stdout


  """

echo '${params.str}'

  """

}



workflow {

 printStr | view ( )

}
```

- En primer lugar, al *script* `nf` le indicaremos que utilizamos `DSL2`.
- Crearemos un parámetro al cual nombraremos str y le asignaremos la frase «Hello world!». El nombre del parámetro está precedido por un punto y la palabra `params`. En esta sección, al ser un único parámetro no hace falta crear un canal, ya que Nextflow le asigna directamente un canal *value*. Hay distintos tipos de canales, como veremos más adelante, y el canal *value* es el más simple.
- El siguiente segmento define un proceso al cual nombraremos printStr. La nomenclatura de los procesos es *process* seguido del nombre que le asignamos. Los procesos están formados de tres partes: *input*, *output* y *script*. En este caso, le indicamos que el *output* sea el estándar, y al ser un valor simple tampoco hace falta que definamos un canal específicamente. Dentro del proceso definiremos qué queremos hacer. Imprimimos el parámetro str.
- Finalmente asignamos la última sección al flujo del *workflow*.

Una vez creado el *script* lo ejecutaremos:

```
$ nextflow run helloworld.nf
```

El resultado se muestra en la figura 2.

```
N E X T F L O W  ~  version 20.10.0
Launching `helloworld.nf` [boring_pare] - revision: 6452698d90
executor >  local (1)
[e7/6bcd02] process > printStr [100%] 1 of 1 ✔
Hello world!
```

Figura 2. Ejecución del *script* helloworld.nf.
Fuente: elaboración propia.

Como se puede observar, se ha ejecutado un proceso `printStr` y se visualiza su resultado. Si quisiéramos modificar un parámetro, en este caso `str`, desde la línea de comandos ejecutaremos:

```
$ nextflow run helloworld.nf --str 'Bye Bye World'
```

Y obtendremos el nuevo resultado.

Ahora le añadiremos un segundo proceso: poner todas las letras en mayúscula. Para ello añadiremos el proceso *allToUpper*.

```
nextflow.enable.dsl=2

params.str = 'Hello world!'

process printStr {

  output:

   path 'test.txt'


  """

echo '${params.str}' > test.txt

  """

}

process allToUpper {

 input:

   path x

  output:

    stdout

  """

cat $x | tr '[a-z]' '[A-Z]'

  """

}


workflow {

 printStr | allToUpper | view ( )

}
```

Y el resultado lo visualizamos en la figura 3.

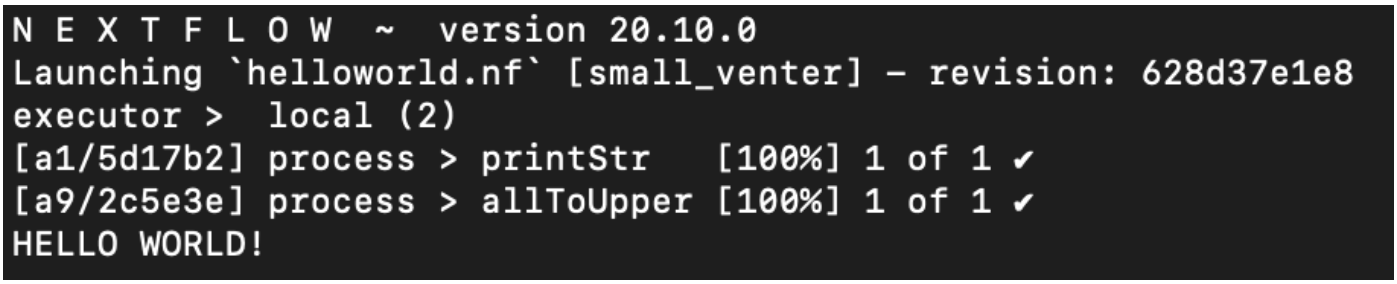

Figura 3. *Output* de dos procesos en Nextflow.
Fuente: elaboración propia.

Como podéis comprobar en la sección *workflow*, primero ejecutamos `printStr`, posteriormente `allToUpper` y finalmente lo visualizamos mediante `view( ). view` es un operador que veremos más adelante.

Seguidamente sustituimos el comando de `allToUpper`:

```
  cat $x | tr '[a-z]' '[A-Z]'
```

por:

```
  rev $x
```

y volvemos a ejecutar el *script*. En este caso, queremos imprimir «Hello world!» del revés. Como el primer paso ya lo habíamos lanzado anteriormente, podemos resumir la ejecución mediante la opción *resume*:

```
  $ nextflow run helloworld.nf -resume
```

y nos generará el *output* de la figura 4.

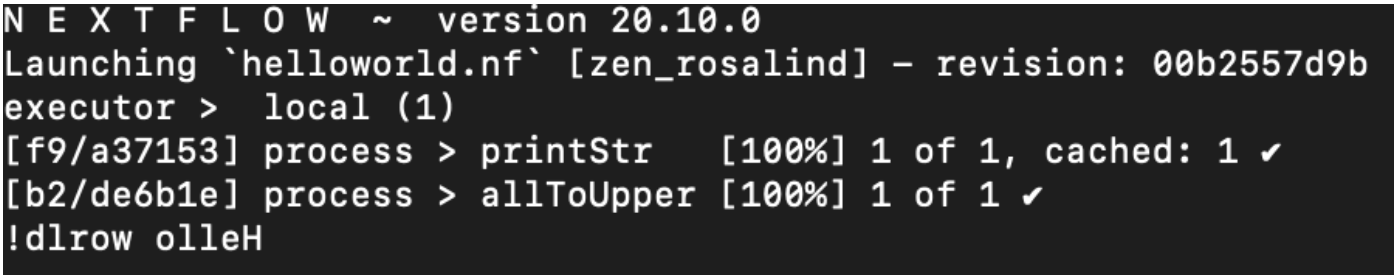

Figura 4. Resumir la ejecución del *script* helloworld.nf.
Fuente: elaboración propia.

Como podéis ver, el primer paso del *script* no ha sido calculado de nuevo, sino que se ha utilizado el proceso ya generado anteriormente (*cached*). De esta forma, si en algún paso del *pipeline* ha habido algún error que ha parado el proceso, se puede resumir previa subsanación del problema.

2. Nextflow

2.5. Canales en Nextflow

Como se ha comentado anteriormente, los canales dirigen el flujo de información a través de los distintos procesos. Para crear explícitamente un canal se debe utilizar un método de Channel Factory proporcionado por Nextflow.

En Nextflow hay dos tipos de canales:

- *Queue channels*: son canales asincrónicos, unidireccionales y FIFO (*first-in-first-out*).

Algunos ejemplos son:

```
- of:

      ch = Channel.of(1, 3, 5, 7)


- fromPath:

      file_ch = Channel.fromPath('test/*.txt')


- fromFilePairs:

      fastq = Channel.fromFilePairs('/my/data/SRR*_{1,2}.fastq')
```

- *Value channels* (*singleton channel*): son muy similares a los *queue channels*, pero solo admiten un valor.

```
          value:
  pi = Channel.value('3.1416')
```

Para entender un poco mejor el funcionamiento de los canales crearemos un *script* llamado *orden.nf* con un canal `value` y otro `of` e imprimimos su resultado.

```
nextflow.enable.dsl=2
pi = Channel.value(3.1416)
queue_ch = Channel.of( 1, 3, 5, 7 )

process ordenE {
    input:
      val x
      val y
    output:
      stdout
"""
     echo $x $y
"""
}
workflow{
ordenE(pi,queue_ch) | view( )
}
```

Como podéis observar, en el apartado *workflow* estamos especificando los *inputs* del proceso ordenE. En el resultado de este *script* (figura 5), el orden de aparición de los valores del canal of no es el mismo que le hemos indicado. Si repetís la ejecución seguramente os saldrá otro resultado distinto. Se producen cuatro procesos de forma no correlativa.

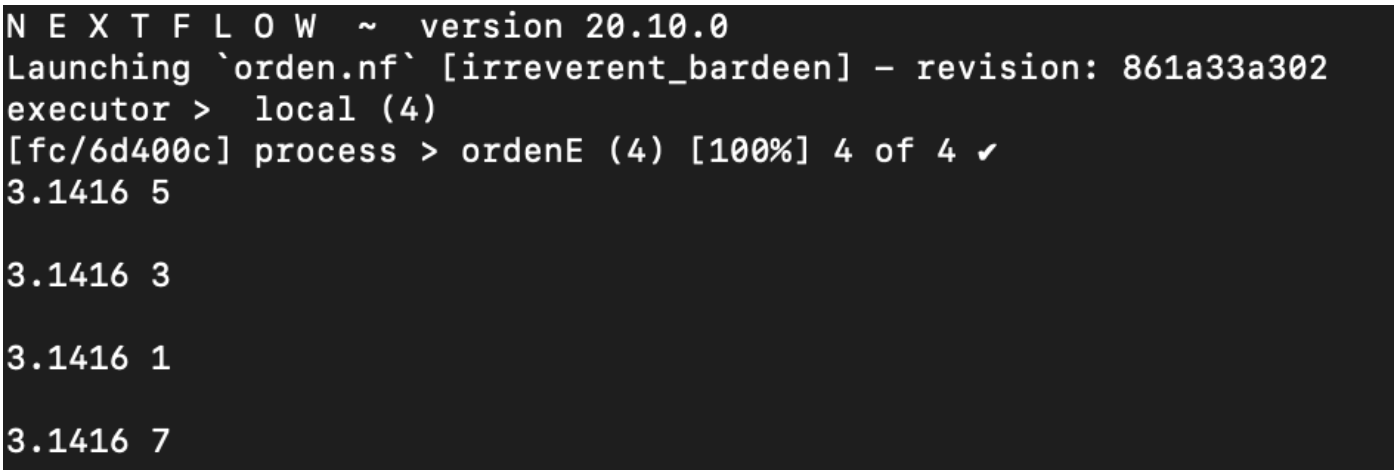

Figura 5. Output del *script* orden.nf.
Fuente: elaboración propia.

Nextflow también es capaz de generar métricas y reportes mediante opciones introducidas vía terminal.

Unos ejemplos serían:

- *with-report*: crea un informe de ejecución.
- *with-trace*: generará un archivo donde se indiquen parámetros de la ejecución, como memoria y *cpus* utilizadas, inicio de ejecución...
- *with-timeline*: permite identificar los cuellos de botella indicando el tiempo que consume cada proceso.

```
$ nextflow run orden.nf -with-timeline<p></p>
```

2. Nextflow

2.6. Operadores

En la sección anterior hemos visto cómo crear canales para dirigir los datos entre los procesos. Para poder modificar el contenido o el comportamiento de un canal, Nexflow ha creado lo que se denominan *operadores*. En los *scripts* anteriores hemos visto el operador *view*, pero podemos encontrar operadores de filtrado, combinación o de operaciones matemáticas entre muchos otros. En esta sección veremos algunos ejemplos.

Los operadores pueden introducirse mediante un *pipe* (|), como hemos visto anteriormente, o precedidos por un punto. Así:

```
workflow{

ordenE(pi,queue_ch) | view( )

}
```

es análogo a:

```
workflow{

ordenE(pi,queue_ch).view( )

}
```

A partir del script anterior eliminaremos el canal *valor pi* y nos quedaremos con un ejemplo más sencillo con el canal *of queue_ch*. Como veréis a continuación añadimos la notación *.view*, y dentro de este operador introducimos un prefijo, *chr*, y un valor *$it* entre {}. Estos paréntesis definen un bloque de código que va junto y utiliza la nomenclatura de *goovy* (*i*t, de *ítem*) para definir los parámetros.

```
nextflow.enable.dsl=2

queue_ch = Channel.of( 1, 3, 5, 7 ).view({"chr$it"})


process ordenE {

    input:

      val x


    output:

      stdout


"""

    echo $x

"""

}

workflow{

ordenE(queue_ch)

}
```

En este caso el *output* se visualiza en la figura 6.

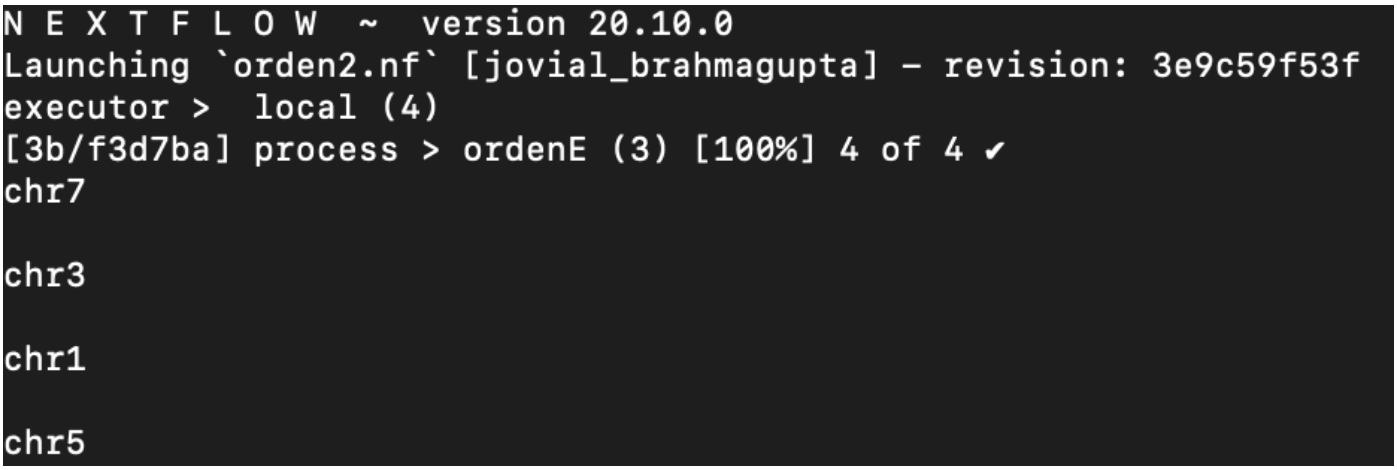

Figura 6. Resultado del operador *view* con prefijo.
Fuente: elaboración propia.

Podemos introducir un filtro en el canal para únicamente mostrar los valores superiores a 4:

```
queue_ch = Channel.of( 1, 3, 5, 7 ).filter { it > 4 }.view({"chr$it"})
```

También podemos combinar canales utilizando el operador *mix*:

```
ch1 = channel.of( 1..22 )

ch2 = channel.of( 'X','Y' )

ch3 = channel.of( 'MT' )


queue_ch  = ch1.mix(ch2,ch3).view({"chr$it"})
```

y hacer operaciones como contar el número de elementos:

```
queue_ch  = ch1.mix(ch2,ch3).count().view()
```

Las posibilidades de los operadores proporcionan una versatilidad muy grande de poder manipular los datos a analizar.

2. Nextflow

2.7. Configuración

Finalmente trataremos los archivos de configuración de Netxflow. Estos archivos son relevantes para poder migrar los *scripts* entre entornos de computación y para el control de los recursos a utilizar.

Podemos encontrar diversos archivos de configuración y algunos pueden entrar en conflicto entre ellos. Por ello Nextflow tiene una priorización:

1. Parámetros especificados en la línea de comandos.
2. Parámetros procedentes del archivo especificado mediante la opción -params-file.
3. Archivo de configuración especificado mediante la opción -c my_config.
4. Archivo de configuración nexflow.config en el directorio de trabajo.
5. Archivo de configuración nexflow.config en el directorio del proyecto de *workflow*.
6. Parámetros definidos en el mismo *script* de Nextflow.

Si un parámetro está en más de una de estas fuentes, Nextflow utiliza la primera fuente como referencia y no utiliza las subsiguientes.

El archivo consta de parejas nombre = valor

```
process.memory = '10G'
```

Un archivo de configuración puede incluirse en otro. Por ejemplo:

```
includeConfig 'path/foo.config'
```

La instrucción `includeConfig` busca el archivo de parámetros y los incluye como propios.

Los parámetros de configuración se pueden especificar en lo que se denominan *scopes*. Son parámetros que afectan específicamente a un tipo de configuración. Hay varios *scopes* de configuración, los más habituales de los cuales son:

- *aws*: Amazon S3.
- *conda*: entornos Conda.
- *docker*: contenedores Docker.
- *k8s*: clúster de Kubernetes.

Para habilitar la utilización de una imagen Docker, por ejemplo, se podría introducir en el nextflow.config:

```
docker.enabled = true
```

o se podría especificar directamente en la línea de comandos:

```
nextflow run <script> -with-docker [imagen Docker]
```

También es posible especificar una imagen Docker para un proceso determinado:

```
process one {

  container 'nombre_imagen_1'


  '''
```

```
  ejecución
  '''
}

process two {
  container 'nombre_imagen_2'

  '''
 ejecución
  '''
}
```

2. Nextflow

2.8. *nf-core*

Como otras iniciativas que ya hemos tratado anteriormente, Nextflow también tiene un repositorio de *workflows* para poder ser ejecutados. Este conjunto de *pipelines* se agrupa en el proyecto *nf-core* (https://nf-co.re/). Para poder trabajar directamente desde el terminal se ha generado el paquete *nf-core tools* para poder gestionar los distintos *workflows* accesibles en *nf-core*. *nf-core tools* está escrito en Python y se puede instalar desde la siguiente dirección: https://nf-co.re/tools.

Para poder ver los distintos paquetes simplemente se puede hacer un listado:

```
$ nf-core list
```

Muchas veces necesitamos ser más específicos en nuestra búsqueda y por ello podemos filtrar:

```
$ nf-core list rna
```

O inclusive ordenar por estrellas:

```
nf-core list rna --sort stars
```

Cada repositorio tiene su propia página con instrucciones y documentación. La utilización de estos repositorios públicos permite un aprendizaje más rápido de las herramientas de Nextflow en el contexto específico de la bioinformática. Igualmente, *nf-core* también proporciona un canal directo de preguntas mediante una cuenta en Slack (https://nf-co.re/join).

Resumen

En este módulo hemos dado una breve pincelada a los Workflow Managers y en especial a Nextflow. Para aumentar el control de los entornos en los cuales se computan los análisis es imprescindible utilizar Workflow Managers, y a poder ser, conjuntamente con entornos tipo Conda o contenedores, como hemos visto en capítulos anteriores. La utilización de este tipo de herramientas permite la portabilidad a cualquier entorno de computación y la implementación de *pipelines* generados por otros grupos de una manera ágil y sencilla. La modularización de los procesos permite que su actualización pueda ser dinámica y que la introducción de código generado por otros programadores no interfiera en otras partes del proceso. La versatilidad de opciones de los Workflow Managers permite un control preciso de todo el proceso, a veces en detrimento de su curva de aprendizaje. La diversidad de Workflow Managers que actualmente tenemos permite poder escoger el ecosistema en que nos encontremos más confortables, variando desde entornos muy visuales tipo Galaxy a otros enteramente programáticos. Es importante llegar a un compromiso entre el aprendizaje necesario para entender el funcionamiento de un *workflow* específico y las herramientas que este nos permite controlar. Finalmente, a la hora de escoger vuestro Workflow Manager se debe tener en cuenta la potencial interoperabilidad. En entornos colaborativos es imprescindible la reproducibilidad de procesos y la compartición de código para poder seguir los principios FAIR de los datos.

Actividades

1. Cread un *script* Nextflow que transforme la palabra «HOLA» (en mayúsculas) a toda en minúsculas.

2. Modificad a través de la llamada del *script* la palabra «HOLA» por «CIAO».

3. Introducid un nuevo proceso que sustituya la *A* por un 4.

4. Cread un *script* con dos canales *of* y un proceso que sume los valores de cada canal e imprima la operación realizada.

5. Buscad y bajaos un *workflow* a través de *nf-core*.

Bibliografía

Bjørn Fjukstad y Lars Ailo Bongo (2017). A Review of Scalable Bioinformatics Pipelines. Data Science and Engineering, 2, p. 245-251. https://doi.org/10.1007/s41019-017-0047-z

Ed H. B. M. Gronenschild y otros (2012). The Effects of FreeSurfer Version, Workstation Type, and Macintosh Operating System Version on Anatomical Volume and Cortical Thickness Measurements. *PLoS ONE*, *7*, e38234. Doi: 10.1371/journal.pone.0038234

Elise Larsonneur, Jonathan Mercier y Nicolas Wiart (2018). Evaluating Workflow Management Systems: A Bioinformatics Use Case. 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), p. 2.773-2.775. Doi: 10.1109/BIBM.2018.8621141

Jeffrey M. Perkel (2019). Workflow Systems Turn Raw Data into Scientific Knowledge. *Nature*, 573, p. 149-150. Doi: 10.1038/d41586-019-02619-z

Jeremy Leipzig (2017). A Review of Bioinformatic Pipeline Frameworks. *Briefings in Bioinformatics*, 18, p. 530-536. Doi: 10.1093/bib/bbw020

Paolo Di Tommaso y otros (2017). Nextflow Enables Reproducible Computational Workflows. *Nature Biotechnology*, 35, p. 316-319. https://doi.org/10.1038/nbt.3820

Paolo Di Tommaso y otros (2015). The Impact of Docker Containers on the Performance of Genomic Pipelines. PeerJ, 3, e1273. https://doi.org/10.7717/peerj.1273

Philip Ewels y otros (2020). The nf-core Framework for Community-curated Bioinformatics Pipelines. *Nature Biotechnology*, 38, p. 276-278. Doi: 10.1038/s41587-020-0439-x

Taylor Reiter y otros (2021). Streamlining Data-intensive Biology with Workflow Systems. *Gigascience*, 10, giaa140. https://doi.org/10.1093/gigascience/giaa140