

Introduction to Bioinformatics Workflows with Nextflow and nf-core

<https://carpentries-incubator.github.io/workflows-nextflow/01-getting-started-with-nextflow>

Setup

- Zoom
- Collaborative Document
- Virtual Machine
- Atom Text Editor
- Teaching Assistants

Workshop Dataset

- We will be working with a RNA-Sequencing experiment
- Budding yeast *Saccharomyces cerevisiae*
- Experiment: Changes in transcriptome following stress response
 - High ethanol concentration (ethanol 60 g/L)
 - Temperature (33 degree celsius)
 - Reference (control budding whole organism)
- Illumina HiSeq 2500 with paired ends (2× 100 base pairs)
- 3 replicates per condition
- Reads were down sampled 1% of total read count

Workshop Dataset

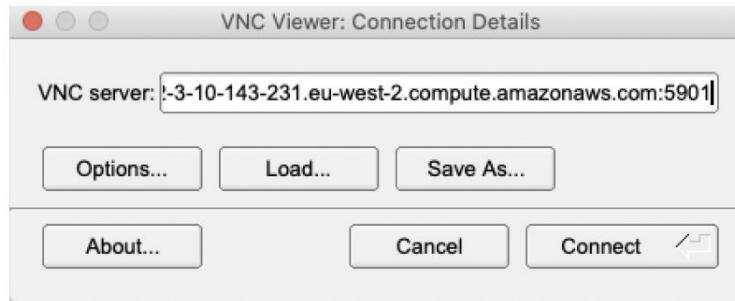
- FASTQ PE reads for `<sample_name>_<rep>_<read>.fq.gz`
 - **ethoh60**: High ethanol concentration (ethanol 60 g/L)
 - **temp33**: Temperature (33 degree celsius)
 - **ref**: Reference (control budding whole organism)
- Transcriptome
 - **Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz**

Collaborative Document

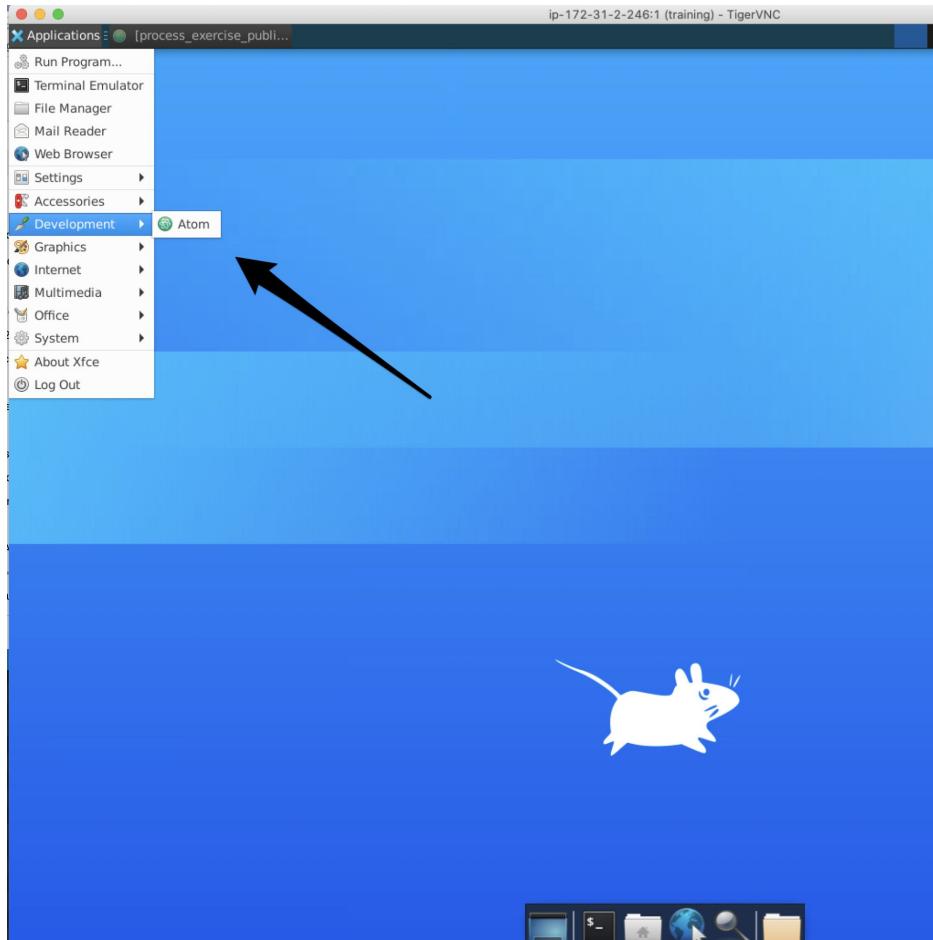
Connecting to VM

Enter the VM address assigned on the day (remember to include the port number)

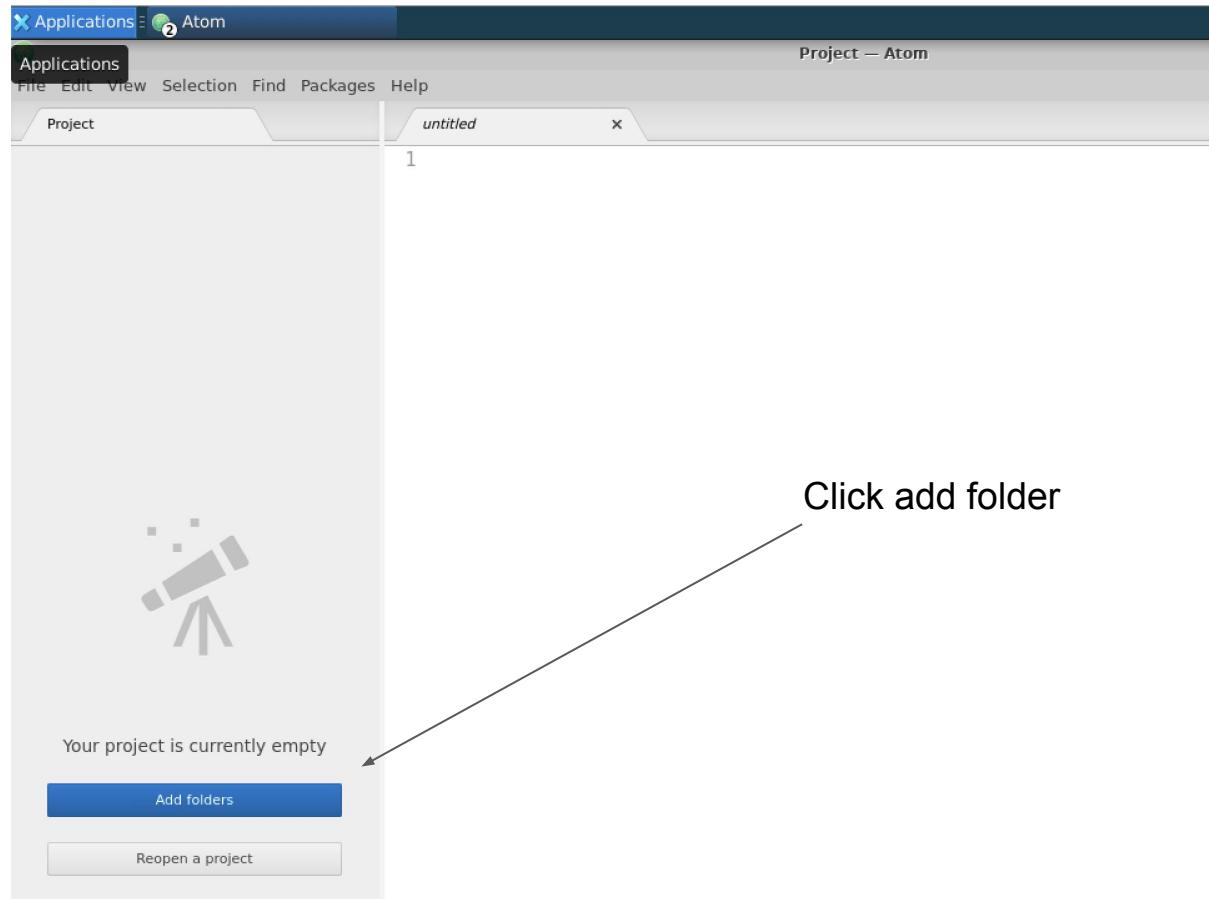
e.g. ec2-3-10-143-231.eu-west-2.compute.amazonaws.com:5901

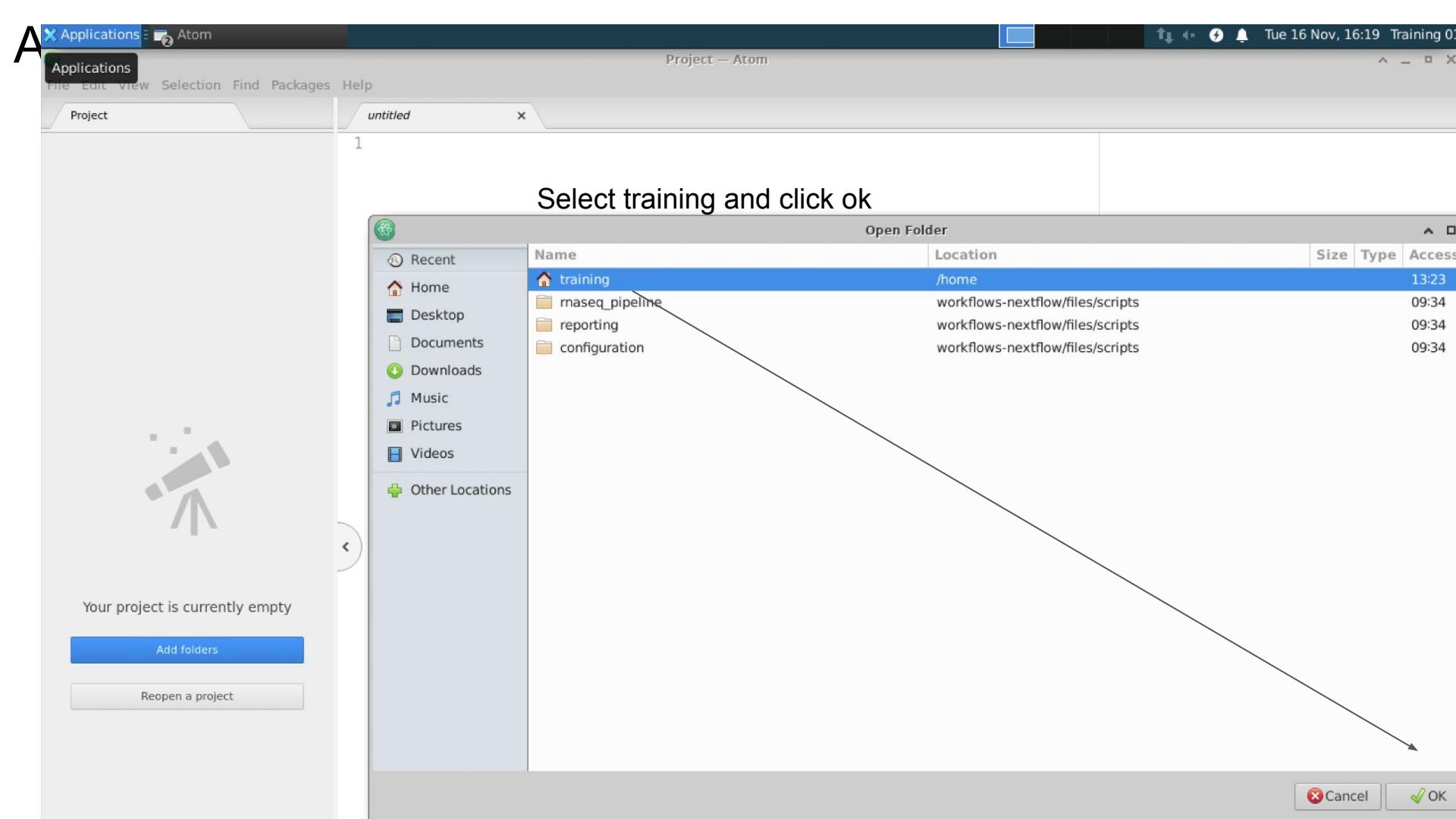


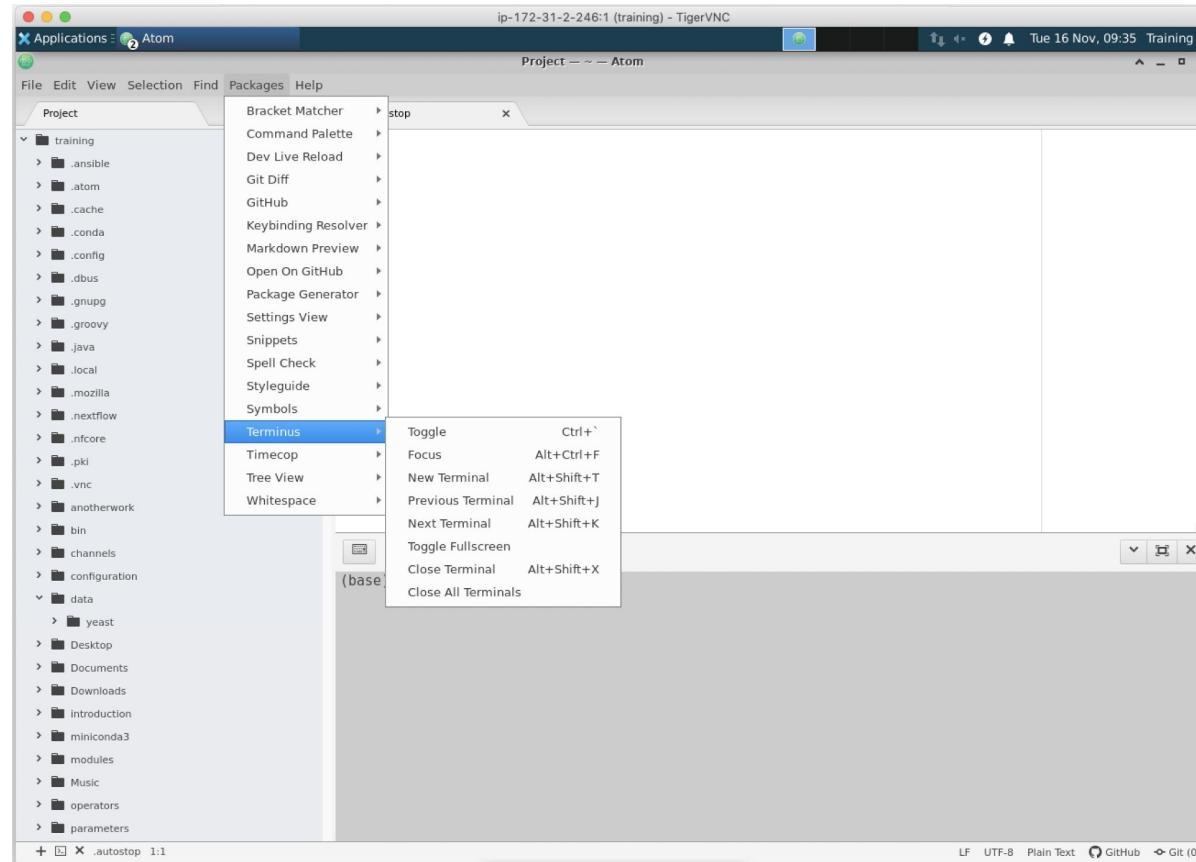
Atom Editor



Atom Editor







Day1: Morning

Episode	Time	Course Material
Getting Started With Nextflow	9:30-10:15	https://carpentries-incubator.github.io/workflows-nextflow/01-getting-started-with-nextflow
Workflow parameterization	10:15-11:00	https://carpentries-incubator.github.io/workflows-nextflow/03-workflow_parameters
Break	11:00-11:15	
Channels	11:15-12:15	https://carpentries-incubator.github.io/workflows-nextflow/04-channels
Break Lunch	12:15-13:15	

Day1: Afternoon

Episode	Time	Course Material
Processes Part 1	13:15-14:15	https://carpentries-incubator.github.io/workflows-nextflow/05-processes-part1
Processes Part 2	14:15-15:00	https://carpentries-incubator.github.io/workflows-nextflow/05-processes-part2
Break	15:00-15:15	
Operators	15:30-15:30-16:30	https://carpentries-incubator.github.io/workflows-nextflow/07-operators

Getting Started With Nextflow

<https://carpentries-incubator.github.io/workflows-nextflow/01-getting-started-with-nextflow/index.html>

Getting Started with Nextflow

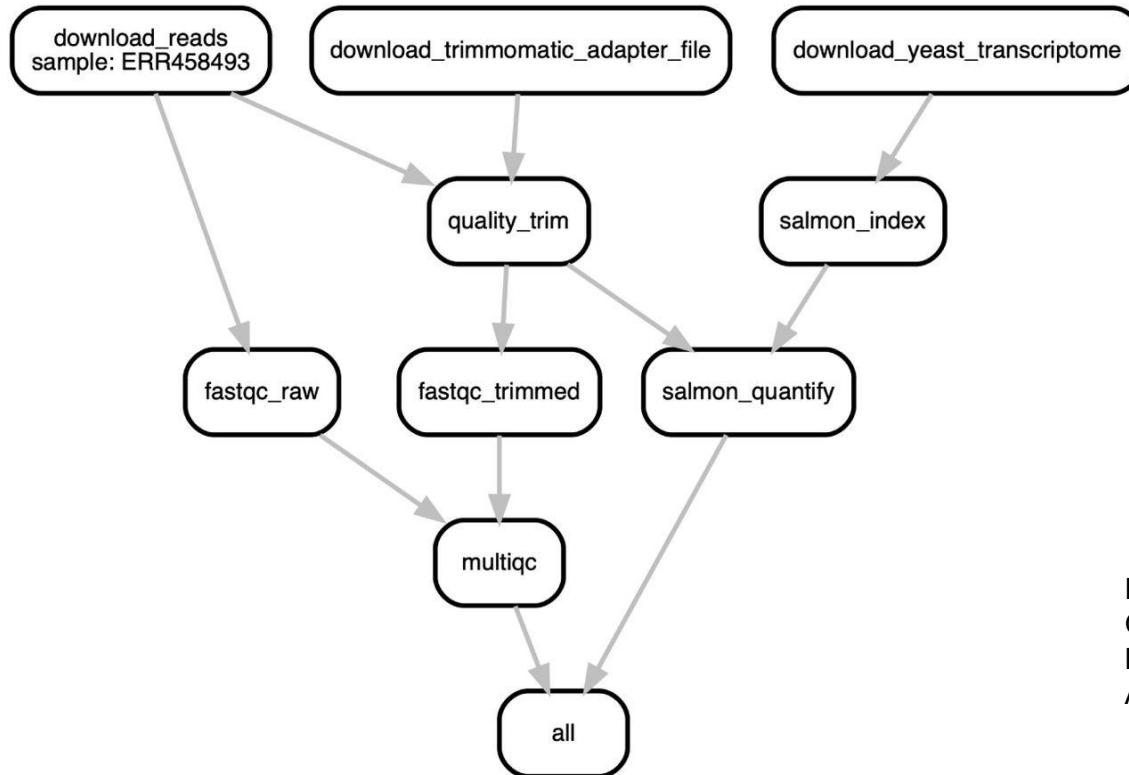
- What is a workflow and what are workflow management systems?
- Why should I use a workflow management system?
- What is Nextflow?
- What are the main features of Nextflow?
- What are the main components of a Nextflow script?
- How do I run a Nextflow script?

What is a workflow?

“A sequence of tasks that processes a set of data. “

Workflow = Pipeline

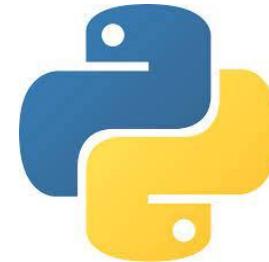
What is a workflow?: RNA workflow



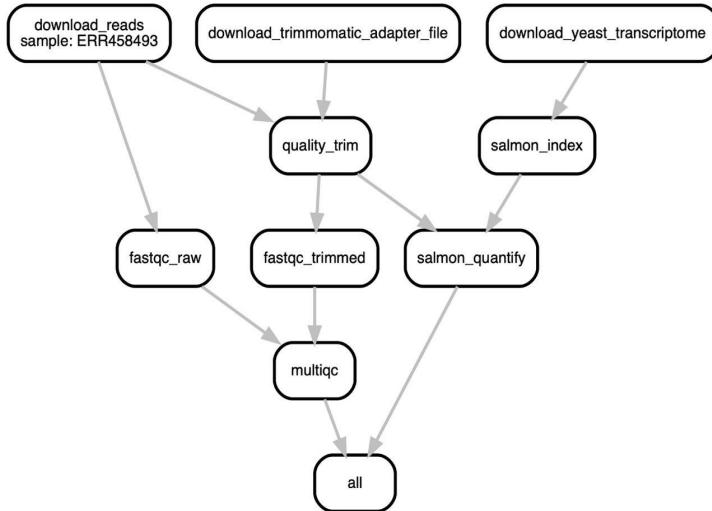
Directed acyclic graph.
One direction
Boxes tasks
Arrows flow of data

What is a Workflow?

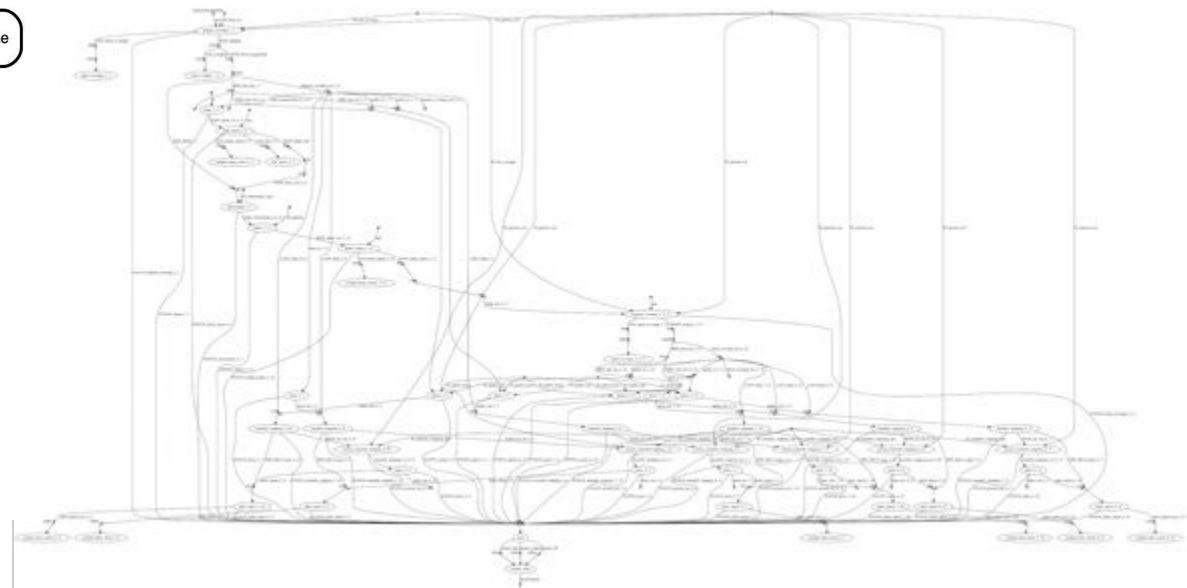
Traditionally these task have been joined together as a workflow using general purpose programming languages such as Bash or Python.



What is a workflow? Workflows can become very complicated



Simple RNA-Seq pipeline



Complex genome Assembly pipeline

<https://twitter.com/odiogosilva/status/98738600748507546>

Workflow Management Systems WfMS

Software and Programming Language developed to provide a framework for the

- Creation,
- Execution
- Monitoring of workflows/pipelines

WfMS: Key features

- Run time management

Management of program execution on the computer's operating system

Parallelisation, splitting tasks and data up to run at the same time

WfMS: Key features

- Run time management
- Software management

Use of software management technology

- containers, such as docker or singularity,
- packaging and environment managers , e.g. **Conda**

That packages up code and all its software dependencies so the application runs reliably from different computing environments.

WfMS: Key features

- Run time management
- Software management
- Portability & Interoperability

Workflows written on one system can be run on another computing infrastructure

- Local computer,
- HPC compute cluster
- Cloud infrastructure.

WfMS: Key features

- Run time management
- Software management
- Portability & Interoperability
- Reproducibility
- Reentrancy

Allow workflows to resume from the last successfully executed steps

Exercise: Discussion

- Join a Breakroom (5 people)
- Discuss for 5 minutes about a time you have used a non-reproducible workflow that negatively your work somehow.
- One person in the group write in Collaborative document the issues and how a Workflow Management System could help?

Bioinformatics Workflow Management Systems WfMS?



2005



Snakemake

2018



2017

What is Nextflow?

nextflow

Runtime

Domain Specific
Language

What is Nextflow? Runtime, software to run software

- In order for software to execute a program, it needs an environment to run in—usually an operating system (OS) like
 - Linux, Unix or MacOS
- Nextflow runtime is a software layer that runs on top of a computer's operating system software and provides resources that a specific Nextflow workflow needs to run
- Nextflow runtime acts as a kind of translator and facilitator between the Nextflow program and the OS.
- Nextflow runtime can run on multiple computing infrastructure

What are Nextflow's core features

- Fast prototyping (DSL)
- Reproducibility (containers, conda, git)
- Portability (separates wf logic from execution)
- Simple parallelism (dataflow programming)
- Continuous checkpoints (re-run from failure)

What is Nextflow?

nextflow

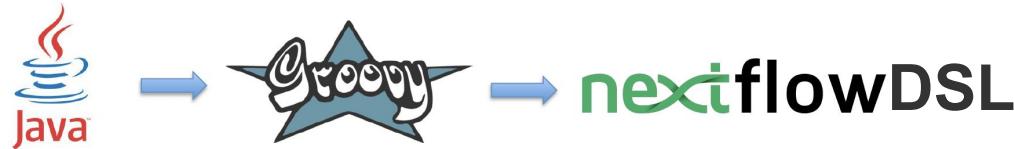
Runtime

Domain Specific
Language

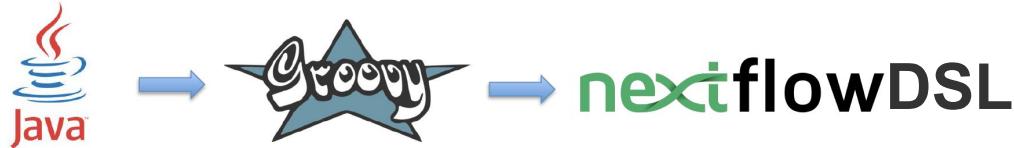
What is Nextflow? Domain Specific Language DSL

- Nextflow scripts are written using a DSL scripting language that simplifies the writing of workflows.
- A DSL is a programming language that is developed to meet a specific need within a particular domain
- General Purpose Language (Do anything)
 - Python, Java
- Domain specific language (A specific task)
 - SQL, (databases)
 - AWK (Text processing)
 - Nextflow (workflows)

What is Nextflow? DSL



What is Nextflow? DSL



.The Nextflow syntax can handle most workflow use cases with ease, and then Groovy can be used to handle corner cases which may be difficult to implement using the DSL.

What is Nextflow? DSL2



Modularity (separating components to provide flexibility and enable reuse),

- Simplifies the writing of complex data analysis pipelines, and enhances workflow readability, and reusability.

Nextflow : Philosophy

Nextflow follows Linux's “[small pieces loosely joined](#)” philosophy:

In which many simple but powerful command-line and scripting tools, when chained together, facilitate more complex data manipulations.

Consequently, the developer can view a pipeline as a collection of tasks, where the execution is orchestrated by Nextflow

Nextflow :Dataflow Paradigm

Nextflow extends that approach using dataflow programming model

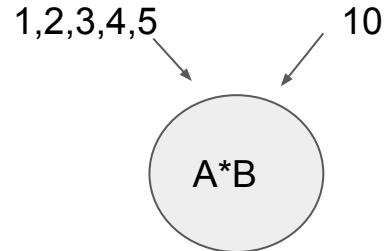
Allows the definition of tasks that execute in parallel.

Tasks are linked via inputs and outputs and run as soon as they receive input.

What is Dataflow Programming?

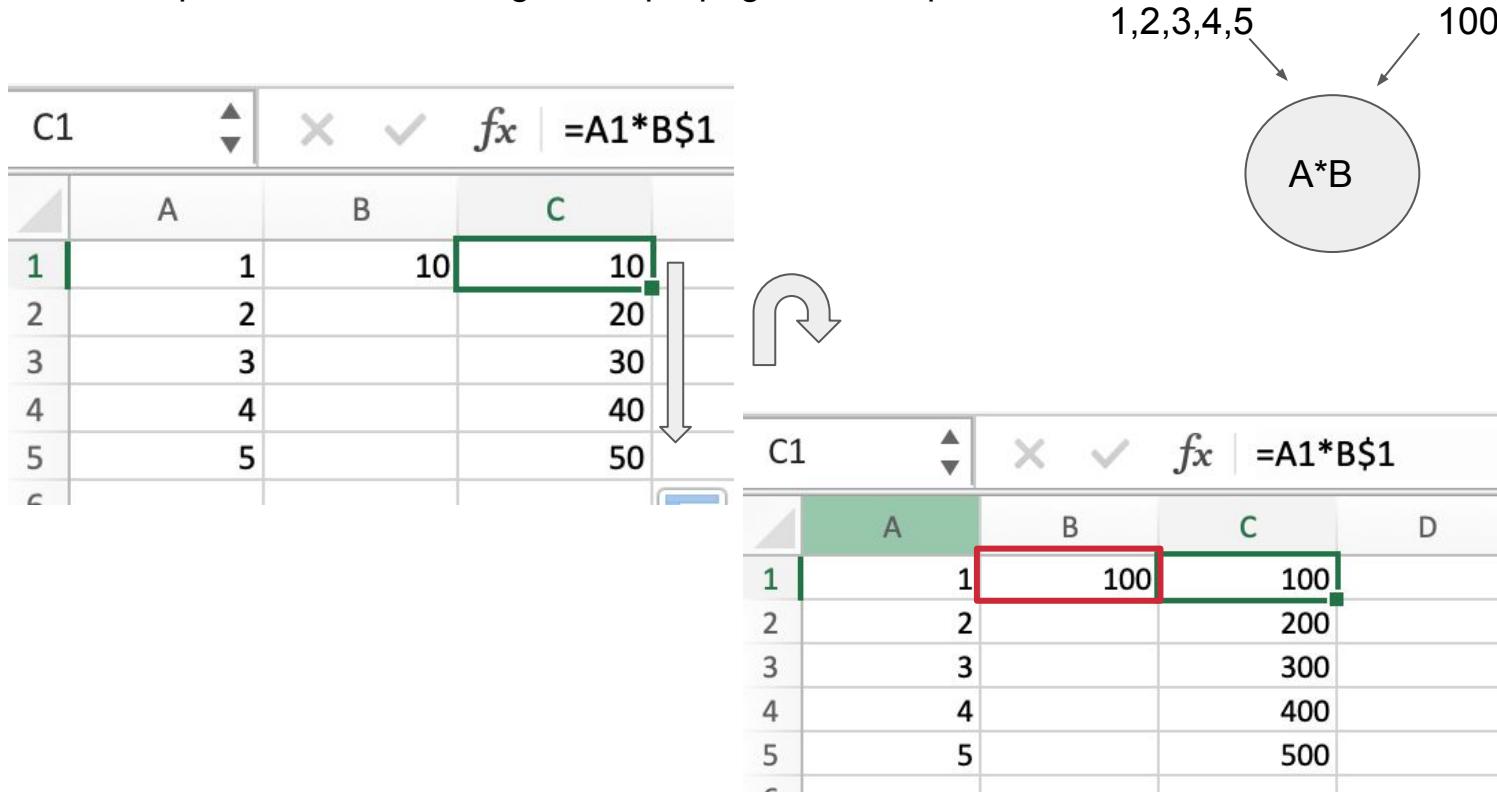
- When an input is modified changes are propagated to dependent tasks

C1	A	B	C
1	1	10	10
2	2		20
3	3		30
4	4		40
5	5		50



What is Dataflow Programming?

- When an input is modified changes are propagated to dependent tasks



What is Dataflow Programming?

2. Independent tasks can run sequentially.

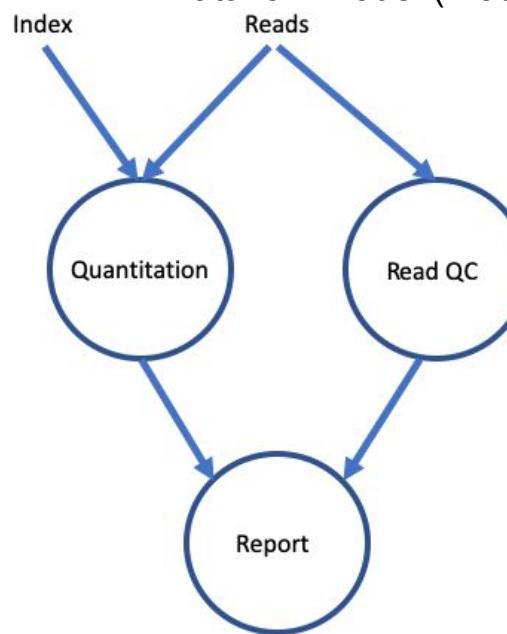
Linear model

1. Quantitation:= Index + Reads
2. Read QC:= Reads
3. Report:= Quantitation + Read QC

(a)

3 units of Time

Dataflow model (modelled as DAG)



(b)

2 units of time (parallel)

Independent Tasks;

Don't require input
from one another.

Run as soon as
they receive input

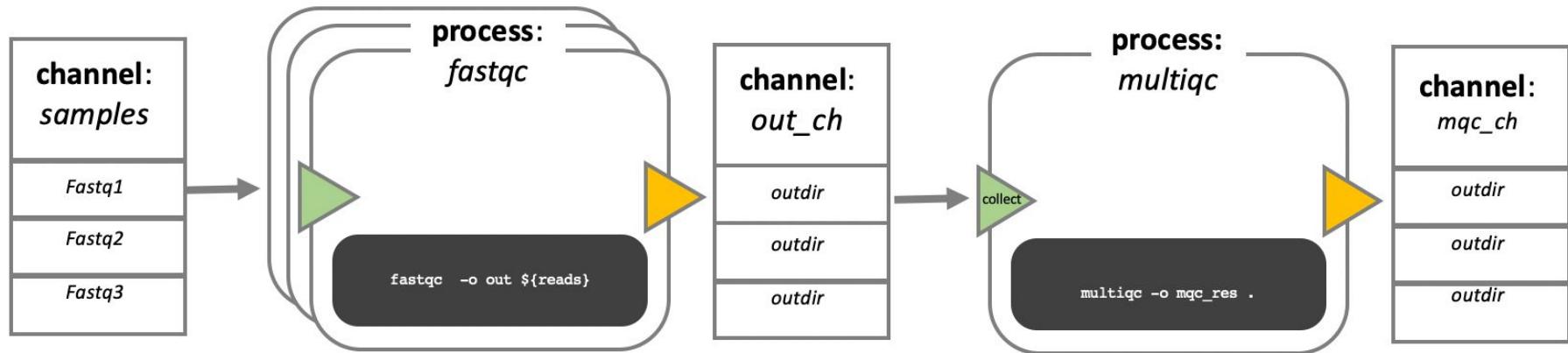
Nextflow concepts

Nextflow workflows have two main components;

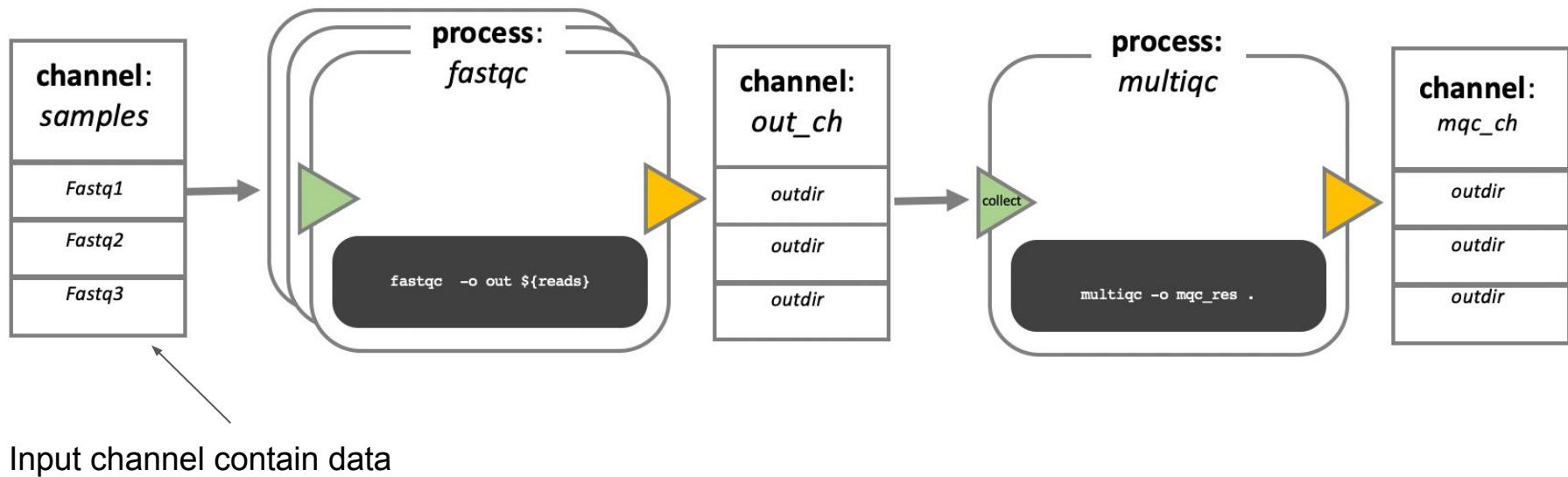
Processes (tasks)

Channels (communication channels between Processes)

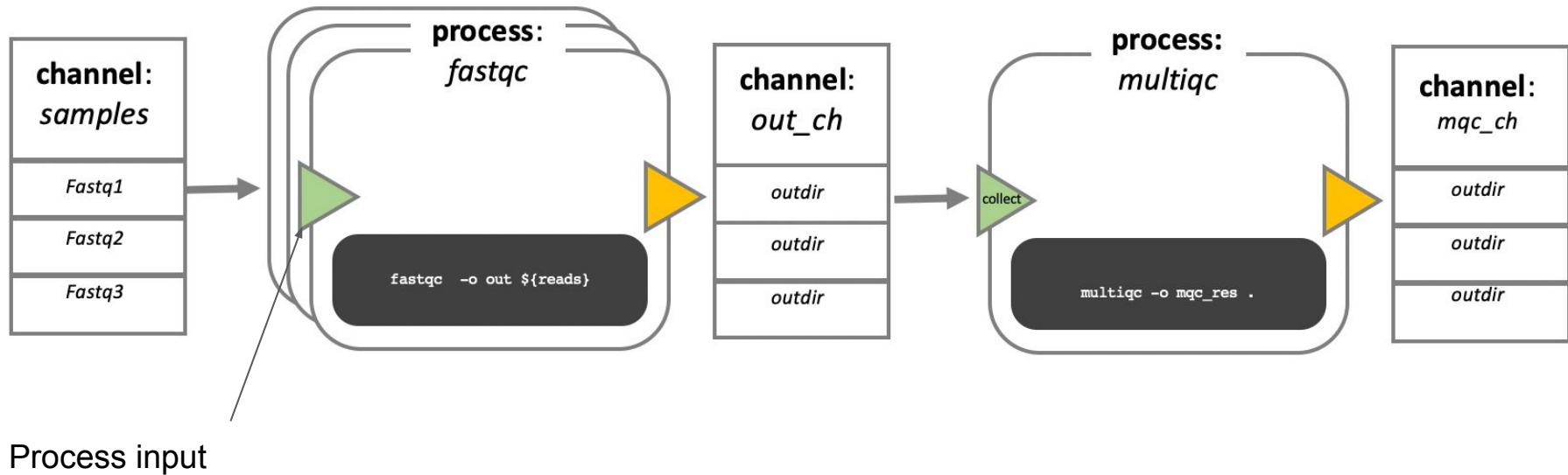
Nextflow: Processes and Channels



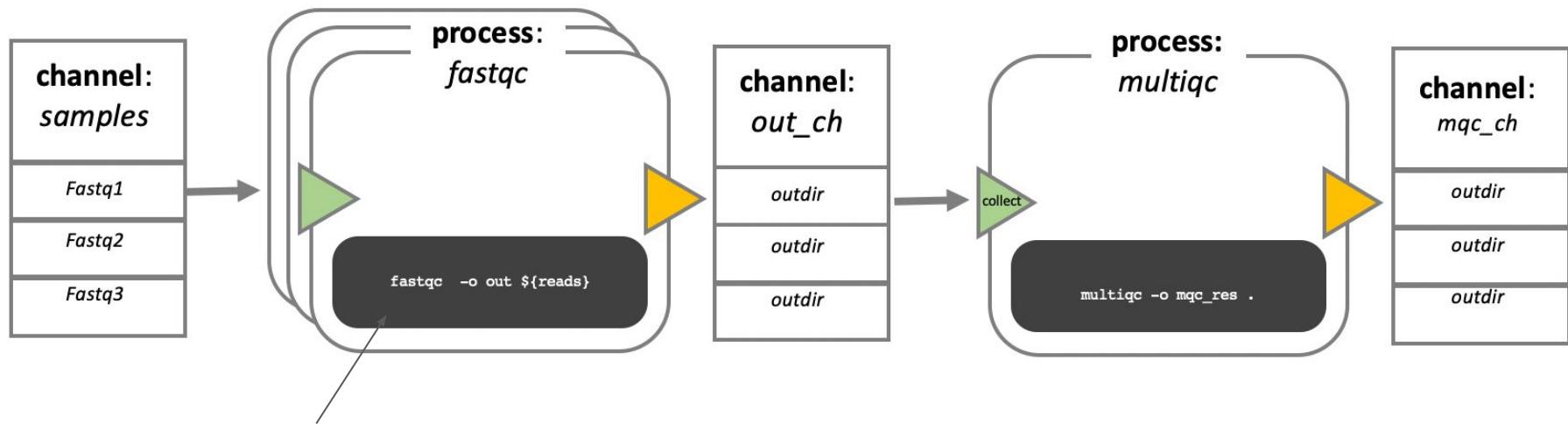
Nextflow: Processes and Channels



Nextflow: Processes and Channels



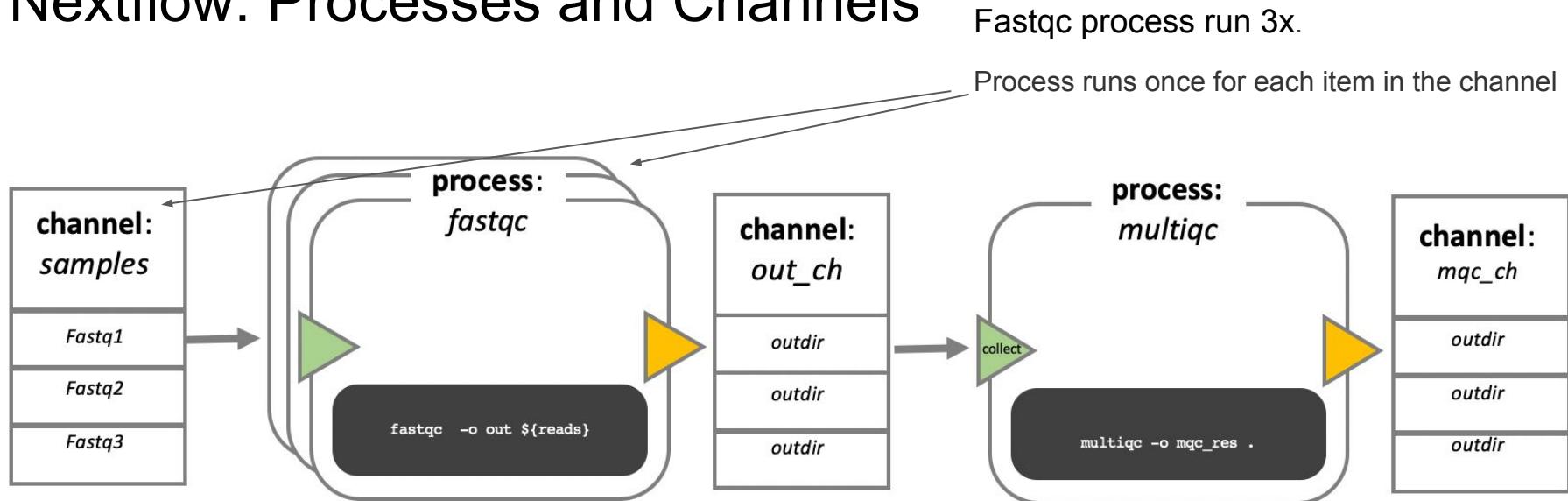
Nextflow: Processes and Channels



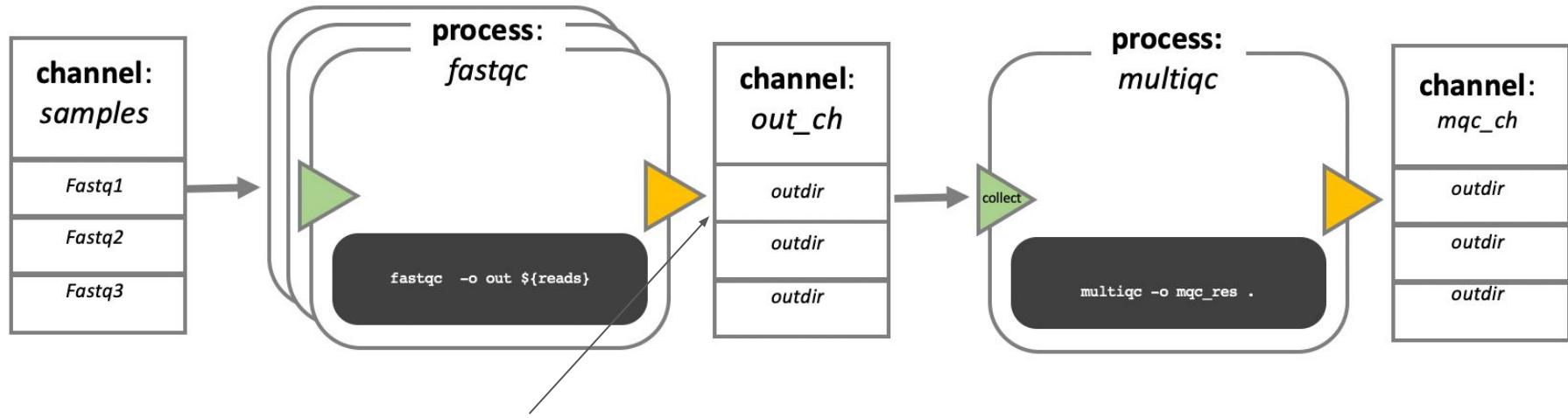
Process script

written in any scripting language that can be executed by the Linux platform (Bash, Perl, Ruby, Python, etc.).

Nextflow: Processes and Channels



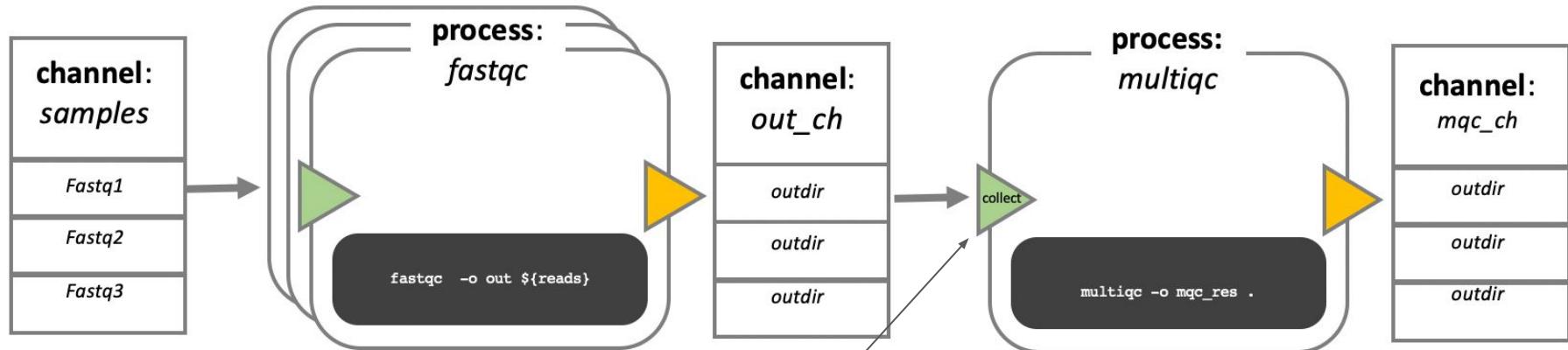
Nextflow: Processes and Channels



Process output

The only way data can be passed between process tasks is channels.

Nextflow: Processes and Channels

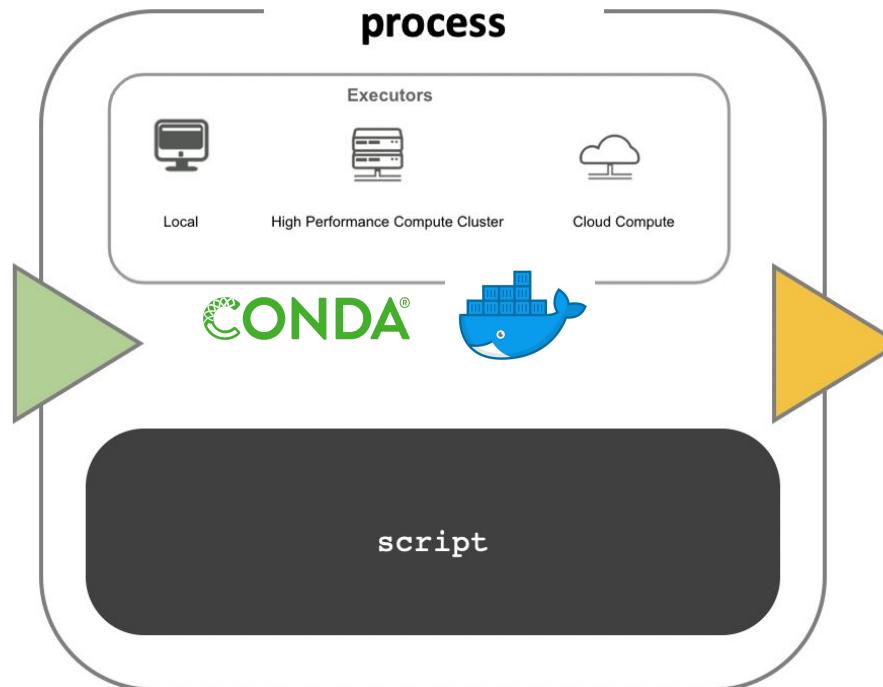


Channel operators

We can change how channels behave.

Collect operator means collects all items together and multiqc runs once

Nextflow: Workflow Execution



Nextflow separates workflow logic from execution

Your first script: **wc.nf**

Counting the number of lines in gzipped files `.**gz**`

Open Atom editor and open script **wc.nf**

wc.nf: Shebang

An optional interpreter directive (“Shebang”) line, specifying the location of the Nextflow interpreter.

```
#!/usr/bin/env nextflow
```

wc.nf: DSL2

To enable DSL2 syntax.

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

wc.nf: comments

```
/* Comments are uninterpreted text included with the script.  
They are useful for describing complex parts of the workflow  
or providing useful information such as workflow usage.
```

Usage:

```
nextflow run wc.nf --input <input_file>
```

Multi-line comments start with a slash asterisk / and finish with an asterisk slash. */*

```
// Single line comments start with a double slash // and finish on the same line
```

wc.nf: Parameters

A pipeline parameter `params.input` which is given a default value, of the relative path to the location of a compressed fastq file, as a string.

```
/* Workflow parameters are written as params.<parameter>
   and can be initialised using the `=` operator. */
params.input = "data/yeast/reads/ref1_1.fq.gz"
```

wc.nf: Workflow

An unnamed `workflow` execution block, which is the default workflow to run.

```
// The default workflow
workflow {

    // Input data is received through channels
    input_ch = Channel.fromPath(params.input)

    /* The script to execute is called by its process name,
       and input is provided between brackets. */
    NUM_LINES(input_ch)

    /* Process output is accessed using the `out` channel.
       The channel operator view() is used to print
       process output to the terminal. */
    NUM_LINES.out.view()
}
```

wc.nf: Channel

A Nextflow channel used to read in data to the workflow

```
// Input data is received through channels
input_ch = Channel.fromPath(params.input)
```

wc.nf: Workflow: calling a process and passing input

A call to the process `NUM_LINES`

```
/* The script to execute is called by its process name,  
   and input is provided between brackets. */  
NUM_LINES(input_ch)
```

wc.nf: Workflow: Operators

An operation on the process output, using the channel operator `view()`.

```
/* Process output is accessed using the `out` channel.  
The channel operator view() is used to print  
process output to the terminal. */  
NUM_LINES.out.view()
```

wc.nf: Process block

A Nextflow `process` block named `NUM_LINES`, which defines what the process does

```
/* A Nextflow process block
Process names are written, by convention, in uppercase.
This convention is used to enhance workflow readability. */
process NUM_LINES {

    input:
    path read

    output:
    stdout

    script:
    /* Triple quote syntax """", Triple-single-quoted strings may span multiple lines. Th
e content of the string can cross line boundaries without the need to split the string i
n several pieces and without concatenation or newline escape characters. */
    """
    printf '${read} '
    gunzip -c ${read} | wc -l
    """
}

}
```

wc.nf: Process block: input

An `input` definition block that assigns the input to the variable `read`, and declares that it should be interpreted as a file `path`.

```
input:  
path read
```

wc.nf: Process block: output

An `output` definition block that uses the Linux/Unix standard output stream `stdout` from the script block.

```
output:  
  stdout
```

wc.nf: Process block: script

A `script` block that contains the bash commands to count the number of lines in the gzipped read file.

```
script:  
/* Triple quote syntax """", Triple-single-quoted strings may span multiple lines. Th  
e content of the string can cross line boundaries without the need to split the string i  
n several pieces and without concatenation or newline escape characters. */  
....  
printf '${read} '  
gunzip -c ${read} | wc -l  
....
```

wc.nf : Running the script

```
$ nextflow run wc.nf
```

Key Points

- A workflow is a sequence of tasks that process a set of data.
- A workflow management system (WfMS) is a computational platform that provides an infrastructure for the set-up, execution and monitoring of workflows.
- Nextflow is a workflow management system that comprises both a runtime environment and a domain specific language (DSL).
- Nextflow scripts comprise of channels for controlling inputs and outputs, and processes for defining workflow tasks.
- You run a Nextflow script using the `nextflow run` command.

Workflow Parameterization

https://carpentries-incubator.github.io/workflows-nextflow/03-workflow_parameters

Questions

- How can I change the data a workflow uses?
- How can I parameterise a workflow?
- How can I add my parameters to a file

wc.nf: Parameters

In the Nextflow script, `wc.nf`, it counted the number of lines in the file defined in the `params.input` variable

```
/* Workflow parameters are written as params.<parameter>
   and can be initialised using the `=` operator. */
params.input = "data/yeast/reads/ref1_1.fq.gz"
```

Pipeline parameters: Command line option

To change the input to script we can make use of pipeline parameters.

Their value can be specified on the command line by prefixing the parameter name with a **double dash** character, e.g.,

```
--input
```

```
$ nextflow run wc.nf --input 'data/yeast/reads/ref2_2.fq.gz'
```

```
/* Workflow parameters are written as params.<parameter>
   and can be initialised using the '=' operator. */
params.input = "data/yeast/reads/ref1_1.fq.gz"
```

Workflow parameterization: Dealing with Multiple Files

In our `data/yeast/reads` directory we have multiple files

```
$ ls data/yeast/reads
etoh60_1_1.fq.gz      etoh60_2_2.fq.gz      ref1_1.fq.gz      ref2_2.fq.gz      temp33_1_1.fq.gz      temp33_2_2.fq.gz
etoh60_1_2.fq.gz      etoh60_3_1.fq.gz      ref1_2.fq.gz      ref3_1.fq.gz      temp33_1_2.fq.gz      temp33_3_1.fq.gz
etoh60_2_1.fq.gz      etoh60_3_2.fq.gz      ref2_1.fq.gz      ref3_2.fq.gz      temp33_2_1.fq.gz      temp33_3_2.fq.gz
```

To match multiple files we use Bash pattern matching syntax, which is called globbing.

Bash Globbing

Bash does not support native regular expressions like some other standard programming languages.

The Bash shell feature that is used for matching or expanding specific types of patterns is called globbing.

Globbing is mainly used to match *filenames*.

Globbing uses wildcard characters to create the pattern.

Bash Globbing, Recap

*

Is used to match zero or more characters.

```
$ ls -1 data/yeast/reads/ref1*.fq.gz  
data/yeast/reads/ref1_1.fq.gz  
data/yeast/reads/ref1_2.fq.gz  
$ █
```

Bash Globbing, Recap

- * * is used to match zero or more characters.
- ** ** same as * but matches multiple directory levels

Bash Globbing, Recap

* * is used to match zero or more characters.

** ** same as * but matches multiple directory levels

{a,b} {} can be used to match filenames with more than one globbing patterns. Each pattern is separated by ',' in curly bracket without any space

```
$ ls -1 data/yeast/reads/ref1_{1,2}.fq.gz
data/yeast/reads/ref1_1.fq.gz
data/yeast/reads/ref1_2.fq.gz
```

Exercise: wc.nf

Change a pipeline's input using a parameter

Re-run the Nextflow script `wc.nf` by changing the pipeline input to all files in the directory `data/yeast/reads/` that begin with `ref` and end with `.fq.gz`:

Solution

Hint:

```
params.reads
```

```
--
```

Workflow parameterization: Adding a parameter to a script

To add a pipeline parameter to a script prepend the prefix `params`

```
params.params_name = value
```

Workflow parameterization: Adding a parameter to a script

To add a pipeline parameter to a script prepend the prefix `params`, separated by a dot character `.`, to a variable name e.g., `params.input`.

```
params.params_name = value
```

Workflow parameterization: Adding a parameter to a script

To add a pipeline parameter to a script prepend the prefix `params`, separated by a dot character `.`, to a variable name e.g., `params.input`.

```
params.params_name = value
```

Workflow parameterization: Adding a parameter to a script

To add a pipeline parameter to a script prepend the prefix `params`, separated by a dot character `.`, to a variable name e.g., `params.input`.

Then use the assignment operator `=` to assign a value e.g. number, string, boolean

```
params.params_name = value
```

Workflow parameterization: Adding a parameter to a script

To add a pipeline parameter to a script prepend the prefix `params`, separated by a dot character `.`, to a variable name e.g., `params.input`.

Then use the assignment operator `=` to assign a value e.g. number, string, boolean

```
params.params_name = value
```

```
params.sleep = 2
```

Exercise: wc-params.nf

Add a pipeline parameter

If you haven't already make a copy of the `wc.nf` as `wc-params.nf`.

Code

```
$ cp wc.nf wc-params.nf
```

Add the param `sleep` with a default value of 2 below the `params.input` line. Add the line `sleep ${params.sleep}` in the process `NUM_LINES` above the line `printf '${read}'`.

Run the new script `wc-params.nf` changing the sleep input time.

What input file would it run and why?

How would you get it to process all `.fq.gz` files in the `data/yeast/reads` directory as well as changing the sleep input to 1 second?

```
params.input = "data/yeast/reads/ref1_1.fq.gz"
```

```
.....  
printf '${read} '\ngunzip -c ${read} | wc -l  
.....
```

Workflow parameterization: Parameter File

- If we have many parameters to pass to a script it is best to create a parameters file.
- Parameters are stored in *JSON* or *YAML* format.

```
{  
    "sleep": 5,  
    "input": "data/yeast/reads/etoh60_1*.fq.gz"  
}
```

We can read in the parameters from the file using `-param-file` option

```
$ nextflow run wc-params.nf -params-file wc-params.json
```

Exercise: wc-params.nf

Create a parameter file `params.json` for the Nextflow file `wc-params.nf`, and run the Nextflow script using the created parameter file, specifying:

- sleep as 10
- input as `data/yeast/reads/ref3_1.fq.gz`

Hint

```
{  
    "param_name": number,  
    "param_name": "path"  
}
```

Key Points

- Pipeline parameters are specified by prepending the prefix `params` to a variable name, separated by dot character.

```
params.input = "data/yeast/reads/ref1_1.fq.gz"
```

Key Points

- Pipeline parameters are specified by prepending the prefix `params` to a variable name, separated by dot character.

```
params.input = "data/yeast/reads/ref1_1.fq.gz"
```

- To specify a pipeline parameter on the command line for a Nextflow run use `--variable_name` syntax.

```
$ nextflow run wc.nf --input 'data/yeast/reads/ref2_*.fq.gz'
```

Key Points

- Pipeline parameters are specified by prepending the prefix `params` to a variable name, separated by dot character.

```
params.input = "data/yeast/reads/ref1_1.fq.gz"
```

- To specify a pipeline parameter on the command line for a Nextflow run use `--variable_name` syntax.

```
$ nextflow run wc.nf --input 'data/yeast/reads/ref2_*.fq.gz'
```

- You can add parameters to a JSON or YAML formatted file and pass them to the script using option `-params-file`

```
{
  "sleep": 5,
  "input": "data/yeast/reads/etoh60_1*.fq.gz"
}
```

```
$ nextflow run wc-params.nf -params-file wc-params.json
```

Break 15 mins

Channels

<https://carpentries-incubator.github.io/workflows-nextflow/04-channels>

Questions

- How do I get data into Nextflow?
- How do I handle different types of input, e.g. files and parameters?
- How do I create a Nextflow channel?
- How can I use pattern matching to select input files?
- How do I change the way inputs are handled?

Channels

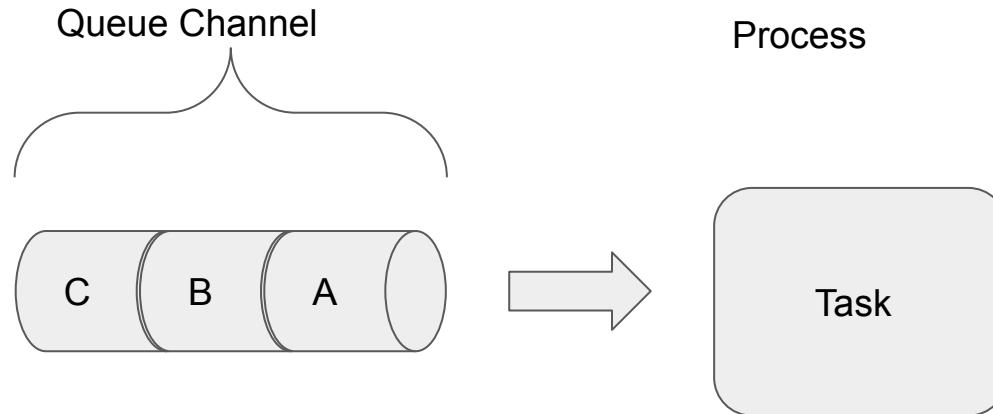
- Channels are how Nextflow sends data around a workflow
- Data can
 - Values, numbers, strings, booleans
 - Files

Channels: Properties

Channels have two important properties

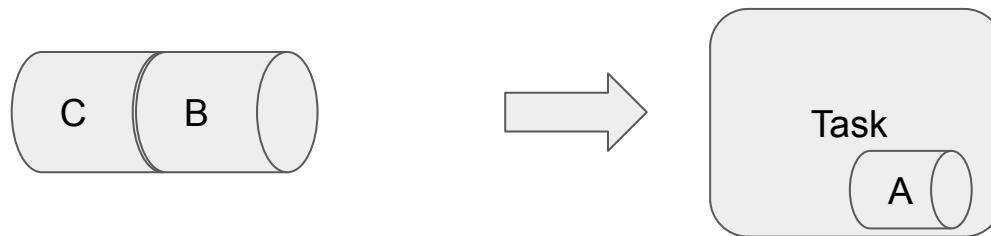
- First In First Out: FIFO

Channel: Properties: FIFO



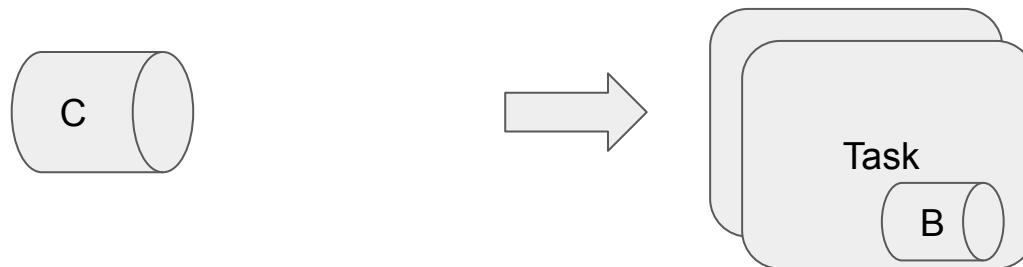
First element into a channel queue is the first out of the queue (First in - First out)

Channel: Properties: FIFO



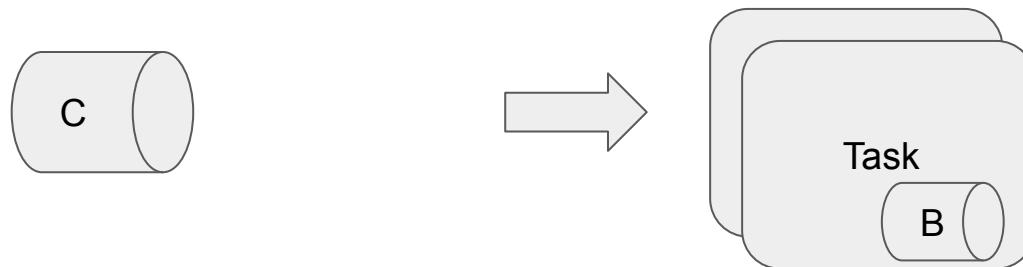
First element into a channel queue is the first out of the queue (First in - First out)

Channel: Properties: FIFO



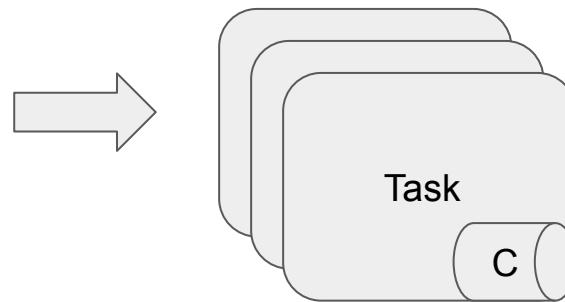
First element into a channel queue is the first out of the queue (First in - First out)

Channel: Properties: FIFO



First element into a channel queue is the first out of the queue (First in - First out)

Channel: Properties: FIFO



First element into a channel queue is the first out of the queue (First in - First out)

Channels: Properties

Channels have two important properties

- First In First Out: FIFO
- Channels are Asynchronous

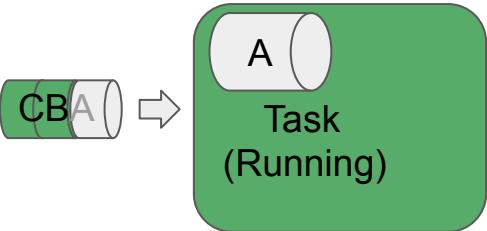
Channel: Properties: Asynchronous

Outputs from a set of processes will not necessarily be produced in the same order as the corresponding inputs went in.



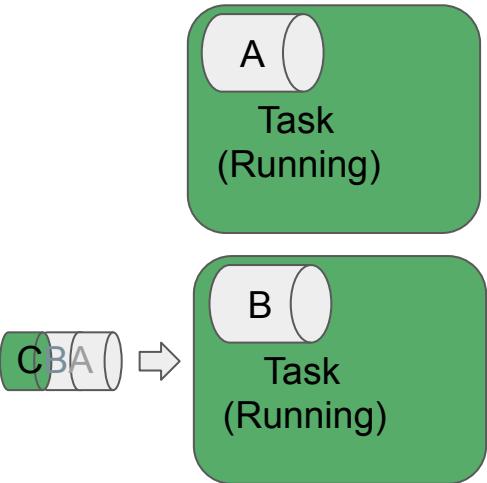
Channel: Properties: Asynchronous

Outputs from a set of processes will not necessarily be produced in the same order as the corresponding inputs went in.



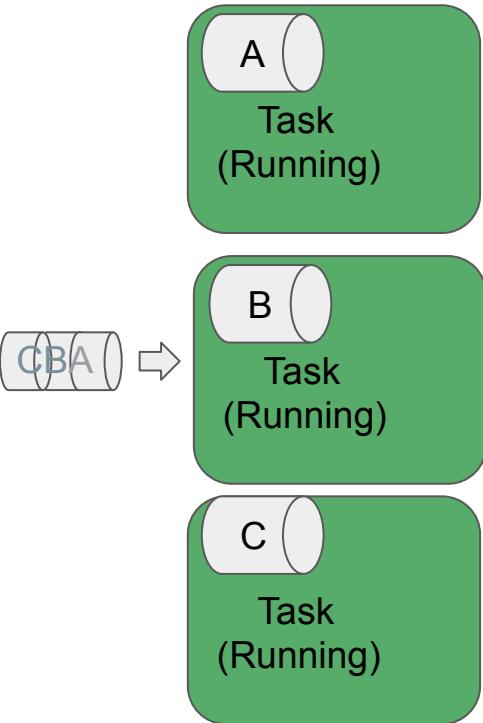
Channel: Properties: Asynchronous

Outputs from a set of processes will not necessarily be produced in the same order as the corresponding inputs went in.



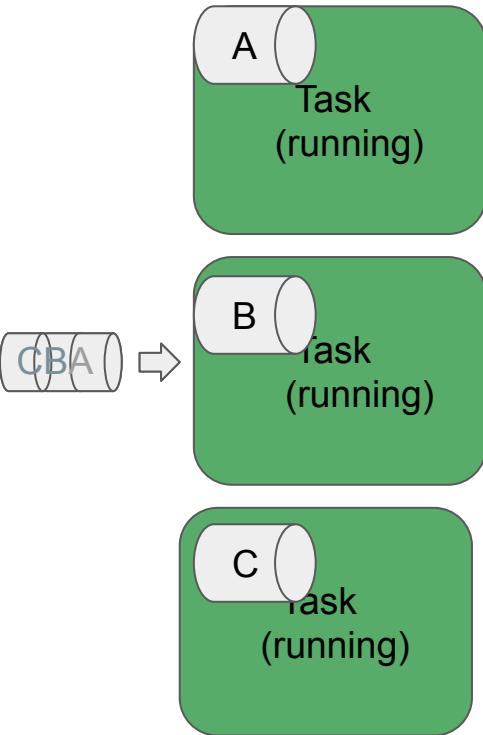
Channel: Properties: Asynchronous

Outputs from a set of processes will not necessarily be produced in the same order as the corresponding inputs went in.



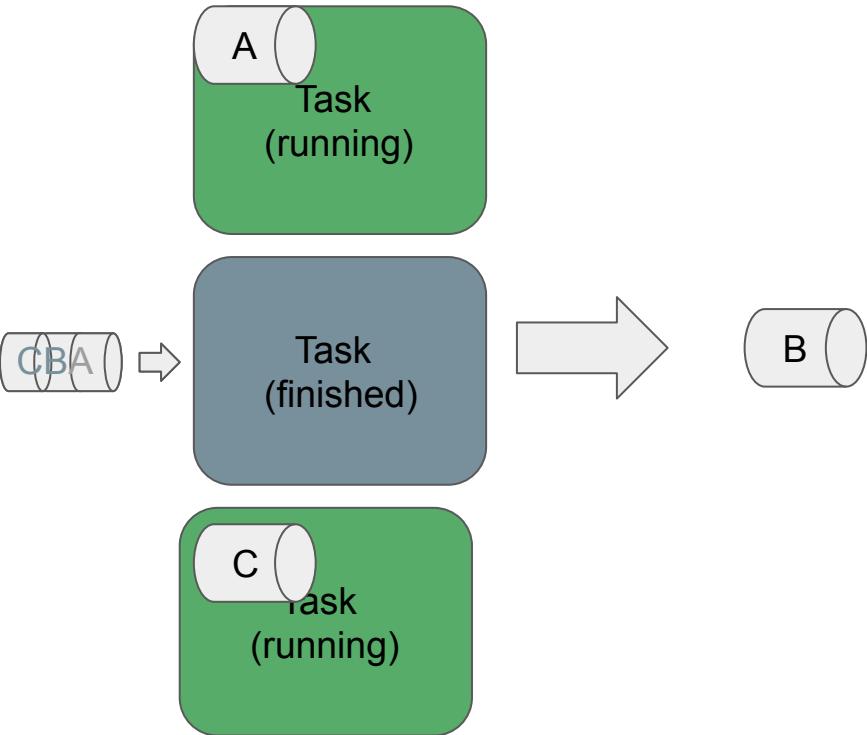
Channel: Properties: Asynchronous

Outputs from a set of processes will not necessarily be produced in the same order as the corresponding inputs went in.



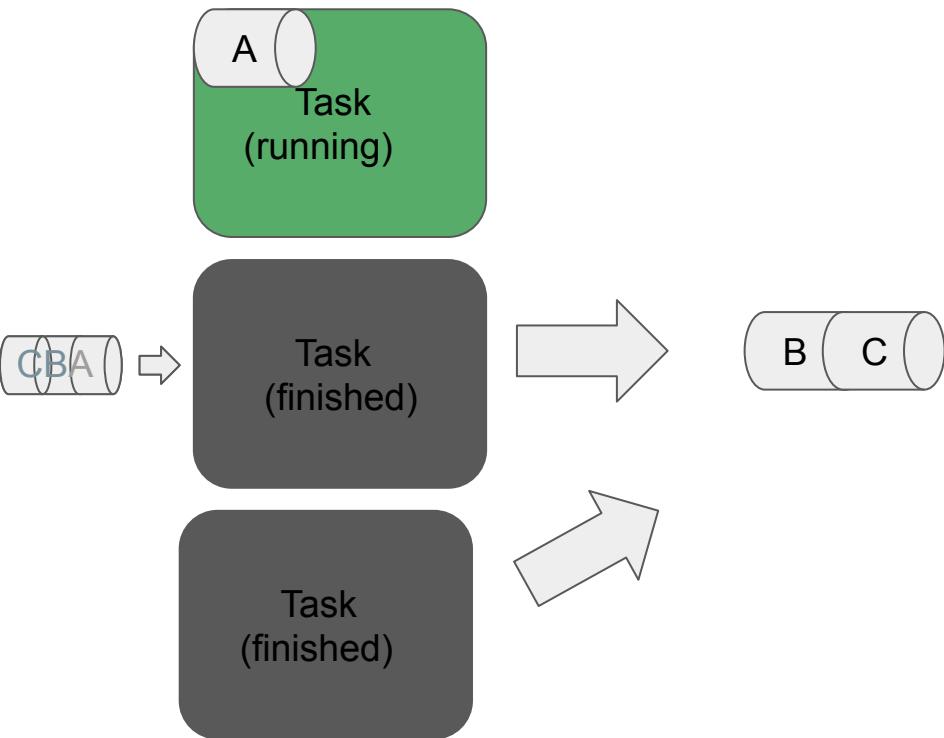
Channel: Properties: Asynchronous

Outputs from a set of processes will not necessarily be produced in the same order as the corresponding inputs went in.



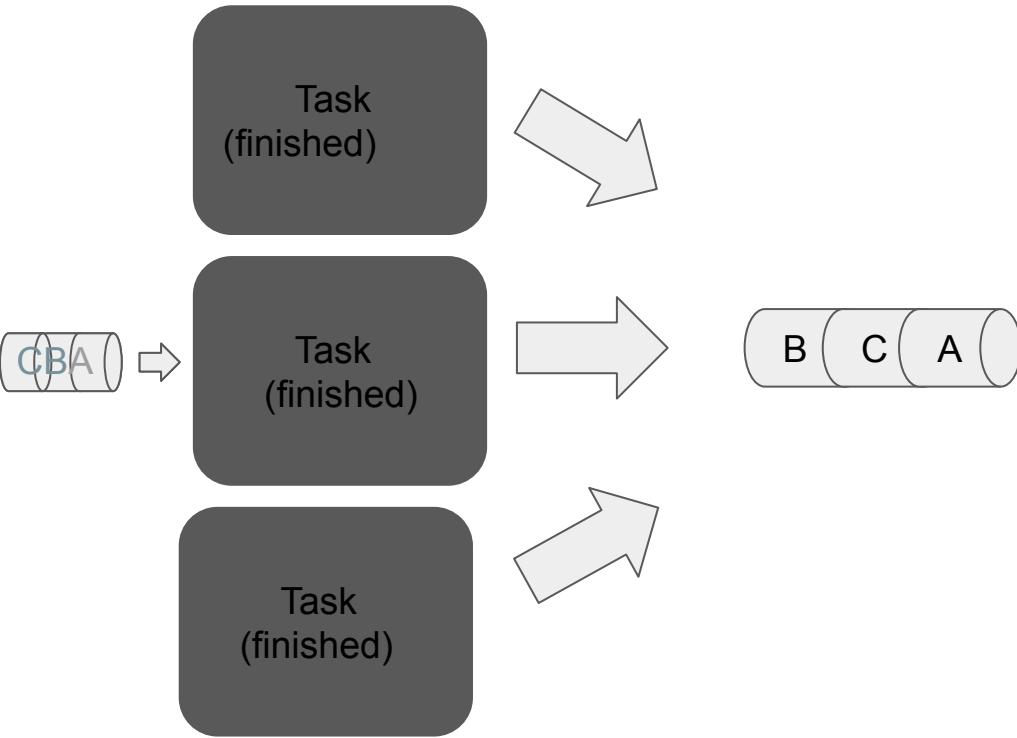
Channel: Properties: Asynchronous

Outputs from a set of processes will not necessarily be produced in the same order as the corresponding inputs went in.



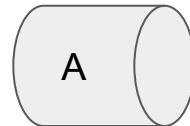
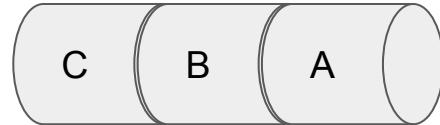
Channel: Properties: Asynchronous

Outputs from a set of processes will not necessarily be produced in the same order as the corresponding inputs went in.



Channels: Channel Types

- Queue Channel
- Value Channel



Queue Channels



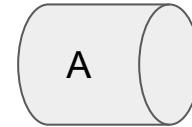
- Can store multiple items
- Including
 - Values
 - Strings 'ch1','chr2','chr3'
 - Numbers 12,14,22
 - Boolean true,false
 - Lists
 - Multiple values per item [12,14],[22,3]
 - Files
 - ref1_1.fq.gz

Queue Channels



- Data is consumed (used up) by a process

Value Channels



- A value channel is bound to a single value.
- Including

- Values
 - Strings 'ch1'
 - Numbers 12
 - Boolean true or false
- List of Values ['chr1','chr2']

Value Channels



A value channel can be used an unlimited number times since its content is not consumed.

Exercise

Queue vs Value Channel.

What type of channel would you use to store the following?

1. Multiple values.
2. A list with one or more values.
3. A single value.

Channels: Creating channels

1. Explicitly with Channel factories, methods to create channels
2. As the output of a process (covered in processes episode)

Channels: Channel Factory, creating channels

Channel factories are used to explicitly create channels

```
Channel.<method>
```

Channels: The value Channel factory

The `value` factory method is used to create a value channel. Values are put inside parentheses `()` to assign them to a channel.

```
ch1 = Channel.value( 'GRCh38' )
```

1. Creates a value channel and binds a string to it.

Channels: The value Channel factory

The `value` factory method is used to create a value channel. Values are put inside parentheses `()` to assign them to a channel.

```
ch1 = Channel.value( 'GRCh38' )
ch2 = Channel.value( ['chr1', 'chr2', 'chr3', 'chr4', 'chr5'] )
```

1. Creates a value channel and binds a string to it.
2. Creates a value channel and binds a list object to it that will be emitted as a single item.

Channels: The value Channel factory

The `value` factory method is used to create a value channel. Values are put inside parentheses `()` to assign them to a channel.

```
ch1 = Channel.value( 'GRCh38' )
ch2 = Channel.value( ['chr1', 'chr2', 'chr3', 'chr4', 'chr5'] )
```

1. Creates a value channel and binds a string to it.
2. Creates a value channel and binds a list object to it that will be emitted as a single item.

To view the contents we can use the `.view()` operator will print one line per item in a channel.

Channel: Queue channel factory methods

- `Channel.of()`

Channels: The of Channel factory (multiple values)

When you want to create a channel containing multiple values you can use the channel factory `Channel.of`.

`Channel.of` allows the creation of a `queue` channel with the values specified as arguments, separated by a `,`.

```
chromosome_ch = Channel.of( 'chr1', 'chr3', 'chr5', 'chr7' )
```

Channels: The of Channel factory (multiple values)

When you want to create a channel containing multiple values you can use the channel factory `Channel.of`.

`Channel.of` allows the creation of a `queue` channel with the values specified as arguments, separated by a `,`.

```
chromosome_ch = Channel.of( 'chr1', 'chr3', 'chr5', 'chr7' )  
chromosome_ch.view()
```

The `view` operator will print one line per item in a channel. Therefore the `view` operator on will print four lines, one for each element in the channel:

Channels: The of Channel factory (ranges) ...

You can specify a range of numbers as a single argument using the Groovy range operator ...

Code

```
chromosome_ch = Channel.of(1..22, 'X', 'Y')  
chromosome_ch.view()
```

Exercise: channel.nf

Create a value and Queue and view Channel contents

1. Create a Nextflow script file called `channel.nf` .
2. Create a Value channel `ch_vl` containing the String `'GRCh38'` .
3. Create a Queue channel `ch_qu` containing the values 1 to 4.
4. Use `.view()` operator on the channel objects to view the contents of the channels.
5. Run the code using

Code

```
$ nextflow run channel.nf
```

Channel: Queue channel factory methods

- Channel.of
- Channel.fromList

Channels: `fromList` Channel factory

Convert a List [1, 2, 3] into a multi-item queue channel

```
aligner_list = ['salmon', 'kallisto']

aligner_ch = Channel.fromList(aligner_list)

aligner_ch.view()
```

Exercise: channel_list.nf

Creating channels from a list

Write a Nextflow script that creates both a `queue` and `value` channel for the list

Code

```
ids = ['ERR908507', 'ERR908506', 'ERR908505']
```

Then print the contents of the channels using the `view` operator. How many lines does the queue and value channel print?

Hint: Use the `fromList()` and `value()` Channel factory methods.

Channels: Queue channel factory methods

- Channel.of
- Channel.fromList
- Channel.fromPath

Channels: `fromPath` Channel factory

A special channel factory method `fromPath` is used when wanting to pass files.

```
read_ch = Channel.fromPath( 'data/yeast/reads/ref1_2.fq.gz' )  
read_ch.view()
```



Path to file from where you run script

Channels: fromPath Channel factory: Multiple Files

You can also use Bash glob syntax to specify pattern-matching behaviour for files.

- An asterisk, `*`, matches any number of characters (including none).
- Two asterisks, `**`, works like `*` but will also search sub directories.
- Braces `{ }` specify a collection of subpatterns. For example: `{bam,bai}` matches “bam” or “bai”

Channels: The fromPath Channel factory: Multiple files

```
read_ch = Channel.fromPath( 'data/yeast/reads/*.fq.gz' )  
read_ch.view()
```

Channels: fromPath Channel factory: Behaviour

You can change the behaviour of `channel.fromPath` method by changing its options. A list of `.fromPath` options is shown below.

Name	Description
glob	When true, the characters <code>*</code> , <code>?</code> , <code>[]</code> and <code>{}</code> are interpreted as glob wildcards, otherwise they are treated as literal characters (default: true)
type	The type of file paths matched by the string, either <code>file</code> , <code>dir</code> or <code>any</code> (default: file)
hidden	When true, hidden files are included in the resulting paths (default: false)
maxDepth	Maximum number of directory levels to visit (default: no limit)
followLinks	When true, symbolic links are followed during directory tree traversal, otherwise they are managed as files (default: true)
relative	When true returned paths are relative to the top-most common directory (default: false)
checkIfExists	When true throws an exception if the specified path does not exist in the file system (default: false)

```
Channel.fromPath('data/yeast/*', hidden: true, checkIfExists: true)
```

Channels: fromPath Channel factory: Behaviour

Code

```
read_ch = Channel.fromPath( 'data/chicken/reads/*.fq.gz' )  
read_ch.view()
```

```
read_ch = Channel.fromPath( 'data/chicken/reads/*.fq.gz', checkIfExists: true )  
read_ch.view()
```

Exercise: channel_fromPath.nf

Using Channel.fromPath

1. Create a Nextflow script `channel_fromPath.nf`
2. Use the `Channel.fromPath` method to create a channel containing all files in the `data/yeast/` directory, including the subdirectories.
3. Add the parameter to include any hidden files.
4. Then print all file names using the `view` operator.

Hint: You need two asterisks, i.e. `**`, to search subdirectories.

Channel: Queue channel factory methods

- Channel.of
- Channel.fromList
- Channel.fromPath
- Channel.fromFilePairs

Channels: fromFilePairs Channel factory

We have seen how to process files using `fromPath`. In Bioinformatics we often want to process files in pairs or larger groups, such as read pairs in sequencing.

Sample group	read1	read2
ref1	data/yeast/reads/ref1_1.fq.gz	data/yeast/reads/ref1_2.fq.gz
ref2	data/yeast/reads/ref2_1.fq.gz	data/yeast/reads/ref2_2.fq.gz
ref3	data/yeast/reads/ref3_1.fq.gz	data/yeast/reads/ref3_2.fq.gz

Channels: **fromFilePairs** Channel factory

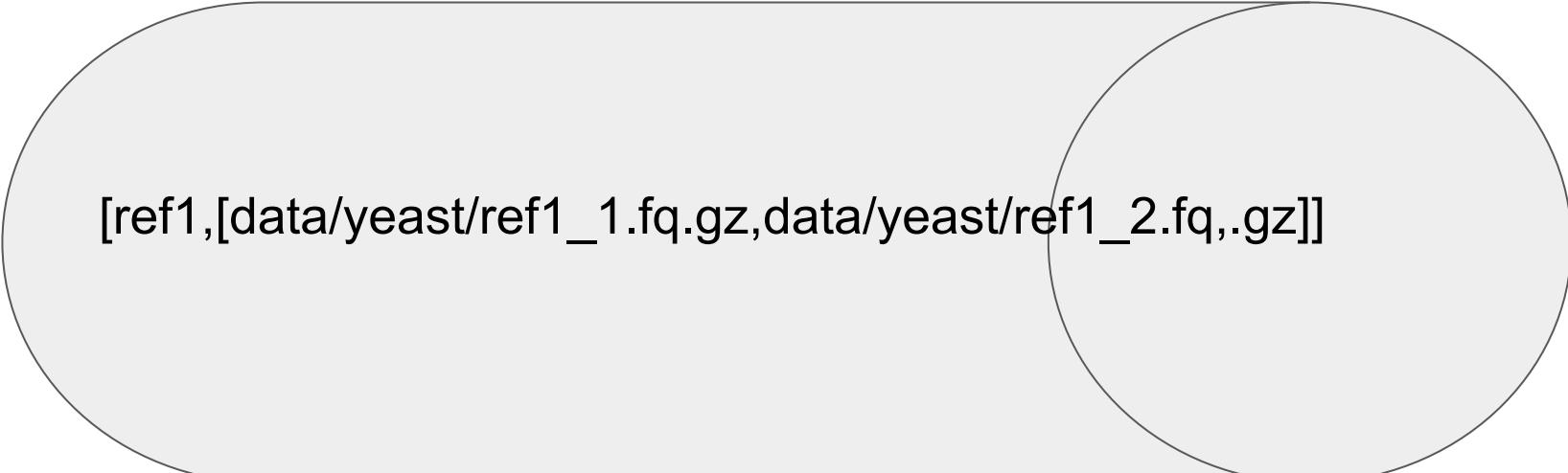
Sample group	read1	read2
ref1	data/yeast/reads/ref1_1.fq.gz	data/yeast/reads/ref1_2.fq.gz

```
Channel.fromFilePairs("data/yeast/ref1_{1,2}.fq.gz")
```

Channels: **fromFilePairs** Channel factory

Sample group	read1	read2
ref1	data/yeast/reads/ref1_1.fq.gz	data/yeast/reads/ref1_2.fq.gz

```
Channel.fromFilePairs("data/yeast/ref1*{1,2}.fq.gz")
```



[ref1,[data/yeast/ref1_1.fq.gz,data/yeast/ref1_2.fq.gz]]

Channels: `fromFilePairs` Tuple

```
Channel.fromFilePairs("data/yeast/ref1*{1,2}.fq.gz")
```

```
[ref1,[data/yeast/ref1_1.fq.gz,data/yeast/ref1_2.fq.gz]]
```

A `tuple` is a grouping of data, represented as a Groovy List.

1. The first element of the tuple emitted from `fromFilePairs` is a string based on the *shared* part of the filenames .

Note: Underscore seems to be ignored

Channels: `fromFilePairs` Tuple

```
Channel.fromFilePairs("data/yeast/ref1*{1,2}.fq.gz")  
[ref1,[data/yeast/ref1_1.fq.gz,data/yeast/ref1_2.fq.gz]]
```

A `tuple` is a grouping of data, represented as a Groovy List.

1. The first element of the tuple emitted from `fromFilePairs` is a string based on the shared part of the filenames (i.e., the `*` part of the glob pattern).
2. The second element is the list of files matching the remaining part of the glob pattern (i.e., the `<string>_{1,2}.fq.gz` pattern). This will include any files ending `_1.fq.gz` or `_2.fq.gz`.

Channels: `fromFilePairs`

```
read_pair_ch = Channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz')
read_pair_ch.view()
```

Channels: fromFilePairs: Options

Changing the behaviour of fromFilePairs

Table of optional parameters available:

Name	Description
type	Type of paths returned, either <code>file</code> , <code>dir</code> or <code>any</code> (default: <code>file</code>)
hidden	When <code>true</code> includes hidden files in the resulting paths (default: <code>false</code>)
maxDepth	Maximum number of directory levels to visit (default: <i>no limit</i>)
followLinks	When <code>true</code> it follows symbolic links during directories tree traversal, otherwise they are managed as files (default: <code>true</code>)
size	Defines the number of files each emitted item is expected to hold (default: 2). Set to <code>-1</code> for any.
flat	When <code>true</code> the matching files are produced as sole elements in the emitted tuples (default: <code>false</code>).
checkIfExists	When <code>true</code> throws an exception if the specified path do not exist in the file system (default: <code>false</code>)

<https://www.nextflow.io/docs/latest/channel.html#fromfilepairs>

Channels: fromFilePairs:size

If you want to capture more than two files for a pattern you will need to change the default `size` argument (the default value is 2) to the number of expected matching files.

Code

```
read_group_ch = Channel.fromFilePairs('data/yeast/reads/ref{1,2,3}*', size:6)
read_group_ch.view()
```

Exercise: channel_fromFilePairs.nf

Create a channel containing groups of files

1. Create a Nextflow script file `channel_fromFilePairs.nf` .
2. Use the `fromFilePairs` method to create a channel containing three tuples. Each tuple will contain the pairs of fastq reads for the three temp33 samples in the `data/yeast/reads` directory

```
$ ls data/yeast/reads/temp33*
data/yeast/reads/temp33_1_1.fq.gz      data/yeast/reads/temp33_2_1.fq.gz      data/yeast/reads/temp33_3_1.fq.gz
data/yeast/reads/temp33_1_2.fq.gz      data/yeast/reads/temp33_2_2.fq.gz      data/yeast/reads/temp33_3_2.fq.gz
```

Channels: Key points

- Channels must be used to import data into Nextflow.

Channels: Key points

- Channels must be used to import data into Nextflow.
- Nextflow has two different kinds of channels: queue channels and value channels.

Channels: Key points

- Channels must be used to import data into Nextflow.
- Nextflow has two different kinds of channels: queue channels and value channels.
- Data in value channels can be used multiple times in workflow.

Channels: Key points

- Channels must be used to import data into Nextflow.
- Nextflow has two different kinds of channels: queue channels and value channels.
- Data in value channels can be used multiple times in workflow.
- Data in queue channels are consumed when they are used by a process or an operator.

Channels: Key points

- Channels must be used to import data into Nextflow.
- Nextflow has two different kinds of channels: queue channels and value channels.
- Data in value channels can be used multiple times in workflow.
- Data in queue channels are consumed when they are used by a process or an operator.
- Channel factory methods, such as `Channel.of`, are used to create channels.

Channels: Key points

- Channels must be used to import data into Nextflow.
- Nextflow has two different kinds of channels: queue channels and value channels.
- Data in value channels can be used multiple times in workflow.
- Data in queue channels are consumed when they are used by a process or an operator.
- Channel factory methods, such as `Channel.of`, are used to create channels.
- Channel factory methods have optional parameters e.g., `checkIfExists`, that can be used to alter the creation and behaviour of a channel.

Lunch

Processes Part 1

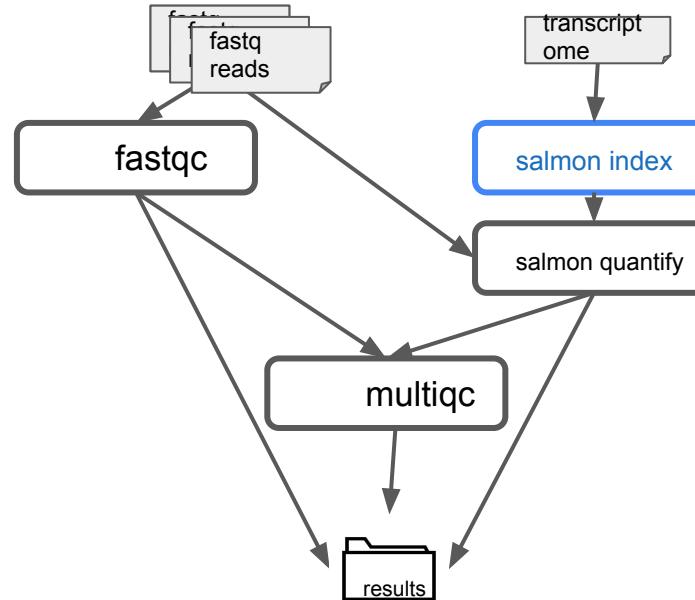
<https://carpentries-incubator.github.io/workflows-nextflow/05-processes-part1>

Question

- How do I run tasks/processes in Nextflow?
- How do I get data, files and values, into a processes?

Process

- A process is the way Nextflow executes commands you would run on the command line or custom scripts.
- A process is independent (can't call process from another process)
- Data is passed via channels. (Think of communication channel)



Process

Command we would type in terminal

```
$ salmon index -t data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \
-i data/yeast/salmon_index \
--kmerLen 31
```

Process: definition

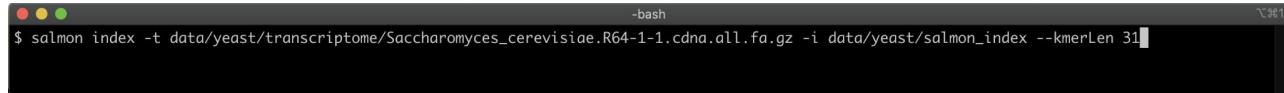


A screenshot of a terminal window titled '-bash'. The command entered is '\$ salmon index -t data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz -i data/yeast/salmon_index --kmerLen 31'.

keyword
→

```
process INDEX {
    script:
        "salmon index -t
            data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \
            -i data/yeast/salmon_index \
            --kmerLen 31"
}
```

Process: definition



```
process INDEX {
    script:
        """
            salmon index -t
            data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \
            -i data/yeast/salmon_index \
            --kmerLen 31
        """
}
```

process name

An arrow points from the word 'INDEX' in the first line of the code to the word 'INDEX' in the terminal window above.

Process: definition: Process body



```
-bash
$ salmon index -t data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz -i data/yeast/salmon_index --kmerLen 31
```

```
process INDEX { ← Curly Brackets {} delimits body of process
  script:
    """
      salmon index -t
      data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \
      -i data/yeast/salmon_index \
      --kmerLen 31
    """
}
```

Process: definition: Script directive



```
process INDEX {  
    script: Script block  
        """  
        salmon index -t  
        data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \  
        -i data/yeast/salmon_index \  
        --kmerLen 31  
        """  
}
```

Process: Calling a Process

```
//process_index.nf

nextflow.enable.dsl=2

process INDEX {
    script:
        "salmon index -t ${projectDir}/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz
         -i data/yeast/salmon_index --kmerLen 31"
    }

workflow {
    //process is called like a function in the workflow block
    INDEX()
}

}

```

To call the process in a workflow add a `workflow` block, and call the process like a function. We will cover the workflow block in another episode.

Exercise: simple_process.nf

A Simple Process

Create a Nextflow script `simple_process.nf` that has one process `SALMON_VERSION` that runs the command.

Bash

```
salmon --version
```

```
nextflow.enable.dsl=2

process NAME {
    script:
}

workflow { }
```

Process: Definition blocks

To control inputs, outputs and how a command is executed a process may contain five definition blocks:

```
process < NAME > {
    [ directives ]
    2 input:
        < process inputs >
    3 output:
        < process outputs >
    4 when:
    5 < condition >
        [script|shell]:
            < user script to be executed >
}
```

Process: script

```
nextflow.enable.dsl=2

process PROCESSBAM {
    script:
        "samtools sort -o ref1.sorted.bam ${projectDir}/data/yeast/bams/ref1.bam"
}

workflow {
    PROCESSBAM()
}
```

Single quotes for one line command

At minimum a process block must contain a `script/shell/exec` block.

Process: script: multiline

```
//process_multi_line.nf
nextflow.enable.dsl=2

process PROCESSBAM {
    script:
    """
        samtools sort -o ref1.sorted.bam ${projectDir}/data/yeast/bams/ref1.bam
        samtools index ref1.sorted.bam
        samtools flagstat ref1.sorted.bam
    """
}

workflow {
    PROCESSBAM()
}
```

Triple quotes for multi line commands



Process: script: Other languages

By default the process command is interpreted as a **Bash** script. However any other scripting language can be used just simply starting the script with the corresponding **Shebang** declaration. For example:

```
//process_python.nf
nextflow.enable.dsl=2

process PYSTUFF {
    script:
    """
    #!/usr/bin/env python
    import gzip

    reads = 0
    bases = 0

    with gzip.open('${projectDir}/data/yeast/reads/ref1_1.fq.gz', 'rb') as read:
        for id in read:
            seq = next(read)
            reads += 1
            bases += len(seq.strip())
            next(read)
            next(read)

    print("reads", reads)
    print("bases", bases)
    """
}

workflow {
    PYSTUFF()
}
```

```
//process_rscript.nf
nextflow.enable.dsl=2

process RSTUFF {
    script:
    """
    #!/usr/bin/env Rscript
    library("ShortRead")
    countFastq(dirPath="data/yeast/reads/ref1_1.fq.gz")
    """
}

workflow {
    RSTUFF()
}
```

Process: script: other languages

However, for large chunks of code it is suggested to save them into separate files and invoke them from the process script.

```
nextflow.enable.dsl=2

process PYSTUFF {

    script:
    """
    myscript.py
    """

}

workflow {
    PYSTUFF()
}
```

Nextflow will automatically add the directory bin into the `PATH` environmental variable. So therefore any executable in the bin folder of a Nextflow pipeline can be called without the need to reference the full path.

Process: script: parameters

The command in the `script` block can be defined dynamically using Nextflow variables

```
//process_script.nf
nextflow.enable.dsl=2

kmer = 31           ← Nextflow variable

process INDEX {

    script:
    """
    salmon index \
    -t $projectDir/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \
    -i index \
    --kmer $kmer
    echo "kmer size is $kmer"
    """

}

workflow {
    INDEX()
}
```

Process: script: parameters

The command in the `script` block can be defined dynamically using Nextflow variables

```
//process_script.nf
nextflow.enable.dsl=2

kmer = 31
          ← Nextflow variable

process INDEX {

    script:
    """
    salmon index \
    -t $projectDir/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \
    -i index \
    --kmer $kmer
    echo "kmer size is $kmer"
    """

}

workflow {
    INDEX()
}
```

Process: Script parameters: Dynamically using Params

Nextflow variable `params` that can be used to assign values from the command line. You would do this by adding a key name to the `params` variable and specifying a value, like `params.keyname = value`

```
//process_script_params.nf
nextflow.enable.dsl=2

params.kmer = 31 ← Nextflow params variable

process INDEX {

    script:
    """
    salmon index \
    -t $projectDir/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \
    -i index \
    --kmer $params.kmer
    echo "kmer size is $params.kmer"
    """

}

workflow {
    INDEX()
}
```

Process: Script parameters: Dynamically using Params

Nextflow variable `params` that can be used to assign values from the command line. You would do this by adding a key name to the `params` variable and specifying a value, like `params.keyname = value`

```
//process_script_params.nf
nextflow.enable.dsl=2

params.kmer = 31
process INDEX {
    script:
    """
    salmon index \
    -t $projectDir/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz \
    -i index \
    --kmer $params.kmer
    echo "kmer size is $params.kmer"
    """
}
workflow {
    INDEX()
}
```

The diagram illustrates the assignment of a Nextflow parameter to a script variable. An arrow points from the line `params.kmer = 31` to the variable `$params.<variable_name>`. Another arrow points from the same line to the script parameter `--kmer $params.kmer`.

Exercise: process_script_params.nf

Script parameters

For the Nextflow script below.

Code

```
//process_script_params.nf
nextflow.enable.dsl=2
params.kmer = 31

process INDEX {

    script:
    """
    salmon index -t $projectDir/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz -i index --kmer $params.kmer
    echo "kmer size is" $params.kmer
    """
}

workflow {
    INDEX()
}
```

Run the pipeline using a kmer value of 27 using the `--kmer` command line option.

Bash

```
$ nextflow run process_script_params.nf --kmer <some value> -process.echo
```

Note: The Nextflow option `-process.echo` will print the process' stdout to the terminal.

Process: Bash Variable

- Nextflow uses the same Bash syntax for variable substitutions \$
- We need to escape bash variables using \

```
//process_escape_bash.nf
nextflow.enable.dsl=2

process INDEX {
    script:
    """
    #set bash variable KMERSIZE
    KMERSIZE=$params.kmer
    salmon index -t $projectDir/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz -i index --kmer \$KMERSIZE
    echo "kmer size is $params.kmer"
    """
}

params.kmer = 31

workflow {
    INDEX()
}
```

Set bash variable

Reference bash variable escape /



Process: Shell directive

You can use shell directive instead of script to use Bash variables without escaping

```
//process_shell.nf
nextflow.enable.dsl=2

params.kmer = 31

process INDEX {
    shell: ...
    #set bash variable KMERSIZE
    KMERSIZE=!{params.kmer}
    salmon index -t !{projectDir}/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz -i index --kmer ${KMERSIZE}
    echo "kmer size is !{params.kmer}"
    ...
}

workflow {
    INDEX()
}
```

shell directive

Set bash variable, escape Nextflow variable
! {nextflow_variable}

Bash variable no escaping

```
graph LR; A[shell:] --> B["shell directive"]; C[! {nextflow_variable}] --> D["Set bash variable, escape Nextflow variable"]; E[$] --> F["Bash variable no escaping"]
```

Process: script: Conditional execution

Sometimes you want to change how a process is run depending on some condition.

In Nextflow scripts we can use conditional statements such as the `if` statement or any other expression evaluating to boolean value `true` or `false`.

```
if( < boolean expression > ) {
    // true branch
}
else if ( < boolean expression > ) {
    // true branch
}
else {
    // false branch
}
```

Process: script: Conditional execution

Different script runs dependant on the outcome of conditional statement

```
//process_conditional.nf
nextflow.enable.dsl=2

params.aligner = 'kallisto'
params.transcriptome = "$projectDir/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz"
params.kmer = 31

process INDEX {
    script:
        if( params.aligner == 'kallisto' ) {
            echo indexed using kallisto
            kallisto index -i index -k $params.kmer $params.transcriptome
        }
        else if( params.aligner == 'salmon' ) {
            echo indexed using salmon
            salmon index -t $params.transcriptome -i index --kmer $params.kmer
        }
        else {
            echo Unknown aligner $params.aligner
        }
    }

workflow {
    INDEX()
}
```

Process: script: Conditional execution

Different script runs dependant on the outcome of conditional statement

```
$ nextflow run process_conditional.nf -process.echo --aligner kallisto
```

```
//process_conditional.nf
nextflow.enable.dsl=2

params.aligner = 'kallisto'
params.transcriptome = "$projectDir/data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz"
params.kmer = 31

process INDEX {
    script:
        if( params.aligner == 'kallisto' ) {
            """
            echo indexed using kallisto
            kallisto index -i index -k $params.kmer $params.transcriptome
            """
        }
        else if( params.aligner == 'salmon' ) {
            """
            echo indexed using salmon
            salmon index -t $params.transcriptome -i index --kmer $params.kmer
            """
        }
        else {
            """
            echo Unknown aligner $params.aligner
            """
        }
}

workflow {
    INDEX()
}
```

Process: Input: Getting data into a process

So far we have put paths into our script to specify the data.

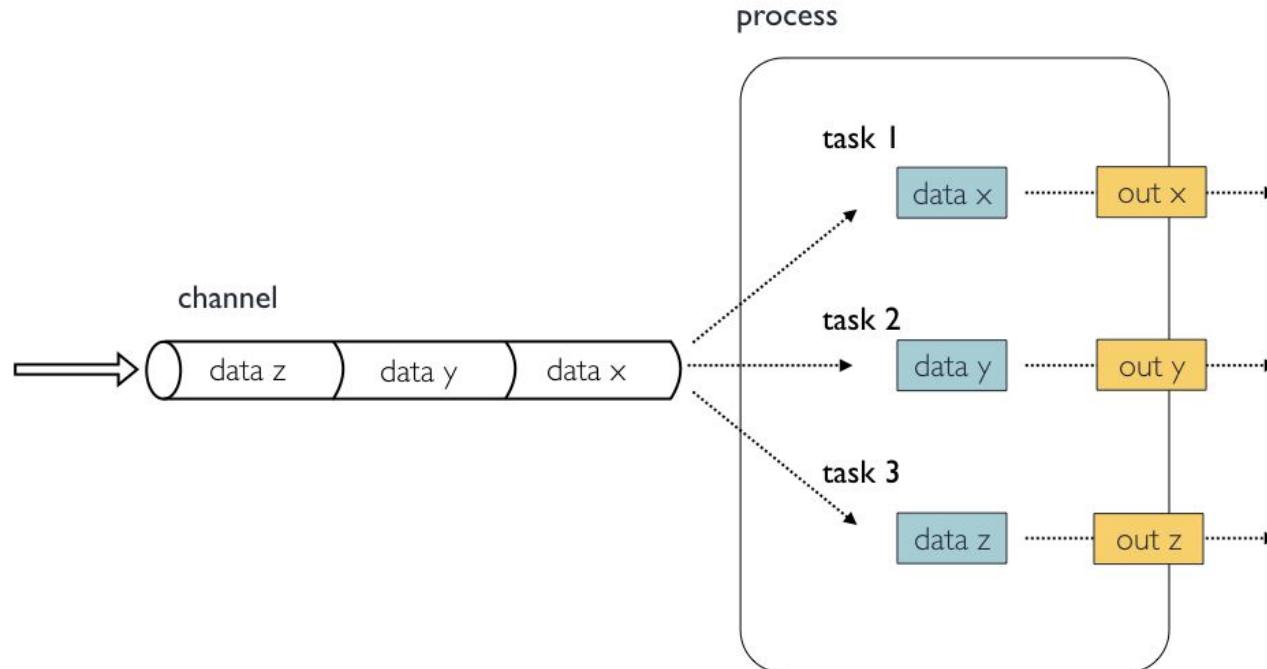
We should use input channels to manage data

```
process < NAME > {
    [ directives ]
    input:
    < process inputs >
    output:
    < process outputs >
    when:
    < condition >
    [script|shell|exec]:
    < user script to be executed >
}
```

Process: Input: Channel recap

Inputs to a process should be defined using Nextflow channels

The number of item in the input channel determines how many times a process will run



Process: Input: Defining inputs

To define input to a process we use keyword `input:` followed by the input qualifier and a name

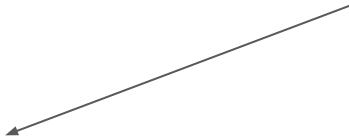
```
input:  
  <input qualifier> <input name>
```

The input qualifier assigns the data type being received.

Process: Input: Qualifiers

The input qualifier `val` defines a value data type to be received as input.

```
input:  
  <input qualifier> <input name>
```



- `val`: value, number, string, boolean.

Process: Input: val

The value can be accessed in the process script by using the specified input name

```
//process_input_value.nf
nextflow.enable.dsl=2

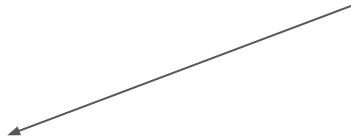
process PRINTCHR {
    input:
        val chr
    script:
        echo processing chromosome $chr
    }
    chr_ch = Channel.of( 1..22, 'X', 'Y' )
}

workflow {
    PRINTCHR(chr_ch)
}
```

Process: Input: File

The input qualifier `path` declares a file as the data to be received.

```
input:  
  <input qualifier> <input name>
```



- `val`: value, number, string, boolean.
- `path`: a file,

Process: Input: File Dynamic Naming

The input file name can be defined dynamically by defining the input name as a Nextflow variable and referenced in the script using the `$variable_name` syntax.

```
//process_input_file.nf
nextflow.enable.dsl=2

process NUMLINES {
    input:
        path read
    script:
        """
        printf '${read} '
        gunzip -c ${read} | wc -l
        """
    }
}

reads_ch = Channel.fromPath( 'data/yeast/reads/ref*.fq.gz' )

workflow {
    NUMLINES(reads_ch)
}
```

Process: Input: File Dynamic Naming

The input file name can be defined dynamically by defining the input name as a Nextflow variable and referenced in the script using the `$variable_name` syntax.

```
//process_input_file.nf
nextflow.enable.dsl=2

process NUMLINES {
    input:
        path read
    script:
        """
        printf '${read} '
        gunzip -c ${read} | wc -l
        """
    }
}

reads_ch = Channel.fromPath( 'data/yeast/reads/ref*.fq.gz' )

workflow {
    NUMLINES(reads_ch)
}
```

Process: Input: Named File

The input name can also be defined as user specified filename inside quotes.

```
//process_input_file_02.nf
nextflow.enable.dsl=2

process NUMLINES {
    input:
    path 'sample.fq.gz'

    script:
    """
    printf 'sample.fq.gz '
    gunzip -c sample.fq.gz | wc -l
    """

}

reads_ch = Channel.fromPath( 'data/yeast/reads/ref*.fq.gz' )

workflow {
    NUMLINES(reads_ch)
}
```

Exercise: process_exercise_input.nf

Add input channel

Add an input channel to the script below that takes the reads channel as input. [FastQC](#) is a quality control tool for high throughput sequence data.

Code

```
//process_exercise_input.nf
nextflow.enable.dsl=2

process FASTQC {
    //add input channel

    script:
    """
    mkdir fastqc_out
    fastqc -o fastqc_out ${reads}
    ls -1 fastqc_out
    """
}

reads_ch = Channel.fromPath( 'data/yeast/reads/ref1*_1,2.fq.gz' )

workflow {
    FASTQC(reads_ch)
}
```

Run using

```
$ nextflow run process_exercise_input.nf -process.echo
```

What input qualifier do you need to capture a file?

What name do you need to give the input variable to match the file referenced in the script block?

Process: Input: Multiple inputs

A key feature of processes is the ability to handle inputs from multiple channels.

To specify multiple inputs we add another qualifier and input name

```
input:  
    <input qualifier> <input name>  
    <input qualifier> <input name>  
    <input qualifier> <input name>  
    ...
```

```
input:  
    val kmer  
    path transcriptome  
    ...
```

Process: Input: Combining input channels

However it's important to understand how the number of items within the multiple channels affect the execution of a process.

```
//process_combine.nf
nextflow.enable.dsl=2
```

```
process COMBINE {
    input:
    val x
    val y
```

```
    script:
    """
        echo $x and $y
    """
}
```

```
num_ch = Channel.of(1, 2, 3)
letters_ch = Channel.of('a', 'b', 'c')
```

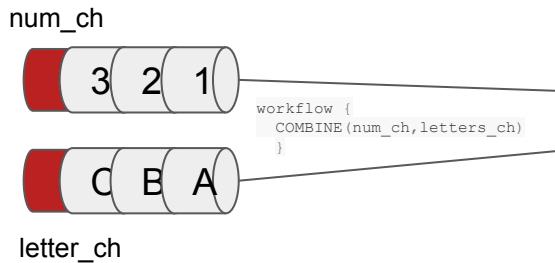
```
workflow {
    COMBINE(num_ch, letters_ch)
}
```

2 value inputs

2 queue channels

Process: Input: Combining input channels

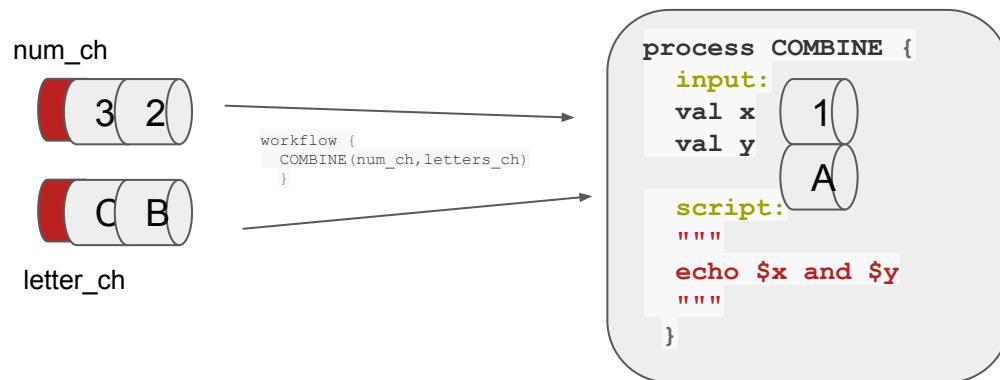
However it's important to understand how the number of items within the multiple channels affect the execution of a process.



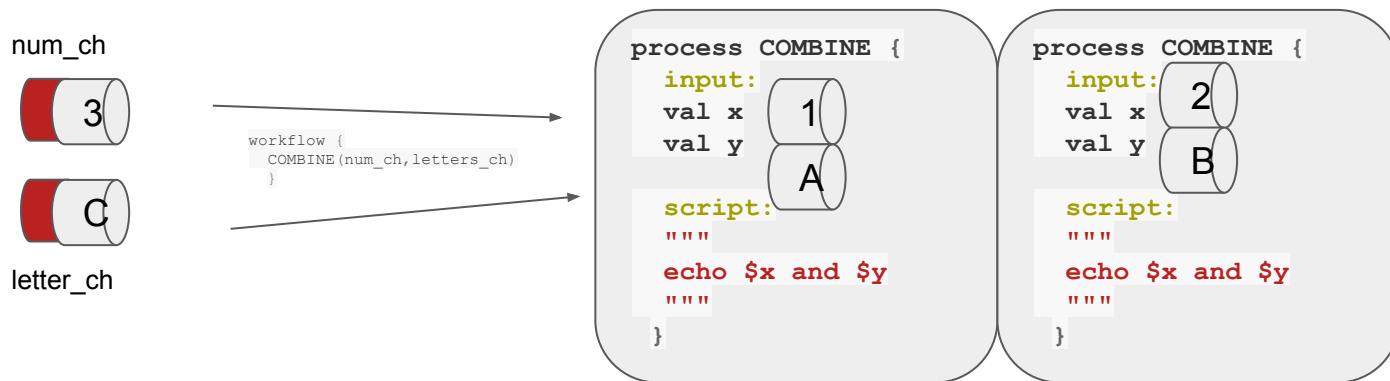
```
process COMBINE {  
    input:  
    val x  
    val y  
  
    script:  
    """  
    echo $x and $y  
    """  
}
```

```
//process_combine.nf  
nextflow.enable.dsl=2  
  
process COMBINE {  
    input:  
    val x  
    val y  
  
    script:  
    """  
    echo $x and $y  
    """  
}  
  
num_ch = Channel.of(1, 2, 3)  
letters_ch = Channel.of('a', 'b', 'c')  
  
workflow {  
    COMBINE(num_ch, letters_ch)  
}
```

Process: Input: Combining input channels



Process: Input: Combining input channels



Process: Input: Combining input channels

STOP

\$ STOP


```
workflow {  
    COMBINE(num_ch,letters_ch)  
}
```

```
process COMBINE {  
    input:  
        val x 1  
        val y A  
  
    script:  
        """  
        echo $x and $y  
        """  
}
```

```
process COMBINE {  
    input:  
        val x 2  
        val y B  
  
    script:  
        """  
        echo $x and $y  
        """  
}
```

```
process COMBINE {  
    input:  
        val x 3  
        val y C  
  
    script:  
        """  
        echo $x and $y  
        """  
}
```

```
$ nextflow run process_combine.nf -process.echo
```

Process: Input: Combining input channels

What about when the number of items in the input channels don't match

```
//process_combine_02.nf
nextflow.enable.dsl=2

process COMBINE {
    input:
        val x
        val y

    script:
        echo $x and $y
    }

    ch_num = Channel.of(1, 2)
    ch_letters = Channel.of('a', 'b', 'c', 'd')

    workflow {
        COMBINE(ch_num, ch_letters)
    }
}
```

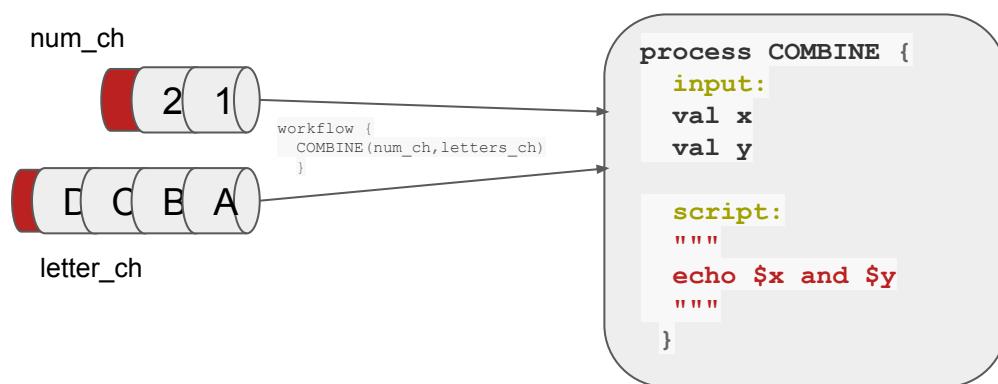
The diagram shows two arrows originating from the 'ch_num' and 'ch_letters' declarations in the code. The first arrow points to the text '2 items in Queue channel'. The second arrow points to the text '4 items in Queue channel'.

2 items in Queue channel

4 items in Queue channel

Process: Input: Combining input channels

What about when the number of items in the input channels don't match



```
//process_combine_02.nf
nextflow.enable.dsl=2

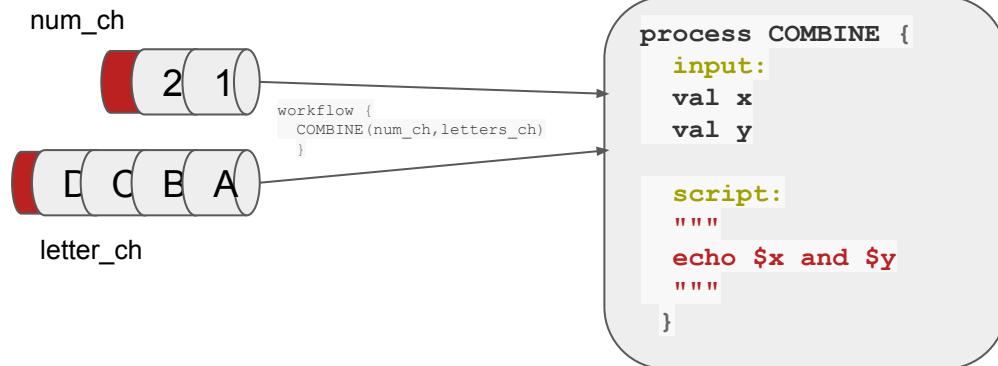
process COMBINE {
    input:
        val x
        val y

    script:
        """
        echo $x and $y
        """
}

ch_num = Channel.of(1, 2)
ch_letters = Channel.of('a', 'b', 'c', 'd')

workflow {
    COMBINE(ch_num, ch_letters)
}
```

Process: Input: Combining input channels



```
//process_combine_02.nf
nextflow.enable.dsl=2

process COMBINE {
    input:
    val x
    val y

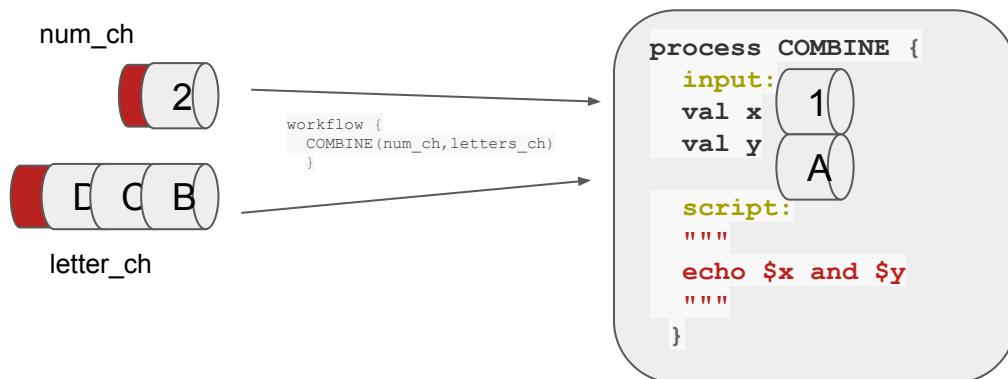
    script:
    """
    echo $x and $y
    """

}

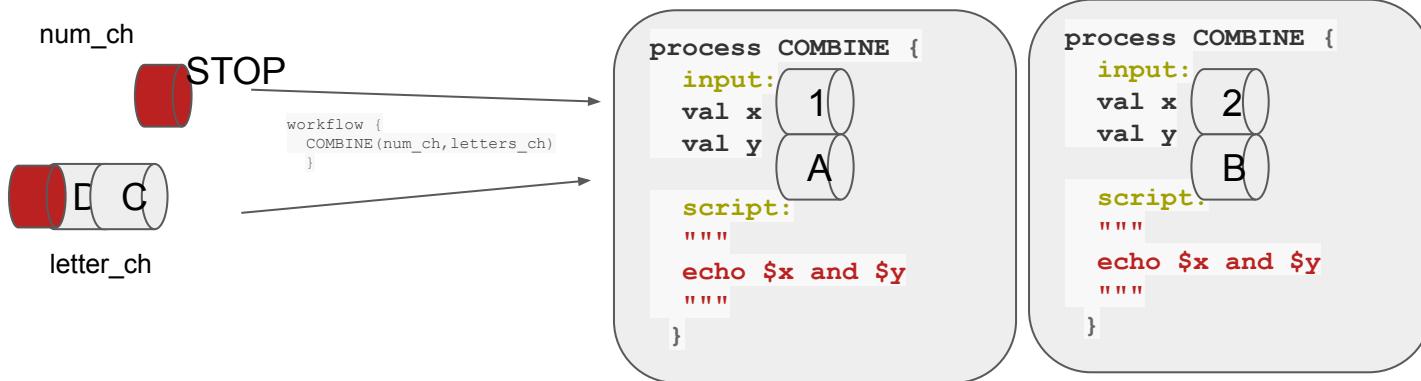
ch_num = Channel.of(1, 2)
ch_letters = Channel.of('a', 'b', 'c', 'd')

workflow {
    COMBINE(ch_num, ch_letters)
}
```

Process: Input: Combining input channels



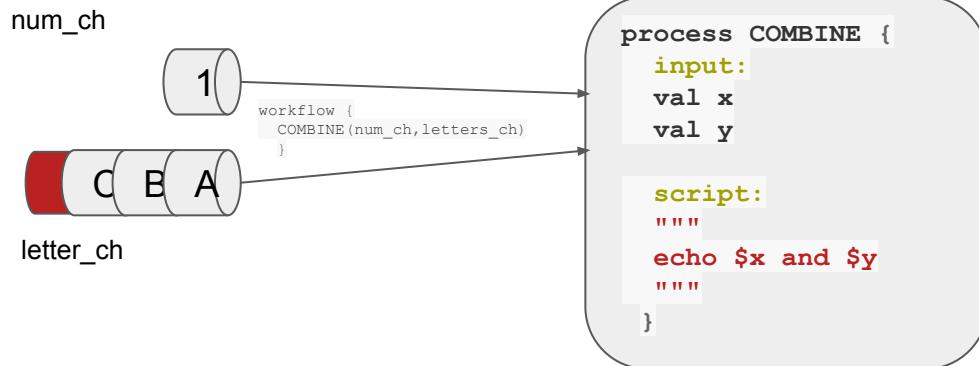
Process: Input: Combining input channels



```
$ nextflow run process_combine_02.nf -process.echo
```

Process: Input: Value channels and process termination

Value channels, `channel.value`, do not affect the process termination.



```
//process_combine_03.nf
nextflow.enable.dsl=2

process COMBINE {
    input:
    val x
    val y

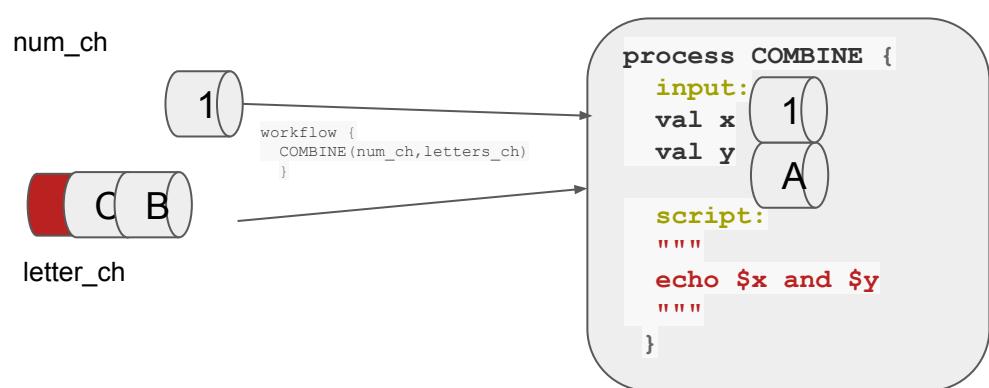
    script:
    """
    echo $x and $y
    """
}

ch_num = Channel.value(1)
ch_letters = Channel.of('a', 'b', 'c')

workflow {
    COMBINE(ch_num, ch_letters)
}
```

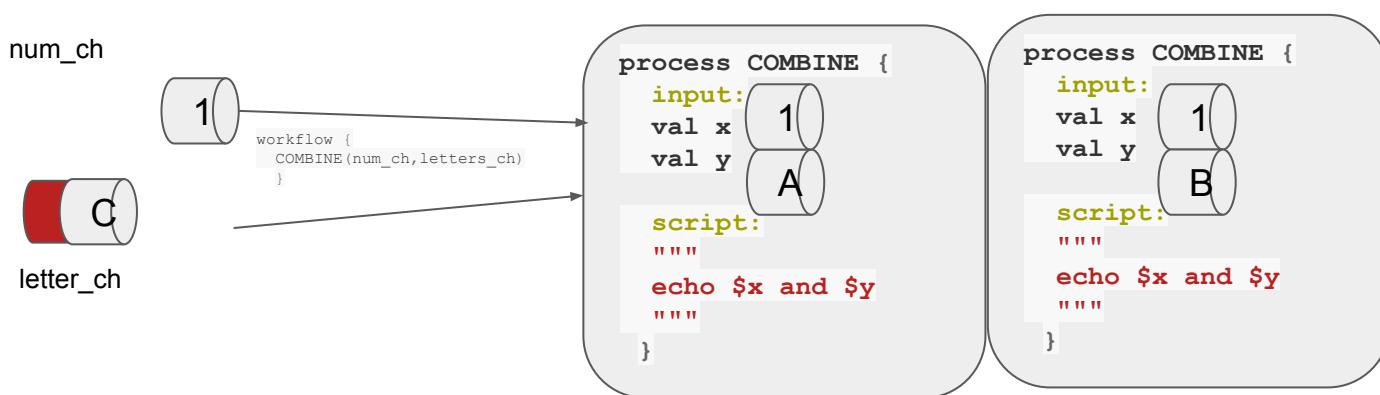
Process: Input: Value channels and process termination

value channels, `channel.value`, do not affect the process termination.



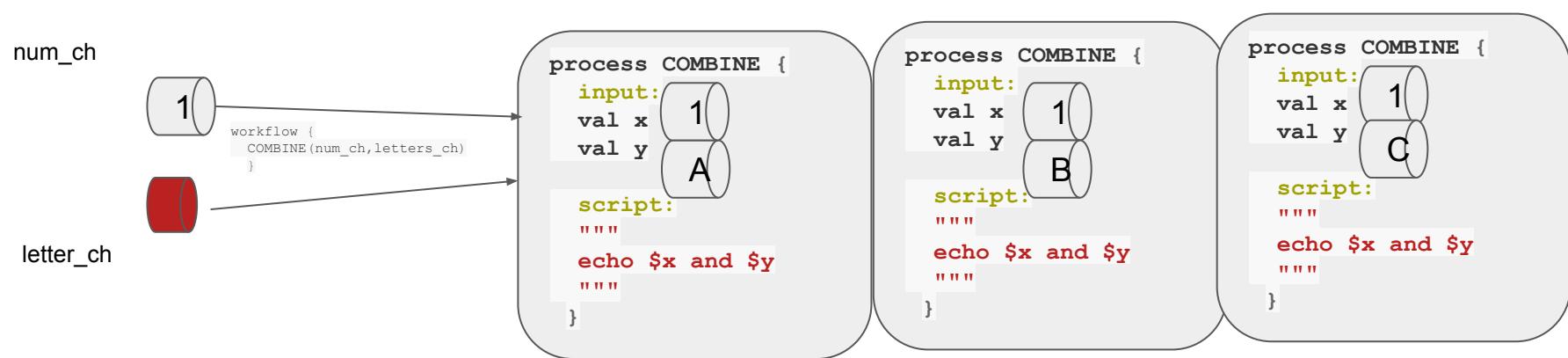
Process: Input: Value channels and process termination

value channels, `channel.value`, do not affect the process termination.



Process: Input: Value channels and process termination

value channels, `channel.value`, do not affect the process termination.



```
$ nextflow run process_combine_03.nf -process.echo
```

Exercise: process_exercise_combine.nf

Combining input channels

Write a nextflow script `process_exercise_combine.nf` that combines two input channels

Code

```
transcriptome_ch = channel.fromPath('data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz')
kmer_ch = channel.of(21)
```

Input should be written in the same order in workflow block (more in workflow episode)

```
COMBINE( transcriptome_ch, kmer_ch )
```

Process: Input: Repeaters

We saw previously that by default the number of times a process runs is defined by the queue channel with the fewest items.

However, the `each` qualifier allows you to repeat the execution of a process for each item in a list or a queue channel, every time new data is received.

For example We can fix the previous example by using the input qualifier `each` for the letters queue channel:

```
//process_repeat.nf
nextflow.enable.dsl=2

process COMBINE {
    input:
        val x
        each y
    script:
        echo $x and $y
    }
}

ch_num = Channel.of(1, 2)
ch_letters = Channel.of('a', 'b', 'c', 'd')

workflow {
    COMBINE(ch_num, ch_letters)
}
```

each qualifier

```
$ nextflow run process_repeat.nf -process.echo
```

Exercise: process_exercise_repeat.nf

Input repeaters

Extend the script `process_exercise_repeat.nf` by adding more values to the `kmer` queue channel e.g. (21, 27, 31) and running the process for each value.

Code

```
//process_exercise_repeat.nf
nextflow.enable.dsl=2
process COMBINE {
    input:
    path transcriptome
    val kmer

    script:
    """
    salmon index -t $transcriptome -i index -k $kmer
    """
}

transcriptome_ch = channel.fromPath('data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz', checkIfExists: true)
kmer_ch = channel.of(21)

workflow {
    COMBINE(transcriptome_ch, kmer_ch)
}
```

How many times does this process run?

Hint: Do the input qualifiers allow repeating?

Process 1: Key Points

- A Nextflow process is an independent step in a workflow

Process 1: Key Points

- A Nextflow process is an independent step in a workflow
- Processes contain up to five definition blocks including:

```
1 process < NAME > {  
2   [ directives ]  
3   input:  
4   < process inputs >  
5   output:  
6   < process outputs >  
7   when:  
8     < condition >  
9     [script|shell]:  
10    < user script to be executed >  
11 }
```

Process 1: Key Points

- A Nextflow process is an independent step in a workflow
- Processes contain up to five definition blocks including:

```
1  process < NAME > {  
2      [ directives ]  
3      input:  
4      < process inputs >  
5      output:  
6      < process outputs >  
7      when:  
8          < condition >  
9          [script|shell]:  
10         < user script to be executed >  
11     }  
12 }
```

- The script block contains the commands you would like to run.

Process 1: Key Points

- A Nextflow process is an independent step in a workflow
- Processes contain up to five definition blocks including:

```
process < NAME > {  
    1   [ directives ]  
    2   input:  
    3   < process inputs >  
    4   output:  
    5   < process outputs >  
    4   when:  
    5   < condition >  
        [script|shell]:  
        < user script to be executed >  
    }  
}
```

- The script block contains the commands you would like to run.
- A process should have a script but the other four blocks are optional

Process 1: Key Points

- A Nextflow process is an independent step in a workflow
- Processes contain up to five definition blocks including:

```
process < NAME > {  
    1   [ directives ]  
    2   input:  
    3   < process inputs >  
    4   output:  
    5   < process outputs >  
    4   when:  
    5   < condition >  
        [script|shell]:  
        < user script to be executed >  
    }  
}
```

- The script block contains the commands you would like to run.
- A process should have a script but the other four blocks are optional
- Inputs are defined in the input block with a type qualifier and a name.

```
input:  
    <input qualifier> <input name>
```

Processes Part 2

Questions

- How do I get data, files, and values, *out* of processes?
- How do I handle grouped input and output?
- How can I control *when* a process is executed?
- How do I control resources, such as number of
 - a. CPUs
 - b. memory, available to processes?
- How do I save output/results from a process?

Process: Outputs

The `output` declaration block allows us to define the channels used by the process to send out the files and values produced.

```
process < NAME > {
    [ directives ]
    input:
    < process inputs >
    output:
    < process outputs >
    when:
    < condition >
    [script|shell|exec]:
    < user script to be
    executed >
}
```

An output block is not required,

Can have multiple outputs

```
output:
<output qualifier> <output name>
<output qualifier> <output name>
...
...
```

Process: Output: Qualifiers

Similarly to process inputs, we need to specify the type of data a process outputs

```
output:  
  <output qualifier> <output name>  
  <output qualifier> <output name>  
  ...
```

The qualifiers that can be used in the output declaration block are the ones listed in the following table:

val	Sends variables with the name specified over the output channel.
path	Sends a file produced by the process with the name specified over the output channel.
env	Sends the variable defined in the process environment with the name specified over the output channel.
stdout	Sends the executed process stdout over the output channel.
tuple	Sends multiple values over the same output channel.

Process: Output: values val

The `val` qualifier allows us to output a value defined in the script.

```
//process_output_value.nf
nextflow.enable.dsl=2

process METHOD {
    input:
    val x
    References input val x

    output:
    val x ← Nextflow value name
    script:
    .....
    echo $x > method.txt
    .....

}

// Both 'Channel' and 'channel' keywords work to generate channels.
// However, it is a good practice to be consistent through the whole pipeline development
methods_ch = channel.of('salmon', 'kallisto')

workflow {
    METHOD(methods_ch)
    // use the view operator to display contents of the channel
    METHOD.out.view({ "Received: $it" })
}
```

Output
qualifier

We can view output
Channel using `view`

Process: Output: file path

If we want to capture a file instead of a value as output we can use the `path` qualifier that can capture one or more files produced by the process, over the specified channel.

```
//process_output_file.nf
nextflow.enable.dsl=2

methods_ch = channel.of('salmon', 'kallisto')

process METHOD {
    input:
    val x

    output:
    path 'method.txt' File name in quotes
    ....
    echo $x > method.txt
    ....
}

workflow {
    METHOD(methods_ch)
    // use the view operator to display contents of the channel
    METHOD.out.view({ "Received: $it" })
}
```

Output
qualifier

Process: Output: Multiple files path

```
//process_output_multiple.nf
nextflow.enable.dsl=2

process FASTQC {
    input:
        path read

    output:
        path "fqc_res/*" ← Output qualifier
        script:
            """
            mkdir fqc_res
            fastqc $read -o fqc_res
            """
}

read_ch = channel.fromPath("data/yeast/reads/ref1*.fq.gz")

workflow {
    FASTQC(read_ch)
    FASTQC.out.view()
}
```

File name in quotes

When an output file name contains a * or ? metacharacter it is interpreted as a pattern match.

This allows us to capture multiple files into a list and output them as a one item channel.

Processes: Output: Multiple files path

```
//process_output_multiple.nf
nextflow.enable.dsl=2

process FASTQC {
    input:
        path read

    output:
        path "fqc_res/*" ← Output qualifier
        script:
            """
            mkdir fqc_res
            fastqc $read -o fqc_res
            """
}

read_ch = channel.fromPath("data/yeast/reads/ref1*.fq.gz")

workflow {
    FASTQC(read_ch)
    FASTQC.out.view()
}
```

File name in quotes

When an output file name contains a * or ? metacharacter it is interpreted as a pattern match.

This allows us to capture multiple files into a list and output them as a one item channel.

Note: There are some caveats on glob pattern behaviour:

- Input files are not included in the list of possible matches.
- Glob pattern matches against both files and directories path.

Exercise: process_exercise_output.nf

Output channels

Modify the nextflow script `process_exercise_output.nf` to include an output block that captures the different index folders `index_$kmer`. Use the `view` operator on the output channel.

Code

```
//process_exercise_output.nf
nextflow.enable.dsl=2

process INDEX {
    input:
    path transcriptome
    each kmer

    //add output block here to capture index folders "index_$kmer"

    script:
    """
    salmon index -t $transcriptome -i index_$kmer -k $kmer
    """
}

transcriptome_ch = channel.fromPath('data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz')
kmer_ch = channel.of(21, 27, 31)

workflow {
    INDEX(transcriptome_ch, kmer_ch)
    INDEX.out.view()
}
```

`index_$kmer` name of output from command

Solution

Process: Inputs & Output: Grouped: tuple

Nextflow can handle groups of values using the `tuple` qualifiers

```
[group_key, [file1, file2, ...]]
```

```
['ref1', ['ref1_1.fq.gz', 'ref1_2.fq.gz']]
```

Process: Input: Grouped: tuple qualifier

```
//process_tuple_input.nf
nextflow.enable.dsl=2

process TUPLEINPUT{
    input:
        tuple val(sample_id), path(reads)

    script:
    """
    echo $sample_id
    echo $reads
    """

}

reads_ch = Channel.fromFilePairs('data/yeast/reads/ref1_{1,2}.fq.gz')

workflow {
    TUPLEINPUT(reads_ch)
}
```

When using channel containing a tuple, such a one created with `.filesFromPairs` factory method, the corresponding input declaration must be declared with a `tuple` qualifier, followed by definition of each item in the tuple.

Process: Input: Grouped: tuple qualifier

```
//process_tuple_input.nf
nextflow.enable.dsl=2

process TUPLEINPUT{
    input:
        tuple val(sample_id), path(reads)
        tuple val(path)
    script:
        echo $sample_id
        echo $reads
    }
}

reads_ch = Channel.fromFilePairs('data/yeast/reads/ref1_{1,2}.fq.gz')

workflow {
    TUPLEINPUT(reads_ch)
}
```

```
nextflow run process_tuple_input.nf -process.echo
```

Process: Output: Grouped: tuple qualifer

Capturing the output as a tuple

```
//process_tuple_io_fastp.nf
nextflow.enable.dsl=2

process FASTP {
    input:
        tuple val(sample_id), path(reads)

    output:
        tuple val(sample_id), path("*FP*.fq.gz")

    script:
        """
        fastp \
            -i ${reads[0]} \
            -I ${reads[1]} \
            -o ${sample_id}_FP_R1.fq.gz \
            -O ${sample_id}_FP_R2.fq.gz
        """
}

reads_ch = Channel.fromFilePairs('data/yeast/reads/ref1_{1,2}.fq.gz')

workflow {
    FASTP(reads_ch)
    FASTP.out.view()
}
```

tuple
val
path
[list]

In this script the output channel would contain a tuple with the grouping key value as the Nextflow variable `sample_id` and a list containing the files matching the following pattern `"*FP*.fq.gz"`.

As Nextflow input `reads` is a list we must reference individual files using [index]

```
nextflow run process_tuple_io.nf
```

Exercise: process_exercise_tuple.nf

Composite inputs and outputs

Fill in the blank __ input and output qualifiers for `process_exercise_tuple.nf`. Note: the output for the FASTQC process is a directory names `fastqc_out`.

Code

```
//process_exercise_tuple.nf
nextflow.enable.dsl=2

process FASTQC {
    input:
        tuple __(sample_id), __(reads)

    output:
        tuple __(sample_id), __("fastqc_out")

    script:
    """
    mkdir fastqc_out
    fastqc $reads -o fastqc_out -t 1
    """
}

reads_ch = Channel.fromFilePairs('data/yeast/reads/ref*_1,2.fq.gz')

workflow{
    FASTQC(reads_ch)
    FASTQC.out.view()
}
```

Output directory

Process: Conditional: when

Allows you to define a condition that must be verified in order to execute the process.

```
process < NAME > {
    [ directives ]
    input:
    < process inputs >
    output:
    < process outputs >
    when:
    < condition >
    [script|shell|exec]:
    < user script to be executed >
}
```

Process: Conditional: when

In the example below the process `CONDITIONAL` will only execute when the value of the `chr` variable is less than or equal to 5:

```
//process_when.nf
nextflow.enable.dsl=2

process CONDITIONAL {
    input:
        val chr

    when:
        chr <= 5

    script:
        """
        echo $chr
        """
}

chr_ch = channel.of(1..22)

workflow {
    CONDITIONAL(chr_ch)
}
```

```
$ nextflow run process_when.nf -process.echo
```

Process: Directives

```
process < NAME > {
    [ directives ]
    input:
    < process inputs >
    output:
    < process outputs >
    when:
    < condition >
    [script|shell|exec]:
    < user script to be
    executed >
}
```

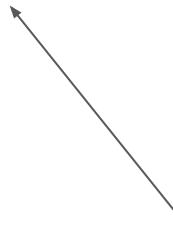


Process: Directives

Using the directive declarations block you can provide optional settings that will affect the execution of the current process.

They must be entered at the top of the process body, before any other declaration blocks (i.e. input, output, etc) and have the following syntax:

```
directive_name value [, value2 [,...]]
```



Directive arguments are separated by commas ,

Processes: Directives

Some common directives

cpus	Define the number of CPUs required for each task.
memory	Define the amount of memory required for each task.
time	Define the amount of time required for each task.
conda	Specify conda environment
echo	Print output to console
tag	Tag process
publishDir	Where to send results

Processes: Directives

```
//process_directive.nf
nextflow.enable.dsl=2

process PRINTCHR {
    tag "tagging with chr$chr"
    cpus 1
    echo true

    input:
    val chr

    script:
    ****
        echo processing chromosome: $chr
        echo number of cpus $task.cpus
    ****
}

chr_ch = channel.of(1..22, 'X', 'Y')

workflow {
    PRINTCHR(chr_ch)
}
```

Directives added above input:, output and script: block

task.cpus special nextflow variable to access number of cpus per task

Exercise: process_exercise_directives.nf

Adding directives

Modify the Nextflow script `process_exercise_directives.nf`

1. Add a `tag` directive logging the `sample_id` in the execution output.
2. Add a `cpus` directive to specify the number of cpus as 2.
3. Change the fastqc `-t` option value to `$task.cpus` in the script directive.

Code

```
//process_exercise_directives.nf
nextflow.enable.dsl=2

process FASTQC {
    //add tag directive
    //add cpu directive

    input:
    tuple val(sample_id), path(reads)

    output:
    tuple val(sample_id), path("fastqc_out")

    script:
    """
    mkdir fastqc_out
    fastqc $reads -o fastqc_out -t 1
    """
}

read_pairs_ch = Channel.fromFilePairs('data/yeast/reads/ref*_1,2.fq.gz')

workflow {
    FASTQC(read_pairs_ch)
    FASTQC.out.view()
}
```

Process: Organising Outputs: publishDir

- Nextflow manages intermediate results from the pipeline's expected outputs independently.
- Files created by a `process` are stored in a task specific working directory which is considered as temporary.
 - a. Normally this is under the `work` directory, which can be deleted upon completion.

Process: Organising Outputs: publishDir

- Nextflow manages intermediate results from the pipeline's expected outputs independently.
- Files created by a `process` are stored in a task specific working directory which is considered as temporary.
 - a. Normally this is under the `work` directory, which can be deleted upon completion.
- The files you want the workflow to return as results need to be defined
 - a. `process output` block
 - b. and then the output directory specified using the directive `publishDir`.

Process: Organising Outputs: publishDir

```
publishDir <directory>, parameter: value, parameter2: value ...
```

```
publishDir "results"
```

Note: A common mistake is to specify an output directory in the `publishDir` directive while forgetting to specify the files you want to include in the `output` block.

Processes: Organising Outputs: publishDir

```
//process_publishDir.nf
nextflow.enable.dsl=2

process QUANT {
    publishDir "results/quant"

    input:
    tuple val(sample_id), path(reads)
    each index

    output:
    tuple val(sample_id), path("${sample_id}_salmon_output")

    script:
    """
    salmon quant -i $index \
    -l A \
    -1 ${reads[0]} \
    -2 ${reads[1]} \
    -o ${sample_id}_salmon_output
    """
}

reads_ch = Channel.fromFilePairs('data/yeast/reads/ref1_{1,2}.fq.gz')
index_ch = Channel.fromPath('data/yeast/salmon_index')

workflow {
    QUANT(reads_ch, index_ch)
    QUANT.out.view()
}
```

For example if we want to capture the results of the QUANT process in a results/quant output directory we need to define the files in the output and specify the location of the results directory in the publishDir directive:

Process: Organising Outputs: publishDir

```
//process_publishDir.nf
nextflow.enable.dsl=2

process QUANT {
    publishDir "results/quant"

    input:
        tuple val(sample_id), path(reads)
        each index

    output:
        tuple val(sample_id), path("${sample_id}_salmon_output")
```

Bash

```
tree results
```

Output

```
results/
└ quant
  └ ref1_salmon_output -> work/48/f97234d7185cbfb86e2f11c1afab5/ref1_salmon_output
```

Symbolic link
(default)

Process: publishDir : Parameters

The `publishDir` directive can take optional parameters

mode	The file publishing method.
overwrite	When <code>true</code> any existing file in the specified folder will be overridden (default: <code>true</code> during normal pipeline execution and <code>false</code> when pipeline execution is resumed).
pattern	Specifies a glob file pattern that selects which files to publish from the overall set of output files.

Processes: publishDir : mode: copy

If we want to make a copy of the results from the working directory we must use mode: "copy"

Code

```
publishDir "results/quant", mode: "copy"
```

Symlink Default: Creates an absolute symbolic link in the published directory for each process output file

Process: Manage semantic sub-directories: pattern

You can use more than one publishDir to keep different outputs in separate directories

Process: Manage semantic sub-directories: pattern

You can use more than one publishDir to keep different outputs in separate directories

To specify which files to put in which output directory use the parameter **pattern** with a glob pattern that selects which files to publish from the overall set of output files.

```
publishDir "results/bams", pattern: "*.bam", mode: "copy"
```

```
output:  
    tuple val(sample_id),path("${sample_id}.bam")
```

Process: Manage semantic sub-directories: pattern

```
//process_publishDir_semantic.nf
nextflow.enable.dsl=2

process QUANT {
    publishDir "results/bams", pattern: "*.bam", mode: "copy"
    publishDir "results/quant", pattern: "${sample_id}_salmon_output", mode: "copy"

    input:
    tuple val(sample_id), path(reads)
    path index

    output:
    tuple val(sample_id), path("${sample_id}.bam")
    path "${sample_id}_salmon_output"

    script:
    """
    salmon quant -i $index \
    -l A \
    -1 ${reads[0]} \
    -2 ${reads[1]} \
    -o ${sample_id}_salmon_output \
    --writeMappings | samtools sort | samtools view -bS -o ${sample_id}.bam
    """
}

reads_ch = Channel.fromFilePairs('data/yeast/reads/ref1_{1,2}.fq.gz')
index_ch = Channel.fromPath('data/yeast/salmon_index')

workflow {
    QUANT(reads_ch, index_ch)
}
```

pattern: `"*.bam"`, mode: `"copy"`

Process: Manage semantic sub-directories: pattern

```
//process_publishDir_semantic.nf
nextflow.enable.dsl=2

process QUANT {
    publishDir "results/bams", pattern: "*.bam", mode: "copy"
    publishDir "results/quant", pattern: "${sample_id}_salmon_output", mode: "copy"

    input:
        tuple val(sample_id), path(reads)
        path index
}

Multiple publishDir lines
    output:
        tuple val(sample_id), path("${sample_id}.bam")
        path "${sample_id}_salmon_output"

    script:
        """
        salmon quant -i $index \
        -l A \
        -1 ${reads[0]} \
        -2 ${reads[1]} \
        -o ${sample_id}_salmon_output \
        --writeMappings | samtools sort | samtools view -bS -o ${sample_id}.bam
        """
}

reads_ch = Channel.fromFilePairs('data/yeast/reads/ref1_{1,2}.fq.gz')
index_ch = Channel.fromPath('data/yeast/salmon_index')

workflow {
    QUANT(reads_ch, index_ch)
}
```

pattern: "\${sample_id}_salmon_output"

Process: Manage semantic sub-directories: pattern

```
//process_publishDir_semantic.nf
nextflow.enable.dsl=2

process QUANT {
    publishDir "results/bams", pattern: "*.bam", mode: "copy"
    publishDir "results/quant", pattern: "${sample_id}_salmon_output", mode: "copy"

    input:
        tuple val(sample_id), path(reads)
        path index

    output:
        tuple val(sample_id), path("${sample_id}.bam")
        path "${sample_id}_salmon_output"

    script:
        """
        salmon quant -i $index \
        -l A \
        -1 ${reads[0]} \
        -2 ${reads[1]} \
        -o ${sample_id}_salmon_output \
        --writeMappings | samtools sort | samtools view -bS -o ${sample_id}.bam
        """
}

reads_ch = Channel.fromFilePairs('data/yeast/reads/ref1_{1,2}.fq.gz')
index_ch = Channel.fromPath('data/yeast/salmon_index')

workflow {
    QUANT(reads_ch, index_ch)
}
```

nextflow run process_publishDir_semantic.nf

Exercise: process_exercise_publishDir.nf

Publishing results

Add a `publishDir` directive to the nextflow script `process_exercise_publishDir.nf` that copies the index directory to the results folder .

Code

```
//process_exercise_pubilishDir.nf
nextflow.enable.dsl=2

process INDEX {
    //add publishDir directive here
    input:
        path transcriptome

    output:
        path "index"

    script:
    """
    salmon index -t $transcriptome -i index
    """
}

params.transcriptome = "data/yeast/transcriptome/Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz"
transcriptome_ch = channel.fromPath(params.transcriptome, checkIfExists: true)

workflow {
    INDEX(transcriptome_ch)
}
```

Process 2 : Key Points

- Outputs to a process are defined using the output blocks.

Process 2 : Key Points

- Outputs to a process are defined using the output blocks.
- You can group input and output data from a process using the tuple qualifer.

Process 2 : Key Points

- Outputs to a process are defined using the output blocks.
- You can group input and output data from a process using the tuple qualifer.
- The execution of a process can be controlled using the `when` declaration and conditional statements.

Process 2 : Key Points

- Outputs to a process are defined using the output blocks.
- You can group input and output data from a process using the tuple qualifer.
- The execution of a process can be controlled using the `when` declaration and conditional statements.
- Files produced within a process and defined as `output` can be saved to a directory using the `publishDir` directive.

Break

Operators

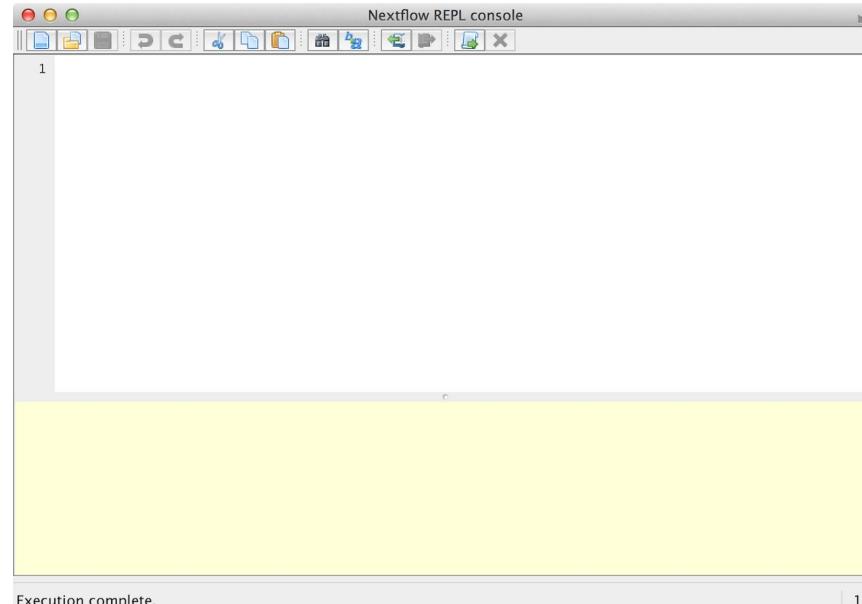
<https://carpentries-incubator.github.io/workflows-nextflow/07-operators>

Questions

- How do I perform operations, such as filtering, on channels?
- What are the different kinds of operations I can perform on channels?
- How do I combine operations?
- How can I use a CSV file to process data into a Channel?

Nextflow console

The Nextflow console is a REPL (read-eval-print loop) environment that allows one to quickly test part of a script or pieces of Nextflow code in an interactive manner.



Operators

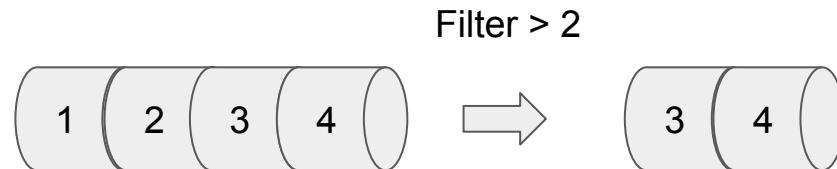
Nextflow operators are methods that allow you to connect channels to each other or to transform values emitted by a channel applying some user provided rules.

Operator Types

Operators can be classed into several groups

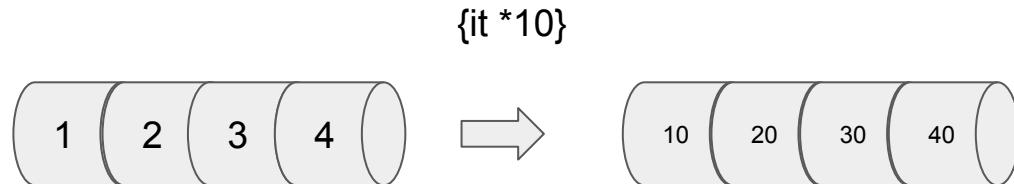
Operators: Filtering

- **Filtering** operators: reduce the number of elements in a channel.



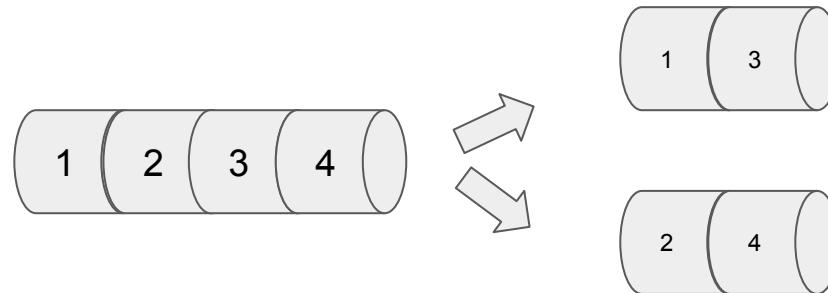
Operators: Transforming

- Filtering operators: reduce the number of elements in a channel.
- **Transforming** operators: transform the value/data in a channel.



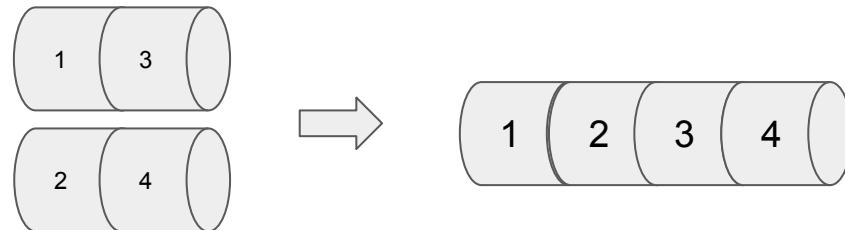
Operators: Split

- **Filtering** operators: reduce the number of elements in a channel.
- **Transforming** operators: transform the value/data in a channel.
- **Splitting** operators: split items in a channel into smaller chunks.



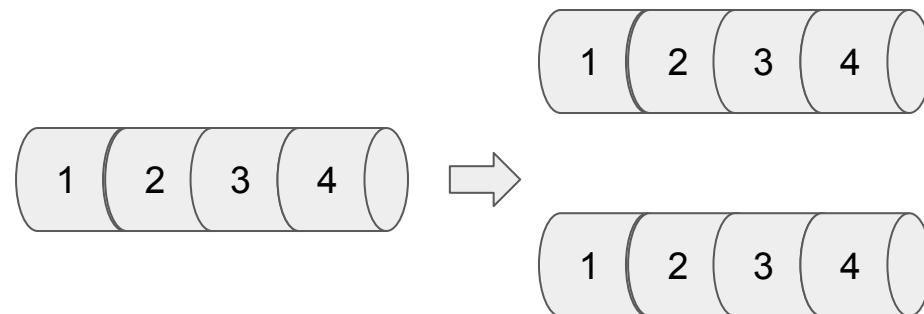
Operators: Combining/Merging Channels

- **Filtering** operators: reduce the number of elements in a channel.
- **Transforming** operators: transform the value/data in a channel.
- **Splitting** operators: split items in a channel into smaller chunks.
- **Combining** operators: join channel together.



Operators: Forking

- **Filtering** operators: reduce the number of elements in a channel.
- **Transforming** operators: transform the value/data in a channel.
- **Splitting** operators: split items in a channel into smaller chunks.
- **Combining** operators: join channel together.
- **Forking** operators: split a single channel into multiple channels.



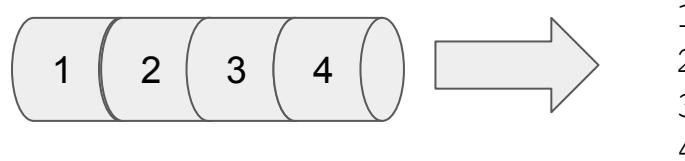
Operators: Maths

- **Filtering** operators: reduce the number of elements in a channel.
- **Transforming** operators: transform the value/data in a channel.
- **Splitting** operators: split items in a channel into smaller chunks.
- **Combining** operators: join channel together.
- **Forking** operators: split a single channel into multiple channels.
- **Maths** operators: apply simple math function on channels.



Operators

- **Filtering** operators: reduce the number of elements in a channel.
- **Transforming** operators: transform the value/data in a channel.
- **Splitting** operators: split items in a channel into smaller chunks.
- **Combining** operators: join channel together.
- **Forking** operators: split a single channel into multiple channels.
- **Maths** operators: apply simple math function on channels.
- **Other:** such as the `view` operator.



Using Operators

To use an operator, the syntax is the channel name, followed by a dot . , followed by the operator name and brackets () .

Dot notation .

```
channel_obj.<operator>()
```



Operators: View

The view operator prints the items emitted by a channel to the console standard output

```
ch = channel.of('1', '2', '3')
ch.view()
```

Operators: View: Chain

We can also chain together the channel factory method `.of` and the operator `.view()` using the dot notation.

```
ch = channel.of('1', '2', '3')
```

Operators: View: Chain

We can also chain together the channel factory method `.of` and the operator `.view()` using the dot notation.

```
ch = channel.of('1', '2', '3').view()
```

Operators: View: Readability

To make code more readable we can split the operators over several lines. The blank space between the operators is ignored and is solely for readability.

```
ch = channel.of('1', '2', '3')  
      .view()
```

Operators: View: Readability

To make code more readable we can split the operators over several lines.

The blank space between the operators is ignored and is solely for readability.

```
ch = channel  
      .of('1', '2', '3')  
      .view()
```

Operators: View: Closures

- We can change the output of using a closure

```
ch = channel  
    .of('1', '2', '3')  
    .view({ "chr$it" })
```

Operators: View: Closures

- We can change the output of using a closure
- A closure is a block of code that can be passed as an argument to a function

```
ch = channel  
    .of('1', '2', '3')  
    .view({ "chr$it" })
```

Operators: View: Closures

- We can change the output of using a closure
- A closure is a block of code that can be passed as an argument to a function
- By default the parameters for a closure are specified with the groovy keyword `$it` ('it' is for 'item')

```
ch = channel  
    .of('1', '2', '3')  
    .view({ "chr$it" })
```

{ } closure, like
anonymous function

Reference item using `it` variable , (in a
string use `$it`)

Operators: View: Closures

- We can change the output of using a closure
- A closure is a block of code that can be passed as an argument to a function
- By default the parameters for a closure are specified with the groovy keyword `$it` ('it' is for 'item')

```
ch = channel
      .of('1', '2', '3')
      .view({ "chr$it" })
ch.view()
```

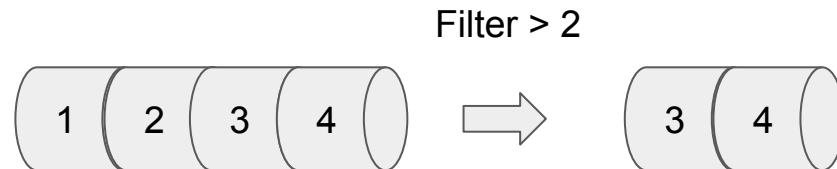
view doesn't alter contents

{} closure, like anonymous function

Reference item using `it` variable , (in a string use `$it`)

Operators: Filtering

- **Filtering** operators: reduce the number of elements in a channel.



Operators: Filtering: Data Type

The filtering condition can be specified by using either:

- a data type qualifier, e.g. Number (any integer,float ...), String, Boolean

```
chr_ch = channel.of( 1..22, 'X', 'Y' )
chr_ch.view()
```

```
autosomes_ch =chr_ch.filter( Number )
autosomes_ch.view()
```

Operators: Filtering: Data Type

The filtering condition can be specified by using either:

- a data type qualifier, e.g. Number (any integer,float ...), String, Boolean

```
chr_ch = channel.of( 1..22, 'X', 'Y' )
chr_ch.view()
```

```
autosomes_ch =chr_ch.filter( String )
autosomes_ch.view()
```

Operators: Filtering: Regex ~

To filter by a regular expression you have to do is to put `~` right in front of the string literal regular expression

```
~"(^ [Nn]extflow)"
```

or use slashy strings which replace the quotes with

```
/.~/^ [Nn]extflow./
```

The following example shows how to filter a channel by using a regular expression `~/^1.*/` inside a slashy string, that returns only strings that begin with 1:

```
chr_ch = channel
    .of( 1..22, 'X', 'Y' )
    .filter(~/^1.*/)
    .view()
```

Operators: Filtering: Boolean

A filtering condition can be defined by using a Boolean expression described by a closure { } and returning a boolean value.

```
channel
    .of( 1..22, 'X', 'Y' )
    .filter(Number)
    .filter { <expression> }
    .view()
```

Operators: Filtering: Boolean

A filtering condition can be defined by using a Boolean expression described by a closure `{ }` and returning a boolean value.

The following code shows how to combine a filter for a type qualifier `Number` with another filter operator using a Boolean expression to emit numbers less than 5:

```
channel
    .of( 1..22, 'X', 'Y' )
    .filter(Number)
    .filter { it < 5 }
    .view()
```

Operators: Filtering: Literal values

Finally, if we only want to include elements of a specific value we can specify a literal value.

In the example below we use the literal value '`'x'`' to filter the channel for only those elements containing the value `x`.

```
channel
    .of( 1..22, 'x', 'Y' )
    .filter('x')
    .view()
```

Exercise

Filter a channel

Add two channel filters to the Nextflow script below to view only the even numbered chromosomes.

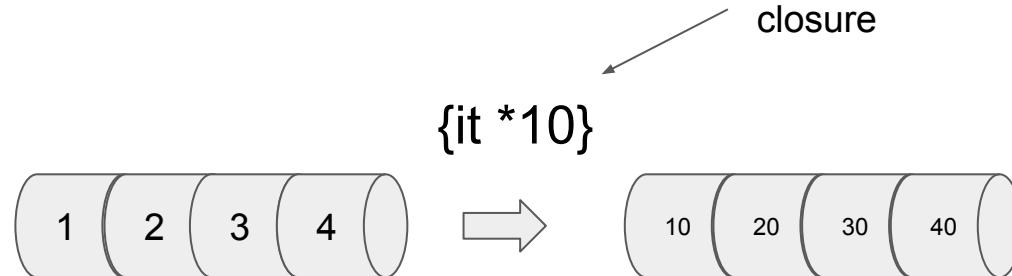
Note: The expression `it % 2` produces the remainder of a division.

Code

```
chr_ch = channel
.of( 1..22, 'X', 'Y' )
.view()
```

Operators: Transforming

.**Transforming** operators: transform the value/data in a channel.



Operators: Transforming : Applying a function to items in a channel

The **map** operator applies a function of your choosing to every item in a channel, and returns the items so obtained as a new channel

```
channel
    .of( 'chr1' , 'chr2' )
    .map ({ it.replaceAll("chr","") })
    .view()
```

Here the map operator uses the groovy string function **replaceAll** to remove the chr prefix from each element.

Operators: Transforming : Creating tuples

We can also use the `map` operator to transform each element into a tuple.

```
fq_ch = channel
    .fromPath( 'data/yeast/reads/*.{fq.gz}' )
    .view()
```

Operators: Transforming : Creating tuples

We can also use the `map` operator to transform each element into a tuple.

```
fq_ch = channel
    .fromPath( 'data/yeast/reads/*.fq.gz' )
    .map ({ it -> [it, it.countFastq()] })
    .view()
```

Nextflow helper function to count number of reads in a fastq file

Operators: Transforming : Creating tuples

We can change the default name of the closure parameter keyword from `it` to a more meaningful name `file` using `->`

```
fq_ch = channel
    .fromPath( 'data/yeast/reads/*.fq.gz' )
    .map ({ file -> [file, file.countFastq()] })
    .view()
```

Nextflow helper function to count number of reads in a fastq file

Operators: Transforming : Creating tuples

When we have multiple parameters we can specify the keywords at the start of the closure, e.g. `file, numreads ->`.

```
fq_ch = channel
    .fromPath( 'data/yeast/reads/*.fq.gz' )
    .map ({ file -> [file, file.countFastq()] })
    .view ({ file, numreads -> "file $file contains
$numreads reads" })
```

Operators: Transforming : Creating tuples

We can then add a `filter` operator to only retain those fastq files with more than 25000 reads.

```
fq_ch = channel
    .fromPath( 'data/yeast/reads/*.fq.gz' )
    .map ({ file -> [file, file.countFastq()] })
    .filter({ file, numreads -> numreads > 25000})
    .view ({ file, numreads -> "file $file contains
$numreads reads" })
```

Exercise:

map operator

Add a `map` operator to the Nextflow script below to transform the contents into a tuple with the file and the file's name, using the `.getName` method. The `getName` method gives the filename. Finally `view` the channel contents.

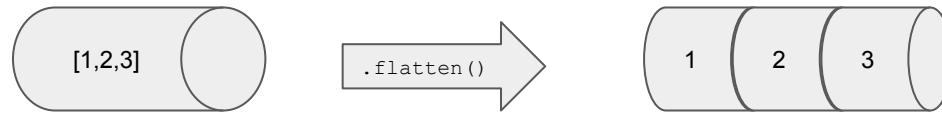
Code

```
channel
.fromPath( 'data/yeast/reads/*.fq.gz' )
.view()
```

```
channel
.fromPath( 'data/yeast/reads/*.fq.gz' )
.map ({ file -> [file, file.countFastq()] })
.filter({ file, numreads -> numreads > 25000})
.view ({ file, numreads -> "file $file contains $numreads reads"
})
```

Operators: Transforming: flatten

The `flatten` operator transforms a channel in such a way that every item in a `list` is flattened so that each single entry is emitted as a sole element by the resulting channel.



```
list1 = [1,2,3]
ch = channel
  .of(list1)
  .view()
```

```
ch =channel
  .of(list1)
  .flatten()
  .view()
```

Operators: Transforming: collect

The `collect` operator collects all the items emitted by a channel to a list and return the resulting object as a sole emission



`ch = channel`

`.of(1, 2, 3)`

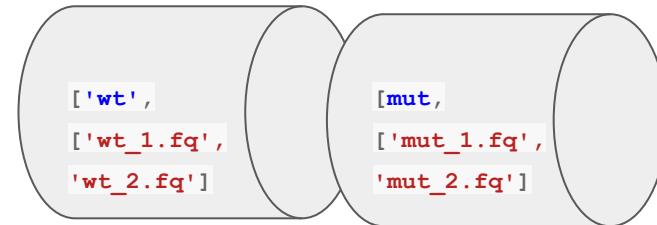
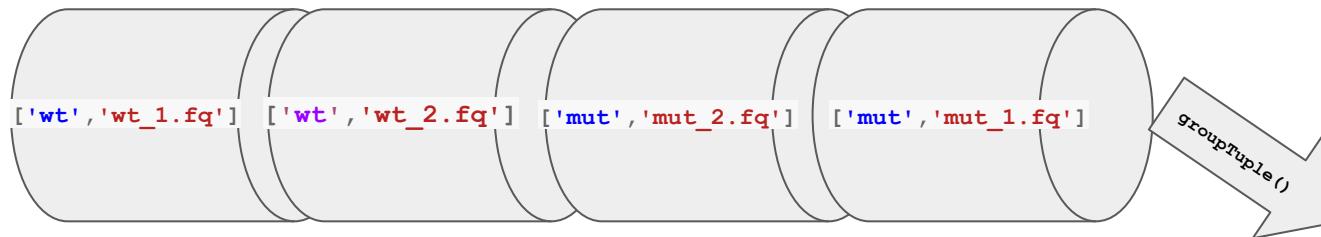
`.collect()`

`.view()`

Collect can be very useful when you have split your data to run in parallel and want to bring the results back together

Operators: Transforming: Grouping contents of a channel by a key

The `groupTuple` operator collects `tuples` or `lists` of values by grouping together the channel elements that share the same key. Finally it emits a new tuple object for each distinct key collected.



Operators: Transforming: Grouping contents of a channel by a key

The `groupTuple` operator collects `tuples` or `lists` of values by grouping together the channel elements that share the same key. Finally it emits a new tuple object for each distinct key collected.

```
ch = channel
    .of( ['wt','wt_1.fq'],
        ['wt','wt_2.fq'],
        ["mut",'mut_1.fq'],
        ['mut', 'mut_2.fq'] )
    .groupTuple()
    .view()
```

Exercise: Group Tuple

```
channel.fromPath('data/yeast/reads/*fq.gz')  
    .view()
```

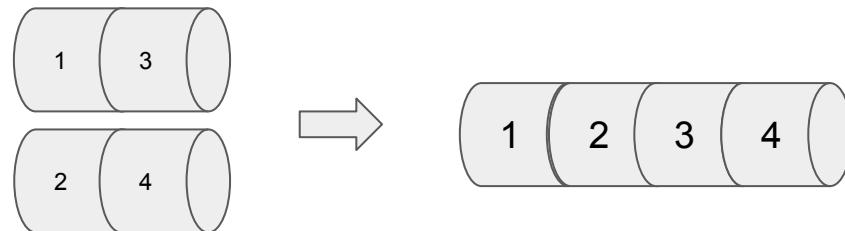
Modify the Nextflow script above to add the `map` operator to create a tuple with the name prefix as the key and the file as the value using the closure below.

```
{ file -> [ file.getName().split('_')[0], file ] }
```

Finally group together all files having the same common prefix using the `groupTuple` operator and `view` the contents of the channel.

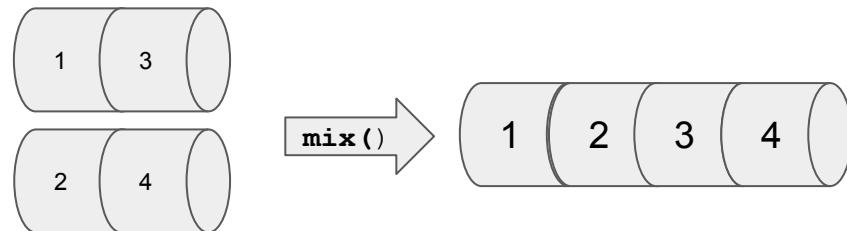
Operators: Combining/Merging Channels

Combining operators: join channel together.



Operators: Combining/Merging Channels: mix

The `mix` operator combines the items emitted by two (or more) channels into a single channel.



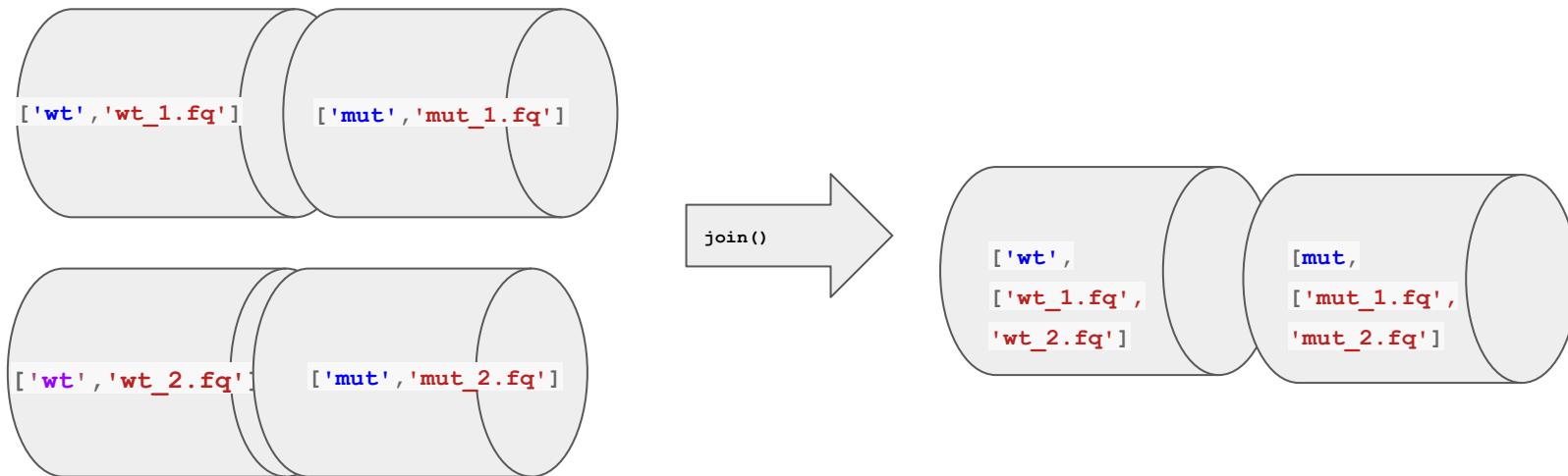
Operators: Merging channels: mix

```
ch1 = channel.of( 1,2,3 )
ch2 = channel.of( 'X','Y' )
ch3 = channel.of( 'mt' )

ch4 = ch1.mix(ch2,ch3).view()
```

Operators: Merging: `Join()`

The `join` operator creates a channel that joins together the items emitted by two channels for which exists a matching **key**. The key is defined, by default, as the first element in each item emitted.



Operators: Merging: Join()

The `join` operator creates a channel that joins together the items emitted by two channels for which exists a matching key. The key is defined, by default, as the first element in each item emitted.

```
reads1_ch = channel
    .of(['wt', 'wt_1.fq'], ['mut', 'mut_1.fq'])
```

```
reads2_ch= channel
    .of(['wt', 'wt_2.fq'], ['mut', 'mut_2.fq'])
```

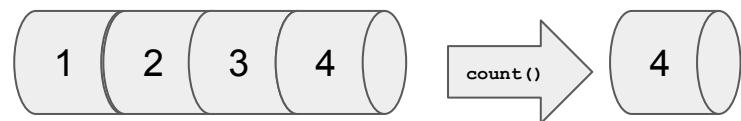
```
reads_ch = reads1_ch
    .join(reads2_ch)
    .view()
```

Operators: Maths

Maths operators: apply simple math function on channels.

The maths operators are:

- count ()
- min ()
- max ()
- sum ()
- toInteger ()

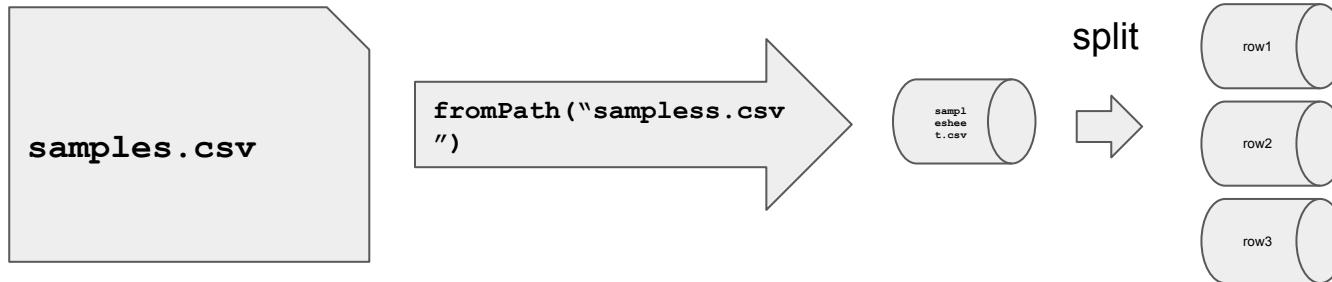


Operators: Maths : Counting items in a channel count ()

```
channel
    .of(1..22, 'X', 'Y')
    .count()
    .view()
```

Operators: Splitting items in a channel

Sometimes you want to split the content of a individual item in a channel, like a file or string, into smaller chunks that can be processed by downstream operators or processes e.g. items stored in a CSV file.



Operators: Splitting items in a channel

Nextflow has a number of splitting operators that can achieve this:

- `splitCsv()`
- `splitFasta()`
- `splitFastq()`
- `splitText()`

Operators: Splitting items in a channel: `splitCsv()`

The file `data/yeast/samples.csv` contains a row for every sample and columns with sample name and fastq read pair location

```
sample_id,fastq_1,fastq_2  
ref1,data/yeast/reads/ref1_1.fq.gz,data/yeast/reads/ref1_2.fq.gz  
ref2,data/yeast/reads/ref2_1.fq.gz,data/yeast/reads/ref2_2.fq.gz
```

`data/yeast/samples.csv`

`channel`

```
.fromPath('data/yeast/samples.csv')  
.splitCsv().view()
```

Operators: Accessing values: `splitCsv()`

Values can be accessed by their positional indexes using the square brackets syntax `[index]`. So to access the first column you would use `[0]` as shown in the following example

```
channel
```

```
    .fromPath('data/yeast/samples.csv')  
        .splitCsv().view({it[0]})
```

```
[sample_id, fastq_1, fastq_2]
```

```
[ref1, data/yeast/reads/ref1_1.fq.gz, data/yeast/reads/ref1_2.fq.gz]
```

```
[ref2, data/yeast/reads/ref2_1.fq.gz, data/yeast/reads/ref2_2.fq.gz]
```

Operators: Splitting items in a channel: `splitCsv()`

When the CSV begins with a header line, defining the column names, you can specify the parameter `header: true` which allows you to reference each value by its name, as shown in the following example:

```
sample_id,fastq_1,fastq_2  
ref1,data/yeast/reads/ref1_1.fq.gz,data/yeast/reads/ref1_2  
.fq.gz  
ref2,data/yeast/reads/ref2_1.fq.gz,data/yeast/reads/ref2_2  
.fq.gz
```

Channel

```
.fromPath('data/yeast/samples.csv')  
.splitCsv(header:true)  
.view({it.fastq_1})
```

Exercise: Parse a CSV file

Modify the Nextflow script to print the first column `sample_id`.

```
csv_ch=channel  
    .fromPath( 'data/yeast/samples.csv' )
```

Operators: Splitting : Tab delimited files

If you want to split a **tab** delimited file or file separated by another character use the **sep** parameter of the split **splitCsv** operator.

Channel

```
.of("val1\tval2\tval3\nval4\tval5\tval6\n")  
.splitCsv(sep: "\t")  
.view()
```

\t tab character

Operators: Key Points

- Nextflow operators are methods that allow you to modify, set or view channels.

Operators: Key Points

- Nextflow operators are methods that allow you to modify, set or view channels.
- Operators can be separated into several groups;
 - filtering
 - transforming
 - splitting,
 - combining
 - forking and
 - Maths operators

Operators: Key Points

- Nextflow operators are methods that allow you to modify, set or view channels.
- Operators can be separated into several groups;
 - filtering
 - transforming
 - splitting,
 - combining
 - forking and
 - Maths operators
- To use an operator use the dot notation after the Channel object e.g.

```
my_ch.view()
```

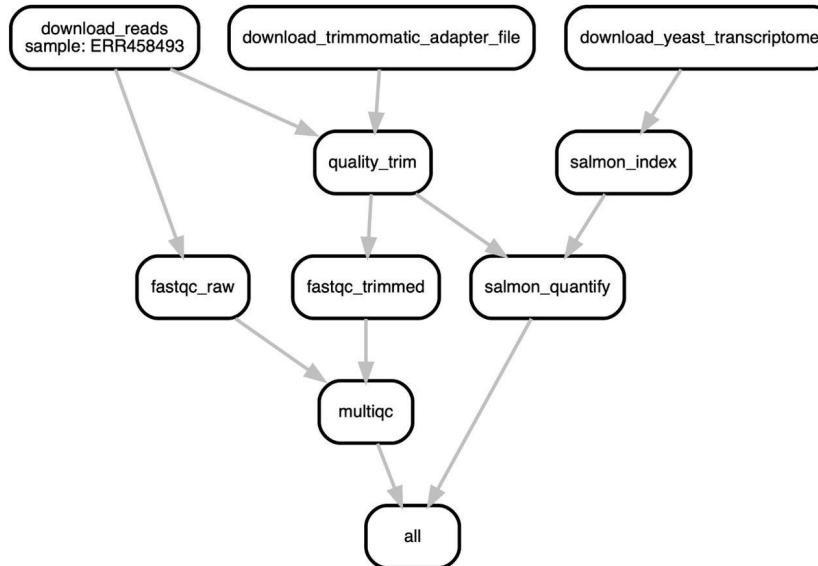
Operators: Key Points

- Nextflow operators are methods that allow you to modify, set or view channels.
- Operators can be separated into several groups;
 - filtering
 - transforming
 - splitting,
 - combining
 - forking and
 - Maths operators
- To use an operator use the dot notation after the Channel object e.g.
`my_ch.view()`.
- You can parse text items emitted by a channel, that are formatted using the CSV format, using the `splitCsv` operator.

Day2

Day 1 recap: Workflows

- A workflow is a sequence of interconnected tasks that processes a set of data.



Day 1: Nextflow a WFMS

- Nextflow is a workflow management system
 - Orchestrates Scientific pipelines, e.g. rnaseq
 - Enables
 - Scalability
 - Reproducibility
- Uses its own Domain Specific Language
 - Now in its second generation DSL2

Day 1 recap: Processes

- Tasks, e.g command you would run on Linux command line

```
$ fastq -o out ${reads}
```

- Define input, outputs and resources (mem,cpu,..etc)

```
process FASTQC {
    cpus 2
    memory 1.G

    input:
        tuple val(sample_id), path(reads)

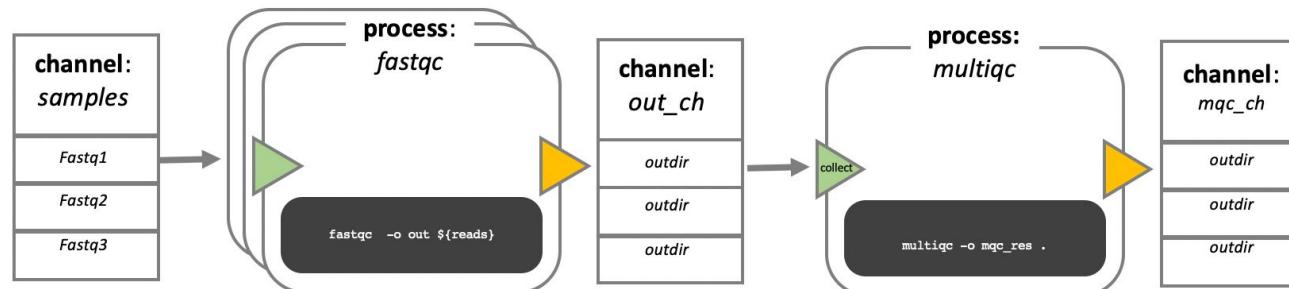
    output:
        path("out")
    script:
        """
            fastq -o ${reads}
        """
}
```

Day 1 recap: Channels

Channels, this is how you connect process input and outputs

- 2 types of channels
 - queue (1 or more items, consumed)
 - value (single item, reusable)
- The number of items within a channel determines the number of times a process runs.
- Create with channel with channel factories e.g.

```
channel.fromFilePairs("*.{1,2}.fq.gz")
```



Day 1 recap: Operators and Parameters

- Operators, How you can modify the content of Channels
 - `view()`,`mix()`,`collect()`,`groupBy()`, `count()`,`filter()`
- Parameterizing workflows for reusability, params
 - From the command line --<params>
 - `$ nextflow run wc.nf --input 'data/yeast/reads/ref2_2.fq.gz'`
 - Combined multiple in parameter in a file -params-file
 - `$ nextflow run wc-params.nf -params-file wc-params.json`

Workshop Dataset (data/yeast)

- RNA-Sequencing experiment
- Yeast, *Saccharomyces cerevisiae*
- Experiment: Measured changes in transcriptome following stress response
- FASTQ PE reads for <sample_name>_<rep>_<read>.fq.gz
 - ethoh60: High ethanol concentration (ethanol 60 g/L)
 - temp33: Temperature (33 degree celsius)
 - ref: Reference (control budding whole organism)
- Transcriptome
 - Saccharomyces_cerevisiae.R64-1-1.cdna.all.fa.gz

Day 2: Morning

Episode	Time	Course Material
Workflow	9:30-10:15	
Reporting	10:15-11:00	
Break	11:00-11:15	
Configuration	11:15-12:15	
Break Lunch	12:15-13:15	

Day 2: Afternoon

Episode	Time	Course Material
Checkpointing and caching	13:15-14:00	
RNA-Seq pipeline	14:00-15:00	
Break	15:00-15:15	
nf-core	15:15-16:05	

Workflows

Connecting Processes

Workflows

- How do I connect channels and processes to create a workflow?
- How do I invoke a process inside a workflow?

Workflow definition

We can connect processes to create our pipeline inside a workflow scope.

```
workflow {
    process1_ch=PROCESS_1()
    PROCESS_2(process1_ch)
}
```

Workflow definition

We can connect processes to create our pipeline inside a workflow scope.

```
workflow {  
    workflow {  
        process1_ch=PROCESS_1()  
        PROCESS_2(process1_ch)  
    }  
}
```

Keyword `workflow`

A workflow definition which does not declare any name is assumed to be the main workflow, and it implicitly executed.

Therefore it's the entry point of the workflow application.

Workflow definition

We can connect processes to create our pipeline inside a workflow scope.

```
workflow {  
    process1_ch=PROCESS_1()  
    PROCESS_2(process1_ch)  
}
```

Matching Curly brackets {}
Body of workflow

Workflow: Invoking a process

As seen previously, a `process` is invoked as a function in the `workflow` scope, passing the expected input channels as arguments as if were.

```
<process_name>(<input_ch1>,<input_ch2>,...)
```

```
workflow {
    process1_ch=PROCESS_1()
    PROCESS_2(process1_ch)
}
```

Calling Processes

To combined multiple processes invoke them in the order they would appear in a workflow

```

//workflow_01.nf
nextflow.enable.dsl=2

process INDEX {
    input:
        path transcriptome
    output:
        path 'index'
    script:
        """
            salmon index -t $transcriptome -i index
        """
}

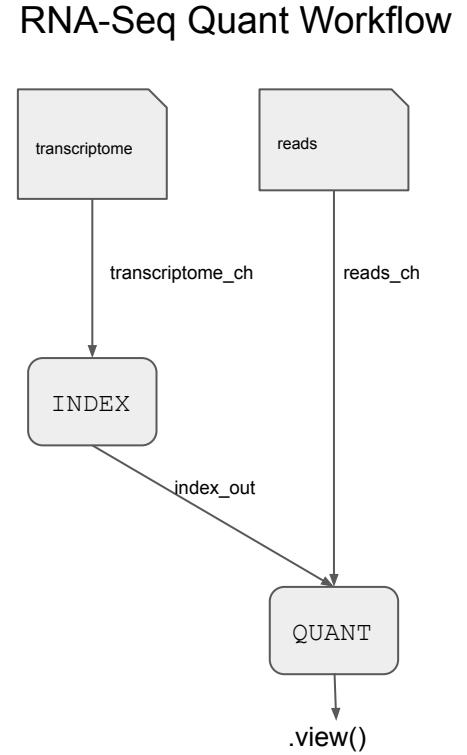
process QUANT {
    input:
        each path(index)
        tuple(val(pair_id), path(reads))
    output:
        path pair_id
    script:
        """
            salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id
        """
}

workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz', checkIfExists: true)
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz', checkIfExists: true)

    //index process takes 1 input channel as a argument
    index_out = INDEX(transcriptome_ch)

    //quant channel takes 2 input channels as arguments
    QUANT(index_out_ch, read_pairs_ch).view()
}

```



```

//workflow_01.nf
nextflow.enable.dsl=2

process INDEX {
    input:
        path transcriptome
    output:
        path 'index'
    script:
        """
            salmon index -t $transcriptome -i index
        """
}

process QUANT {
    input:
        each path(index)
        tuple(val(pair_id), path(reads))
    output:
        path pair_id
    script:
        """
            salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id
        """
}

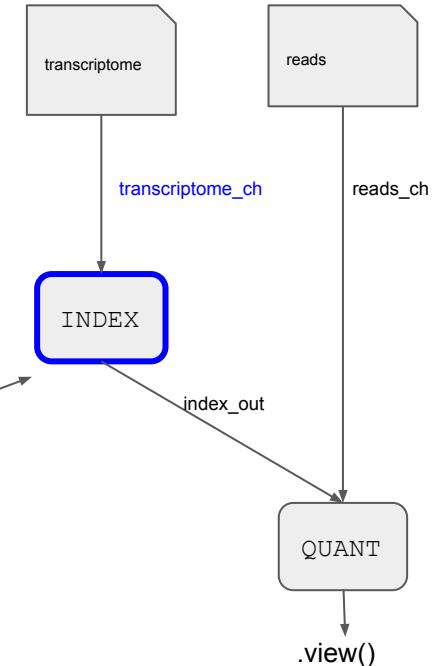
workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz', checkIfExists: true)
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz', checkIfExists: true)

    //index process takes 1 input channel as a argument
    index_out = INDEX(transcriptome_ch)

    //quant channel takes 2 input channels as arguments
    QUANT(index_out_ch, read_pairs_ch).view()
}

```

RNA-Seq Quant Workflow

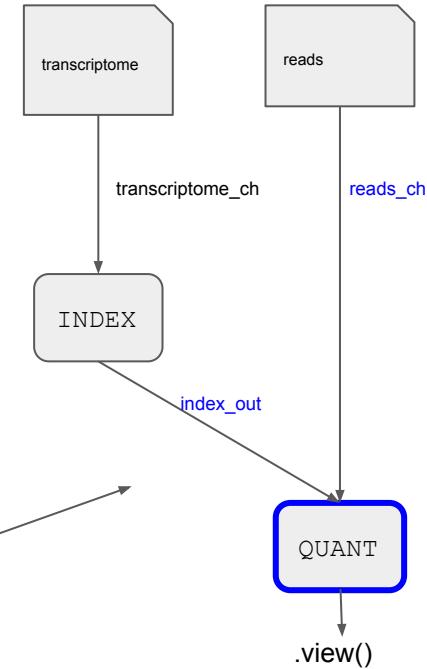


the INDEX process is invoked first

```
//workflow_01.nf  
nextflow.enable.dsl=2
```

```
process INDEX {  
    input:  
        path transcriptome  
    output:  
        path 'index'  
    script:  
        """  
            salmon index -t $transcriptome -i index  
        """  
}  
  
process QUANT {  
    input:  
        each path(index)  
        tuple(val(pair_id), path(reads))  
    output:  
        path pair_id  
    script:  
        """  
            salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id  
        """  
}  
  
workflow {  
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz', checkIfExists: true)  
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz', checkIfExists: true)  
  
    //index process takes 1 input channel as a argument  
    index_out = INDEX(transcriptome_ch)  
  
    //quant channel takes 2 input channels as arguments  
    QUANT(index_out_ch, read_pairs_ch).view()  
}
```

RNA-Seq Quant Workflow



The `INDEX` output channel, assigned to the variable `index_out`, is passed as the first argument to the `QUANT` process. The `read_pairs_ch` channel is passed as the second argument.

Workflow: Process Composition

Processes having matching `input-output` declaration can be composed so that the output of the first process is passed as input to the following process.

```
workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz', checkIfExists: true)
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz', checkIfExists: true)

    //index process takes 1 input channel as a argument
    index_out = INDEX(transcriptome_ch)

    //quant channel takes 2 input channels as arguments
    QUANT(index_out_ch, read_pairs_ch).view()
}
```

Workflow: Process Composition: Nested Call

We can also pass the Process call as an input to another Process.

```
workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz', checkIfExists: true)
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz', checkIfExists: true)

    //index process takes 1 input channel as a argument
    index_out = INDEX(transcriptome_ch)

    //quant channel takes 2 input channels as arguments
    QUANT(index_out_ch, read_pairs_ch).view()
}
```

```
workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz')
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz')

    // pass INDEX process as a parameter to the QUANT process
    QUANT(INDEX(transcriptome_ch), read_pairs_ch ).view()
}
```

Workflow: Process Outputs

In the previous examples we have connected the INDEX process output to the QUANT process by;

1. Assigning it to a variable index_out = INDEX(transcriptome_ch) and passing it to the QUANT process as an argument.

```
workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz',checkIfExists: true)
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz',checkIfExists: true)

    //index process takes 1 input channel as a argument
    index_out = INDEX(transcriptome_ch)

    //quant channel takes 2 input channels as arguments
    QUANT(index_out_ch,read_pairs_ch).view()
}
```

Workflow: Process Outputs

In the previous examples we have connected the INDEX process output to the QUANT process by;

1. Assigning it to a variable `index_out = INDEX(transcriptome_ch)` and passing it to the QUANT process as an argument.
2. Calling the process as an argument within the QUANT process, `QUANT(INDEX(transcriptome_ch), read_pairs_ch)`

```
workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz')
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz')

    // pass INDEX process as a parameter to the QUANT process
    QUANT(INDEX(transcriptome_ch), read_pairs_ch).view()
}
```

Workflow: Process Outputs

A process's output channel can also be accessed calling the process and then using the `out` attribute for the respective `process` object.

```
[..truncated..]

workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz')
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz')

    //call INDEX process
    INDEX(transcriptome_ch)                                Call process

    // INDEX process output accessed using the `out` attribute
    QUANT(INDEX.out,read_pairs_ch)                         <PROCESS_OBJ>.out
    QUANT.out.view()
}

}
```

Workflow: Process Outputs

When a process defines two or more output channels, each of them can be accessed using

1. The list element operator e.g. `out[0]`, `out[1]`

Workflow: Process Outputs

When a process defines two or more output channels, each of them can be accessed using

1. The list element operator e.g. `out[0]`, `out[1]`
2. Named output

```
output:
```

```
path 'index', emit: salmon_index
```

Workflow: Process Outputs

When a process defines two or more output channels, each of them can be accessed using

1. The list element operator e.g. `out[0]`, `out[1]`
2. Named output

```
output:
```

```
path 'index', emit: salmon_index
```

```
process INDEX { .. }
```

```
INDEX.out.salmon_index
```

```
workflow { .. }
```

Code

```
//workflow_02.nf
nextflow.enable.dsl=2

process INDEX {
    input:
    path transcriptome

    output:
    path 'index', emit: salmon_index
    script:
    """
    salmon index -t $transcriptome -i index
    """
}

process QUANT {
    input:
    each path(index)
    tuple(val(pair_id), path(reads))
    output:
    path pair_id
    script:
    """
    salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id
    """
}

workflow {
    transcriptome_ch = channel.fromPath('data/yeast/transcriptome/*.fa.gz')
    read_pairs_ch = channel.fromFilePairs('data/yeast/reads/*_{1,2}.fq.gz')

    //call INDEX process
    INDEX(transcriptome_ch)

    //access INDEX object named output
    QUANT(INDEX.out.salmon_index, read_pairs_ch).view()
}
```

Workflow: Process Named Outputs

emit: <name>

Access output using name

Workflow: Accessing script parameters

- A workflow component can access any variable and parameter defined in the outer scope:
- The scope is the part of the Nextflow script where a named variable is accessible.

Workflow
scope



```
//workflow_03.nf
[..truncated..]

params.transcriptome = 'data/yeast/transcriptome/*.fa.gz'
params.reads = 'data/yeast/reads/ref1*_1,2.fq.gz'

workflow {
    transcriptome_ch = channel.fromPath(params.transcriptome)
    read_pairs_ch = channel.fromFilePairs(params.reads)

    INDEX(transcriptome_ch)
    QUANT(INDEX.out.salmon_index, read_pairs_ch).view()
}
```

Exercise: workflow_exercise.nf

```
//workflow_exercise.nf
nextflow.enable.dsl=2
params.reads = 'data/yeast/reads/*_{1,2}.fq.gz'

process FASTQC {
    input:
        tuple val(sample_id), path(reads)

    output:
        path "fastqc_${sample_id}_logs/*.zip"

    script:
        //flagstat simple stats on bam file
        """
        mkdir fastqc_${sample_id}_logs
        fastqc -o fastqc_${sample_id}_logs -f fastq -q ${reads} -t ${task.cpus}
        """
}

process PARSEZIP {
    publishDir "results/fqpass", mode:"copy"
    input:
        path flagstats

    output:
        path 'pass_basic.txt'

    script:
        """
        for zip in *.zip; do zipgrep 'Basic Statistics' \$zip|grep 'summary.txt'; done > pass_basic.txt
        """
}

read_pairs_ch = channel.fromFilePairs(params.reads,checkIfExists: true)
workflow {
    //connect process FASTQC and PARSEZIP
    // remember to use the collect operator on the FASTQC output
}
```

Create a workflow by connecting the output of the process **FASTQC** to **PARSEZIP** in the Nextflow script **workflow_exercise.nf**.

The process **FASTQC** generates basic quality control metrics for raw sequencing data using the **FASTQC** program.

The process **PARSEZIP** extracts the Basic Statistics from FASTQC compressed output file and writes it to a file.

Note: You will need to pass the **read_pairs_ch** as an argument to **FASTQC** and you will need to use the **collect** operator to gather the items in the **FASTQC** channel output to a single List item.

Look at the contents of the file **pass_basic.txt** in **results/fqpass** folder. How many lines does the file have?

Workflows: Key Points

- A Nextflow workflow is defined by invoking processes inside the workflow scope.
- A process is invoked like a function inside the workflow scope passing any required input parameters as arguments. e.g. `INDEX(transcriptome_ch)`.
- Process outputs can be accessed using the out attribute for the respective process. Multiple outputs from a single process can be accessed using the `[]` e.g `[0]` or `[1]` or output name e.g. `INDEX.out.salmon_index`.

Reporting

Questions

- How do I get information about my pipeline run?
- How can I see what commands I ran?
- How can I create a report from my run?

Reporting: Nextflow log

Once a script has run, Nextflow stores a log of all the workflows executed in the current folder.

Similar to an electronic lab book, this means you have a record of all processing steps and commands run.

You can print Nextflow's execution history and log information using the `nextflow log` command.

```
$ nextflow log
```

Reporting: Nextflow log

This will print a summary of the executions log and runtime information for all pipelines run.

By default, included in the summary, are the

1. Timestamp: Date and time it ran
2. Duration: How long it ran for
3. Run name: Unique Name, adjective_scientist-surname
4. Status: Did it FAIL or Run OK
5. Revision ID: Unique ID hash for Nextflow run
6. Session ID
7. Command: The command run on the command line.

TIMESTAMP	DURATION	RUN NAME	STATUS	REVISION ID	SESSION ID	COMMAND
-----------	----------	----------	--------	-------------	------------	---------

Exercise

Show Execution Log

Listing the execution logs of previous invocations of all pipelines in a directory.

Bash

```
$ nextflow log
```

Reporting: Pipeline execution report

If we want to get more information about an individual run we can add the run name or session ID to the log command.

```
$ nextflow log tiny_fermat
```

This will list the work directory for each process.

Reporting: Fields

If we want to print more metadata we can use the `log` command and the option `-f` (fields) followed by a comma delimited list of fields.

This can be composed to track the provenance of a workflow result.

```
$ nextflow log tiny_fermat -f 'process,exit,hash,duration'
```

Reporting: Fields

The complete list of available fields can be retrieved with the command:

```
$ nextflow log tiny_fermat -f 'script'
```

Reporting: script

If we want a log of all the commands executed in the pipeline we can use the `script` field.

It is important to note that the resultant output can not be used to run the pipeline steps.

```
$ nextflow log -l
```

Reporting: Filtering

The output from the log command can be very long.

We can subset the output using the option `-F` (filter) specifying the filtering criteria.

This will print only those tasks matching a pattern using the syntax `=~/<pattern>/`

filter for process with the name `fastqc` we would run:

```
$ nextflow log tiny_fermat -F 'process =~ /fastqc/'
```

Exercise

View run log

Use the Nextflow `log` command specifying a `run name` and the fields. `name`, `hash`, `process` and `status`

Filter pipeline run log

Use the `-F` option and a regular expression to filter the for a specific process e.g. `multiqc`.

Hint `=~/<pattern>/`

Reporting: Templates

The `-t` option allows a template (string or file) to be specified. This makes it possible to create a custom report in any text based format.

For example you could save this markdown snippet to a file e.g. `my-template.md`:

```
## $name

script:

$script

exist status: $exit
task status: $status
task folder: $folder
```

Reporting: Templates

The following `log` command will output a markdown file containing the `script`, `exit status` and `folder` of all executed tasks:

```
## $name  
  
script:  
  
    $script  
  
    exist status: $exit  
    task status: $status  
    task folder: $folder
```

```
$ nextflow log elegant_descartes -t my-template.md > execution-report.md
```

Reporting: Templates HTML

The template file can also be written in HTML. (`template.html`)

```
<div>
    <h2>${name}</h2>
    <div>
        Script:
        <pre>${script}</pre>
    </div>

    <ul>
        <li>Exit: ${exit}</li>
        <li>Status: ${status}</li>
        <li>Work dir: ${workdir}</li>
        <li>Container: ${container}</li>
    </ul>
</div>
```

```
$ nextflow log elegant_descartes -t template.html > provenance.html
```

Exercise

Generate an HTML run report

Generate an HTML report for a run using the `-t` option and the `template.html` file.

Reporting: Key Points

- Nextflow can produce a custom execution report with run information using the `log` command.
- You can generate a report using the `-t` option specifying a template file.

Break

Configuration

Configuring your workflow via file

<https://carpentries-incubator.github.io/workflows-nextflow/08-configuration/>

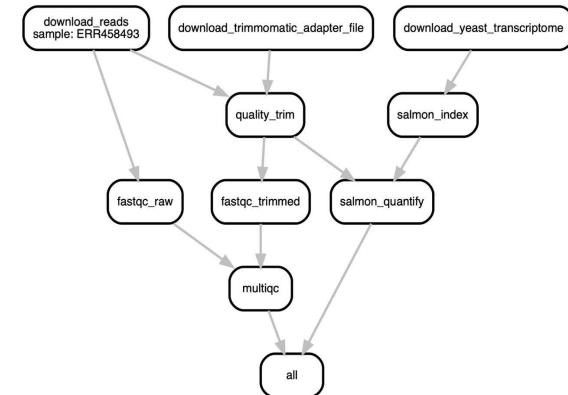
Configuration

A key Nextflow feature is the ability to decouple the [workflow implementation](#) from the [workflow configuration](#)

Configuration

A key Nextflow feature is the ability to decouple the **workflow implementation** from the workflow configuration

Workflow implementation describes the flow of data and operations to perform on that data. (visualized by DAG)



Configuration

A key Nextflow feature is the ability to decouple the workflow implementation from the **workflow configuration**

Workflow configuration is the settings required by the underlying execution platform

This enables the workflow to be portable, allowing it to run on different computational platforms such as an institutional HPC or cloud infrastructure, without needing to modify the workflow implementation

Nextflow configuration

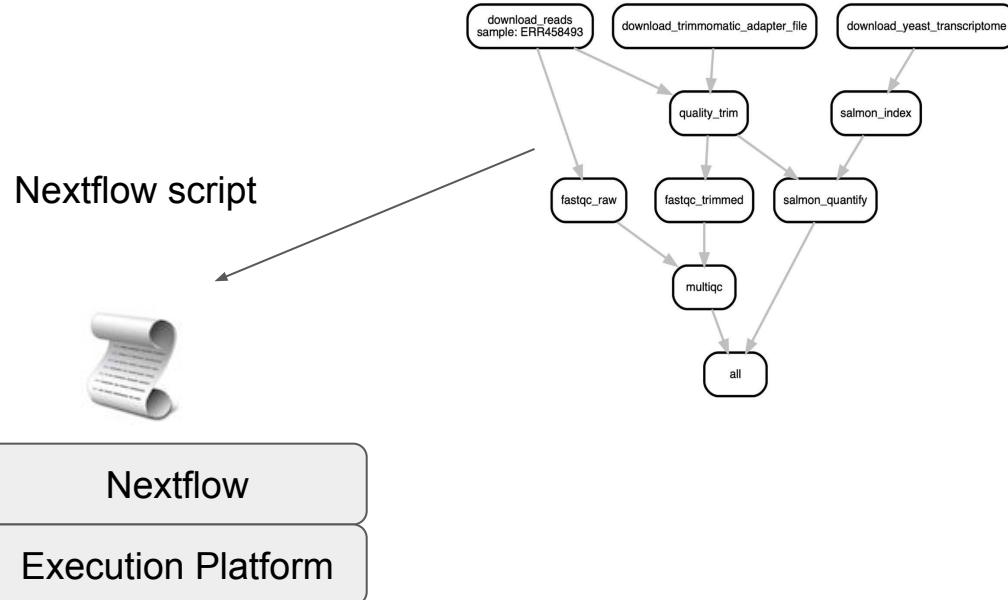
Workflow Configuration

Configuration

- Number of cpus
- Memory
- Software management
 - conda,docker
- Executor
 - HPC, Local machine

Workflow Implementation

Flow of data and operations to perform on data



Configuration

- How do I configure a Nextflow workflow?

Nextflow configuration: Process directives

Workflow Configuration

Process directive

Code

```
//process_directive.nf
nextflow.enable.dsl=2

process PRINTCHR {
    tag "tagging with chr$chr"
    cpus 1
    echo true
```

Nextflow script

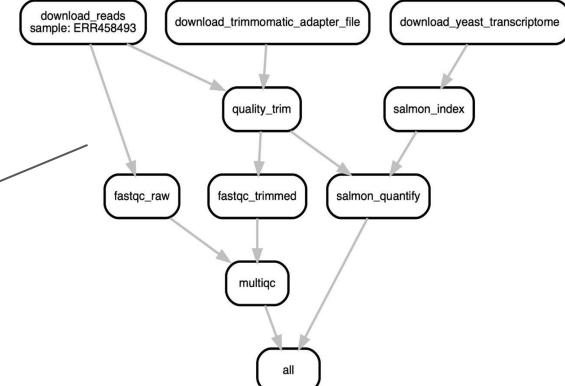


Nextflow

Execution Platform

Workflow Implementation

Flow of data and operations to perform on data



Nextflow configuration: Params

Workflow Configuration

Params



Code

```
{  
    "sleep": 5,  
    "input": "data/yeast/read...fq.gz"  
}
```

Process directive

Code

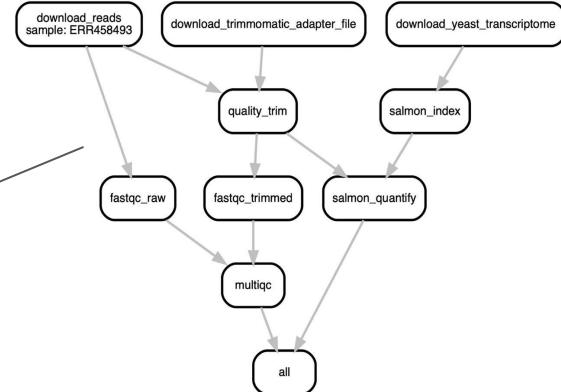
```
//process_directive.nf  
nextflow.enable.dsl=2  
  
process PRINTCHR {  
    tag "tagging with chr$chr"  
    cpus 1  
    echo true
```

Nextflow script



Workflow Implementation

Flow of data and operations to perform on data



Nextflow

Execution Platform

Nextflow configuration: Config files: .config

These configuration settings can be separated from the workflow implementation, into a configuration file.

Workflow Configuration

Configuration file

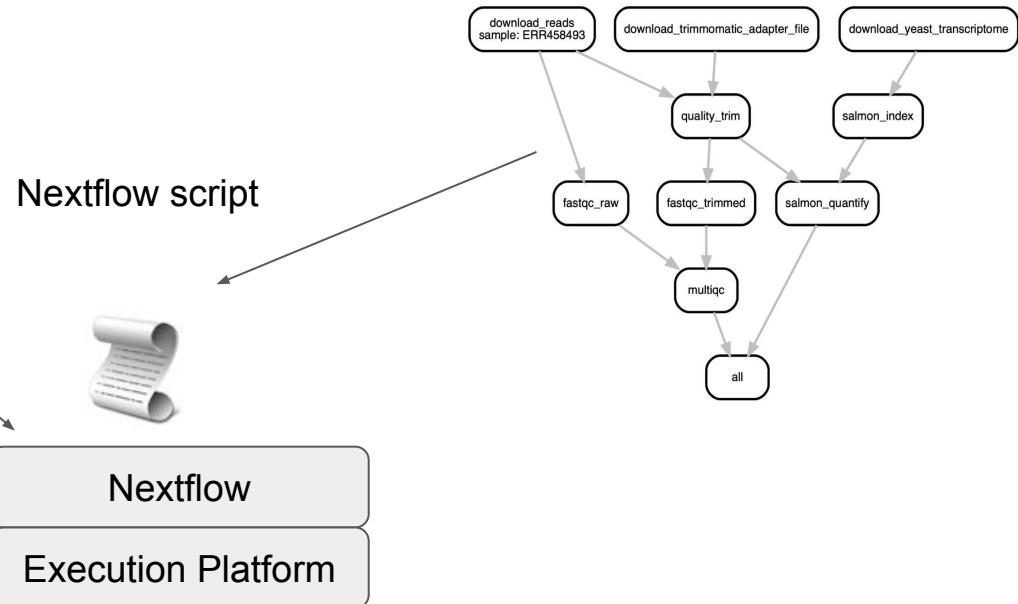


nextflow.config

- Params
- Number of cpus
- Memory
- Software environment or container
- Executor
 - HPC, Local machine

Workflow Implementation

Flow of data and operations to perform on data



Configuration: Config File

Configuration files are plain text file and normally have `.conf` file type extension

The file name `nextflow.config` has special meaning and is automatically read in if it is located in.

- `$HOME`
- Calling directory

Configuration File Settings

Settings in a configuration file are sets of name-value pairs

```
name = value
```

The `name` is a specific property to set,

The `value` can be anything you can assign to a variable for example,

- Strings; "some text"
- Booleans, true or false
- Numeric 1 or 1.234

Configuration File Settings

It is also possible to access any variable defined in the host using
\$environment_variable_name,
e.g. **\$PATH**, **\$HOME**, **\$PWD**

Code

```
// nextflow.config
my_home_dir = "$HOME"
```

Generally, variables and functions defined in a configuration file are not accessible from the workflow script.

Config scopes

Configuration settings can be organized in different scopes which govern the behaviour of different elements of the workflow.

- Parameters params
- Process
- Executors
- Software scopes
 - Conda
 - Docker
 - Singularity
- ...

Config scopes: Grouping Settings

Configuration can be written in either of two ways.

dot prefix

Code

```
params.input = ''          // The workflow parameter "input" is assigned an empty string to use as a default value
params.outdir = './results' // The workflow parameter "outdir" is assigned the value './results' to use by default.
```

Config scopes: Grouping Settings

Configuration can be written in either of two ways.

dot prefix

Code

```
params.input = ''           // The workflow parameter "input" is assigned an empty string to use as a default value
params.outdir = './results' // The workflow parameter "outdir" is assigned the value './results' to use by default.
```

Curly brackets {}

Code

```
params {
    input  = ''
    outdir = './results'
}
```

Accessing variables in your configuration file

Generally, variables and functions defined in a configuration file are not accessible from the workflow script. Only variables defined using the `params` scope and the `env` scope (without `env` prefix) can be accessed from the workflow script

Config file e.g. `nextflow.config`

Code

```
params {  
    input  = ''  
    outdir = './results'  
}
```

Nextflow script

Code

```
workflow {  
    MY_PROCESS->params.input  
}
```

Configuration Settings Priority

Configuration settings can be spread across several files. This also allows settings to be overridden by other configuration files. The priority of a setting is determined by the following order, ranked from highest to lowest.

1. Parameters specified on the command line (`--<param_name> value`).

highest

lowest



Configuration Settings Priority

- 
- 1. Parameters specified on the command line (`--<param_name> value`).
 - 2. Parameters provided using the `-params-file` option.
- highest
- lowest

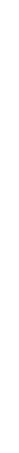
Configuration Settings Priority

- 
- highest
1. Parameters specified on the command line (`--<param_name> value`).
 2. Parameters provided using the `-params-file` option.
 3. Config file specified using the `-c my_config` option. E.g. `local.config`
- lowest

Configuration Settings Priority

highest

1. Parameters specified on the command line (`--<param_name> value`).
2. Parameters provided using the `-params-file` option.
3. Config file specified using the `-c my_config` option.
4. The config file named `nextflow.config` in the current directory.



lowest

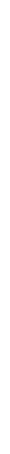
Configuration Settings Priority

- 
- highest
1. Parameters specified on the command line (`--<param_name> value`).
 2. Parameters provided using the `-params-file` option.
 3. Config file specified using the `-c my_config` option.
 4. The config file named `nextflow.config` in the current directory.
 5. The config file named `nextflow.config` in the workflow project directory
 - a. `($projectDir: the directory where the script to be run is located)`.
- lowest

Configuration Settings Priority

- 
- highest
1. Parameters specified on the command line (`--<param_name> value`).
 2. Parameters provided using the `-params-file` option.
 3. Config file specified using the `-c my_config` option.
 4. The config file named `nextflow.config` in the current directory.
 5. The config file named `nextflow.config` in the workflow project directory
 - a. `$projectDir`: the directory where the script to be run is located.
 6. The config file `$HOME/.nextflow/config`.
- lowest

Configuration Settings Priority

- 
- highest
1. Parameters specified on the command line (`--<param_name> value`).
 2. Parameters provided using the `-params-file` option.
 3. Config file specified using the `-c my_config` option.
 4. The config file named `nextflow.config` in the current directory.
 5. The config file named `nextflow.config` in the workflow project directory
 - a. (`$projectDir`: the directory where the script to be run is located).
 6. The config file `$HOME/.nextflow/config`.
 7. Values defined within the workflow script itself (e.g., `main.nf`).
- lowest

Determine script output

Determine the outcome of the following script executions. Given the script `print_message.nf` :

Code

```
nextflow.enable.dsl = 2

params.message = 'hello'

workflow {
    PRINT_MESSAGE(params.message)
}

process PRINT_MESSAGE {
    echo true

    input:
    val my_message

    script:
    """
    echo $my_message
    """

}
```

and configuration (`print_message.config`):

Code

```
params.message = 'Are you tired?'
```

What is the outcome of the following commands?

1. `nextflow run print_message.nf`
2. `nextflow run print_message.nf --message '?Que tal?'`
3. `nextflow run print_message.nf -c print_message.config`
4. `nextflow run print_message.nf -c print_message.config --message '?Que tal?'`

Configuration: Process Scope

How do I assign different resources to different processes using a config file?

Configuring Global Process behaviour

The `process` configuration scope allows the setting of any process directives in the Nextflow configuration file.

Code

```
// nextflow.config
process {
    cpus = 2
    memory = 8.GB
    time = '1 hour'
    publishDir = [ path: params.outdir, mode: 'copy' ]
}
```

process scope { }

These settings are applied to all processes in the workflow.

Process selectors

Configuration: Specific process behaviour: `withName`:

The resources for a specific process can be defined using `withName`: followed by the process name

Code

```
// process_resources.config
process {
    withName: INDEX {
        cpus = 4
        memory = 8. GB
    }
    withName: FASTQC {
        cpus = 2
        memory = 4. GB
    }
}
```

Configuration: Groups of Processes: `withLabel`:

Annotate the processes using the `label` directive

Code

```
// configuration_process_labels.nf
nextflow.enable.dsl=2

process P1 {
    label "big_mem"
    script:
    """
    echo P1: Using $task.cpus cpus and $task.memory memory.
    """
}
process P2 {
    label "big_mem"
    script:
    """
    echo P2: Using $task.cpus cpus and $task.memory memory.
    """
}
workflow {
    P1()
    P2()
}
```

Code

```
// configuration_process-labels.config
process {
    withLabel: big_mem {
        cpus = 16
        memory = 64.GB
    }
}
```

Any process with the label `big_mem` will have the same resources

Configuration: Process selectors: priority

When mixing generic process configuration and selectors, the following priority rules are applied (from highest to lowest):

- `withName` selector definition.

Code

```
// process_resources.config
process {
    withName: INDEX {
        cpus = 4
        memory = 8. GB
    }
    withName: FASTQC {
        cpus = 2
        memory = 4. GB
    }
}
```

Configuration: Process selectors: priority

When mixing generic process configuration and selectors, the following priority rules are applied (from highest to lowest):

- `withName` selector definition.
- `withLabel` selector definition.

Code

```
// configuration_process_labels.nf
nextflow.enable.dsl=2

process P1 {
    label "big_mem"
    script:
    """
    echo P1: Using $task.cpus cpus and $task.memory memory.
    """
}

process P2 {
    label "big_mem"
    script:
    """
    echo P2: Using $task.cpus cpus and $task.memory memory.
    """
}

workflow {
    P1()
    P2()
}
```

Configuration: Process selectors: priority

When mixing generic process configuration and selectors, the following priority rules are applied (from highest to lowest):

- `withName` selector definition.
- `withLabel` selector definition.
- Process specific `directive` defined in the workflow script.

Configuration: Process selectors: priority

When mixing generic process configuration and selectors, the following priority rules are applied (from highest to lowest):

- `withName` selector definition.
- `withLabel` selector definition.
- Process specific `directive` defined in the workflow script.
- Process generic `process` configuration. e.g. in `nextflow.config` file

Configuration: Dynamic expressions: calculating resources

Code

```
process FASTQC {  
  
    input:  
        tuple val(sample), path(reads)  
  
    script:  
        ....  
        fastqc -t $task.cpus $reads  
        ....  
}
```

Code

```
// nextflow.config  
process {  
    withName: FASTQC {  
        cpus = 2  
        memory = { 2.GB * task.cpus } ← {dynamic expressions}  
        publishDir = { "fastqc/$sample" } ← {dynamic expressions}  
    }  
}
```

{dynamic expressions}

Process selectors

Create a Nextflow config, `process-selector.config`, specifying different `cpus` and `memory` resources for the two processes `P1` (cpus 1 and memory 2.GB) and `P2` (cpus 2 and memory 1.GB), where `P1` and `P2` are **defined** as follows:

Code

```
// process-selector.nf
nextflow.enable.dsl=2

process P1 {
    echo true

    script:
    """
    echo P1: Using $task.cpus cpus and $task.memory memory.
    """

}

process P2 {
    echo true

    script:
    """
    echo P2: Using $task.cpus cpus and $task.memory memory.
    """

}

workflow {
    P1()
    P2()
}
```

Code

```
// process-selector.config
process {
    withName: P1 {
        cpus = 1
        memory = 2.GB
    }
    withName: P2 {
        cpus = 2
        memory = 1.GB
    }
}
```

Bash

```
$ nextflow run process-selector.nf -c process-selector.config -process.echo
```

Output

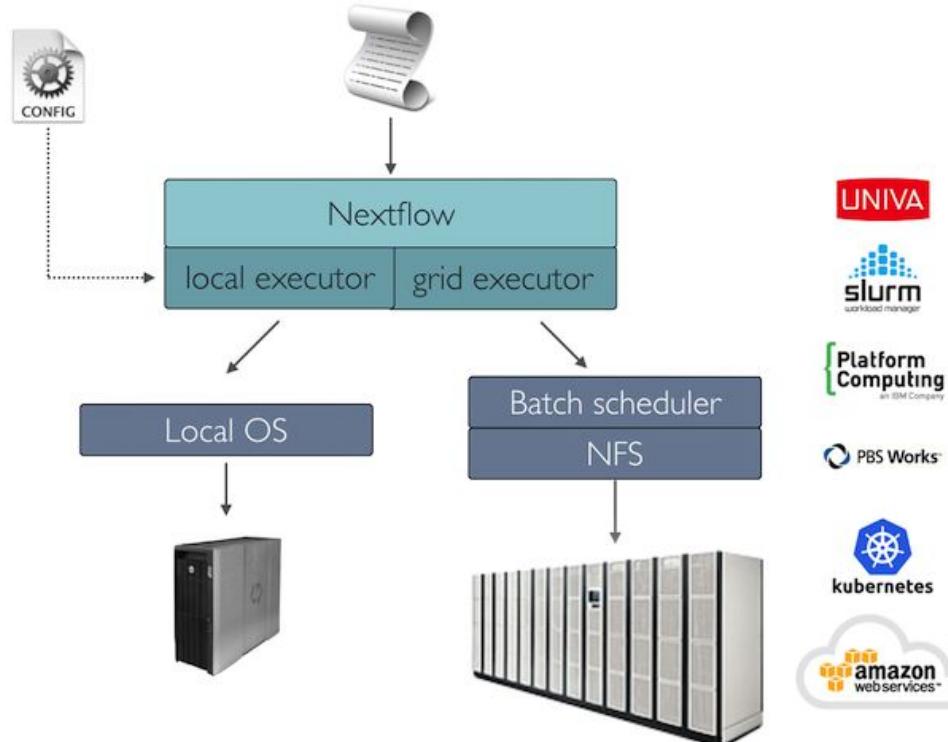
```
N E X T F L O W ~ version 21.04.0
```

```
Launching `process-selector.nf` [clever_borg] -
revision: e765b9e62d
executor > local (2)
[de/86cef0] process > P1 [100%] 1 of 1 ✓
[bf/8b332e] process > P2 [100%] 1 of 1 ✓
P2: Using 2 cpus and 1 GB memory.
```

```
P1: Using 1 cpus and 2 GB memory.
```

Executors: Configuring the execution platforms

Nextflow supports a wide range of execution platforms, from running locally, to running on HPC clusters or cloud infrastructures.



Configuration: executor scope

The default executor configuration is defined within the `executor` scope

Code

```
// nextflow.config
executor {
    name = 'sge'
    queueSize = 10
}
```

Configuration: executor scope

The `process.executor` directive allows you to override the executor to be used by a specific process.

Code

```
// nextflow.config
executor {
    name = 'sge'
    queueSize = 10
}
```

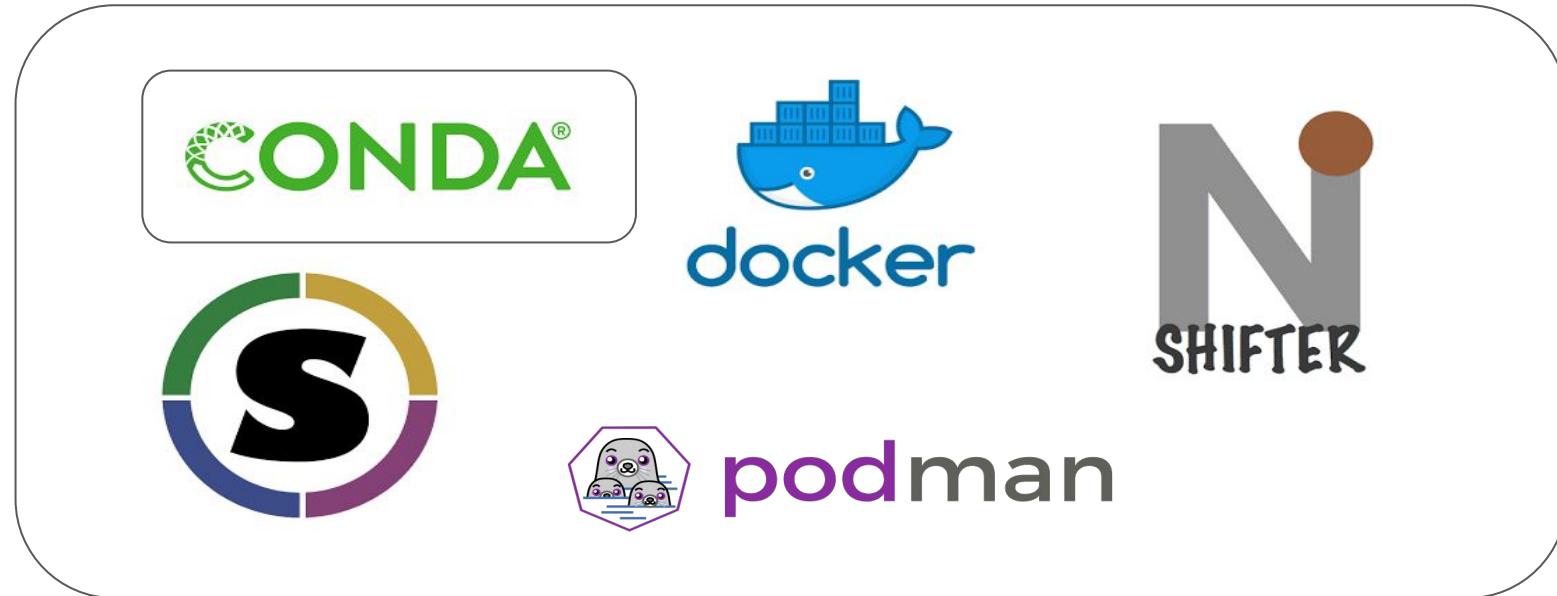
Code

```
//nextflow.config
executor {
    name = 'sge'
    queueSize = 10
}
process {
    withLabel: 'short' {
        executor = 'local'
    }
}
```

`process.executor` allows override

Configuring software: Containers and Software environments

An important feature of Nextflow is the ability to manage software using different technologies.



- Portability
- Reproducibility

Software configuration: `conda` scope

Software environment specification is managed from the `process` scope,

Code

```
process {
```

A Conda environment can be configured in several ways:

optional `conda` scope <https://www.nextflow.io/docs/latest/config.html#config-conda>

Software configuration: `conda` scope

Software environment specification is managed from the `process` scope,

Code

```
process {  
    conda = "/home/user/miniconda3/envs/my_conda_env"
```

A Conda environment can be configured in several ways:

- Provide a path to an existing Conda environment.

Software configuration: **conda** scope

Software environment specification is managed from the `process` scope,

Code

```
process {  
    conda = "/home/user/miniconda3/envs/my_conda_env"  
    withName: FASTQC {  
        conda = "environment.yml"  
    }  
}
```

A Conda environment can be configured in several ways:

- Provide a path to an existing Conda environment.
- Provide a path to a Conda environment specification file (written in YAML).

Software configuration: `conda` scope

Software environment specification is managed from the `process` scope,

Code

```
process {
    conda = "/home/user/miniconda3/envs/my_conda_env"
    withName: FASTQC {
        conda = "environment.yml"
    }
    withName: SALMON {
        conda = "bioconda:::salmon=1.5.2"
    }
}
```

A Conda environment can be configured in several ways:

- Provide a path to an existing Conda environment.
- Provide a path to a Conda environment specification file (written in YAML).
- Specify the software package(s) using the `<channel>::<package_name>=<version>` syntax (separated by spaces), which then builds the Conda environment when the process is run.

Define a software requirement in the configuration file using conda

Create a config file for the Nextflow script `configure_fastp.nf`. Add a conda directive for the process name `FASTP` that includes the bioconda package `fastp`, version 0.12.4-0. Hint You can specify the conda packages using the syntax `<channel>::<package_name>=<version>` e.g. `bioconda::salmon=1.5.2`. Run the Nextflow script `configure_fastp.nf` with the configuration file using the `-c` option.

Code

```
// configure_fastp.nf
nextflow.enable.dsl = 2

params.input = "data/yeast/reads/ref1_1.fq.gz"

workflow {
    FASTP( Channel.fromPath( params.input ) ).out.view()
}

process FASTP {

    input:
    path read

    output:
    stdout

    script:
    """
    fastp -A -i ${read} -o out.fq 2>&1
    """
}
```

Solution

Code

```
// fastp.config
process {
    withName: 'FASTP' {
        conda = "bioconda:::fastp=0.12.4-0"
    }
}
```

Bash

```
nextflow run configure_fastp.nf -c fastp.config -process.echo
```

Output

```
N E X T F L O W ~ version 21.04.0
Launching `configuration_fastp.nf` [berserk_jepsen] - revision: 28fadd2486
executor > local (1)
[c1/c207d5] process > FASTP (1) [100%] 1 of 1 ✓
Creating Conda env: bioconda:::fastp=0.12.4-0 [cache /home/training/work/conda/env-a7a3a0d820eb46bc41ebf4f72d955e5f]
ref1_1.fq.gz 58708
Read1 before filtering:
total reads: 14677
total bases: 1482377

Q20 bases: 1466210(98.9094%)
Q30 bases: 1415997(95.5221%)

Read1 after filtering:
total reads: 14671
total bases: 1481771
Q20 bases: 1465900(98.9289%)
Q30 bases: 1415769(95.5457%)

Filtering result:
reads passed filter: 14671
reads failed due to low quality: 6
reads failed due to too many N: 0
reads failed due to too short: 0

JSON report: fastp.json
HTML report: fastp.html
```

Configuration Profiles

How do I separate and provide configuration for different computational systems?

Configuration **profiles** (grouping config settings)

Code

```
//configuration_profiles.config
profiles {

    standard {
        params.genome = '/local/path/ref.fasta'
        process.executor = 'local'
    }
}
```

Dot notation

```
$ nextflow run main.nf -profile standard -c configuration_profiles.config
```

Configuration profiles (grouping config settings)

Code

```
//configuration_profiles.config
profiles {

    standard {
        params.genome = '/local/path/ref.fasta'
        process.executor = 'local'
    }

    cluster {
        params.genome = '/data/stared/ref.fasta'
        process.executor = 'sge'
        process.queue = 'long'
        process.memory = '10GB'
        process.conda = '/some/path/env.yml'
    }
}
```

Unique name

```
$ nextflow run main.nf -profile cluster -c configuration_profiles.config
```

Configuration profiles (grouping config settings)

Code

```
//configuration_profiles.config
profiles {

    standard {
        params.genome = '/local/path/ref.fasta'
        process.executor = 'local'
    }

    cluster {
        params.genome = '/data/stared/ref.fasta'
        process.executor = 'sge'
        process.queue = 'long'
        process.memory = '10GB'
        process.conda = '/some/path/env.yml'
    }

    cloud {
        params.genome = '/data/stared/ref.fasta'
        process.executor = 'awsbatch'
        process.container = 'cbcrg/imagex'
        docker.enabled = true
    }

}
```

```
$ nextflow run main.nf -profile cloud -c configuration_profiles.config
```

Configuration: Key Points

- Nextflow configuration can be managed using a Nextflow configuration file.
- Nextflow configuration files are plain text files containing a set of properties.
- You can define process specific settings, such as cpus and memory, within the `process` scope.
- You can assign different resources to different processes using the `process` selectors `withName` or `withLabel`.
- You can define a profile for different configurations using the `profiles` scope. These profiles can be selected when launching a pipeline execution by using the `-profile` command-line option
- Nextflow configuration settings are evaluated in the order they are read-in.
- Workflow configuration settings can be inspected using `nextflow config <script> [options]`.

Lunch Break

Day 2 Afternoon

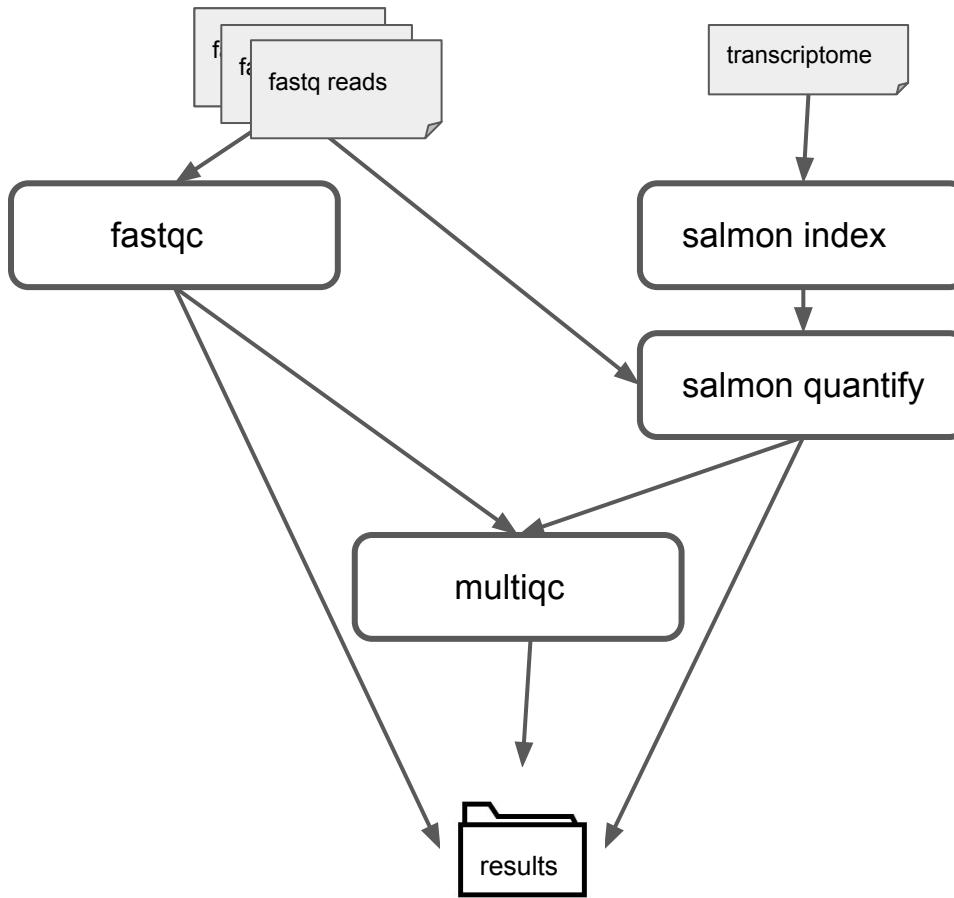
Checkpointing and Caching

https://ggrimes.github.io/workflows-nextflow/13-workflow_checkpoint_caching/index.html

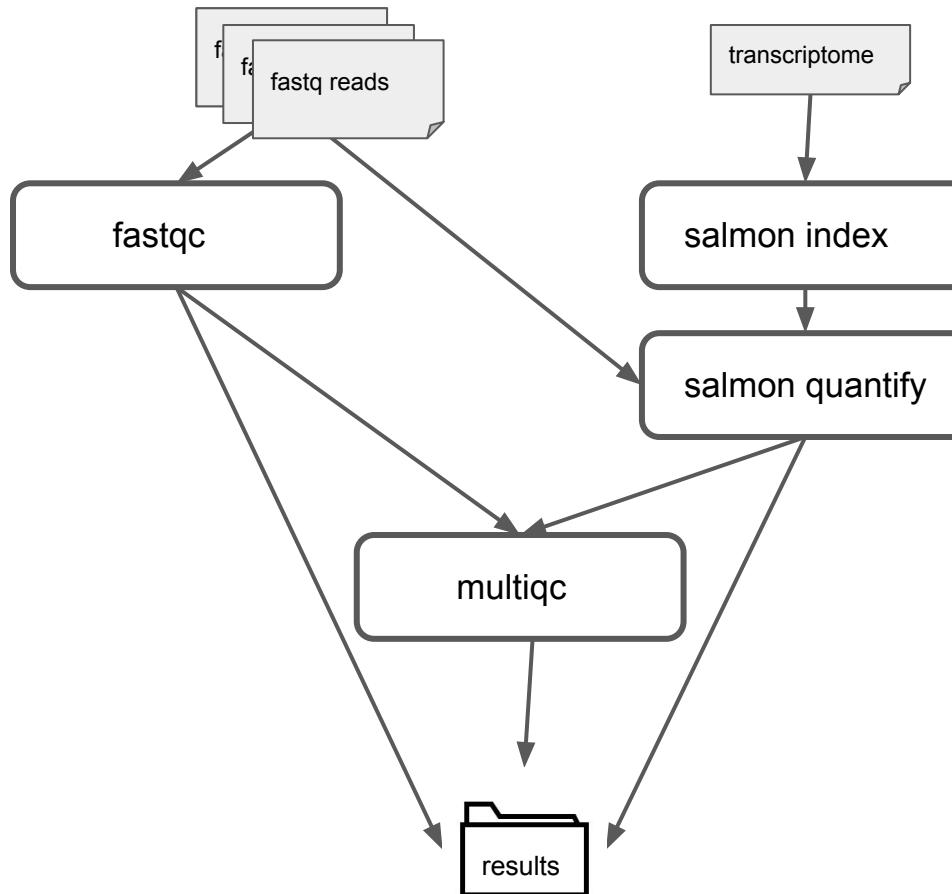
Key features of a Workflow Management Systems

- Run time management
- Software management
- Portability & Interoperability
- Reproducibility
- **Re-entrancy**
 - Checkpoint and caching

Checkpointing and Caching: Re-entrancy

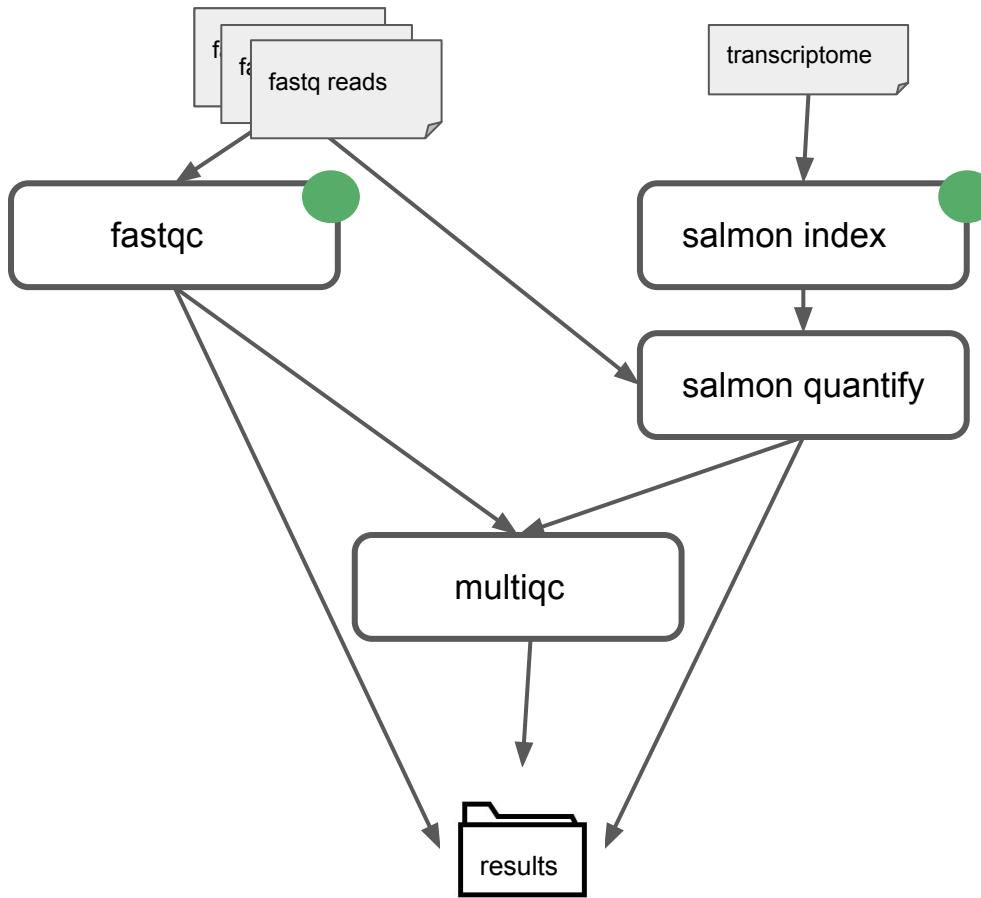


Checkpointing and Caching: Re-entrancy

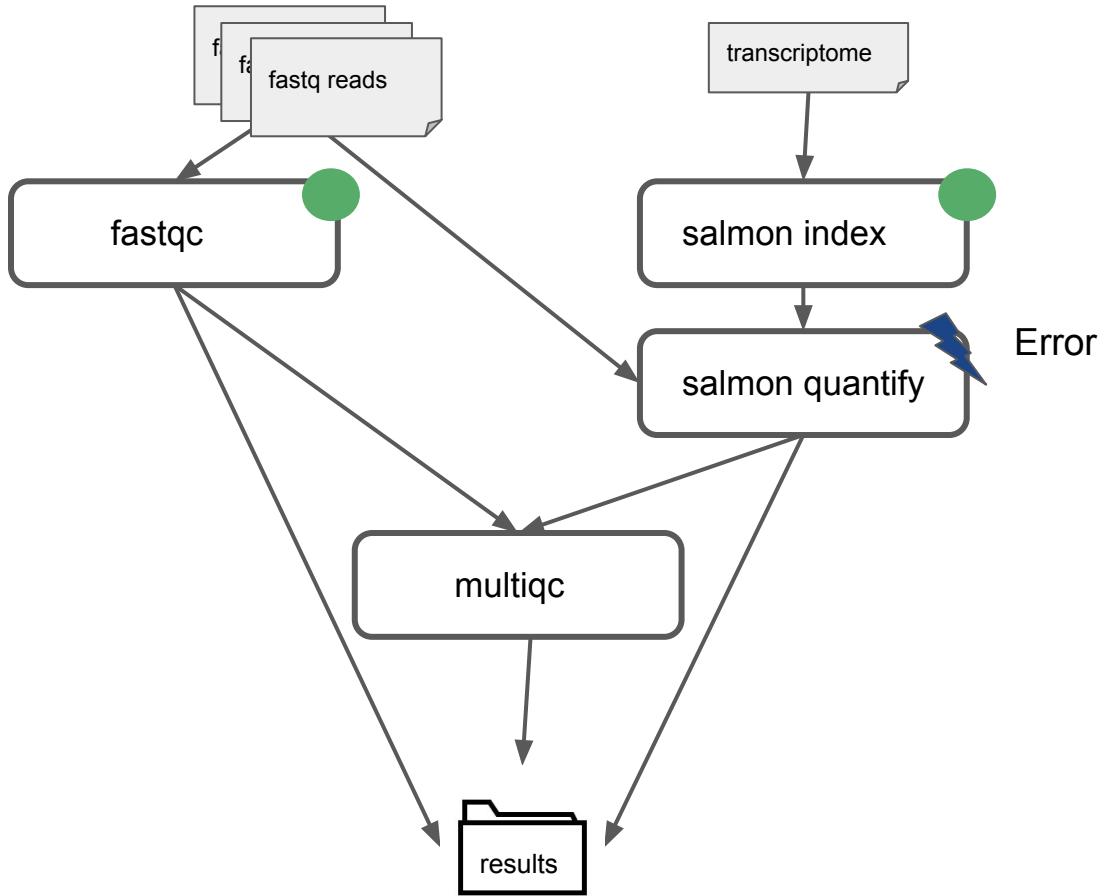


```
$ nextflow run rnaseq-pipe.nf --reads "data/*_{1,2}.fq"
```

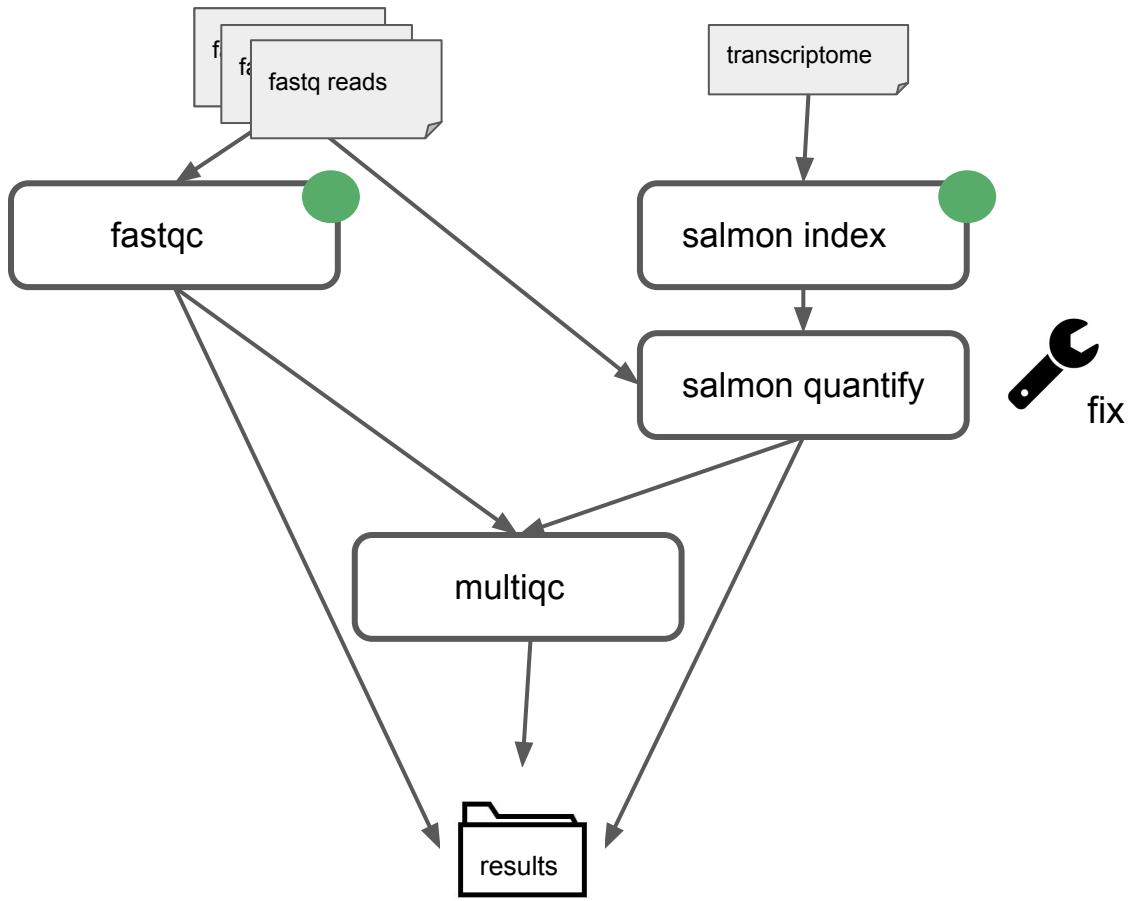
Checkpointing and Caching: Re-entrancy



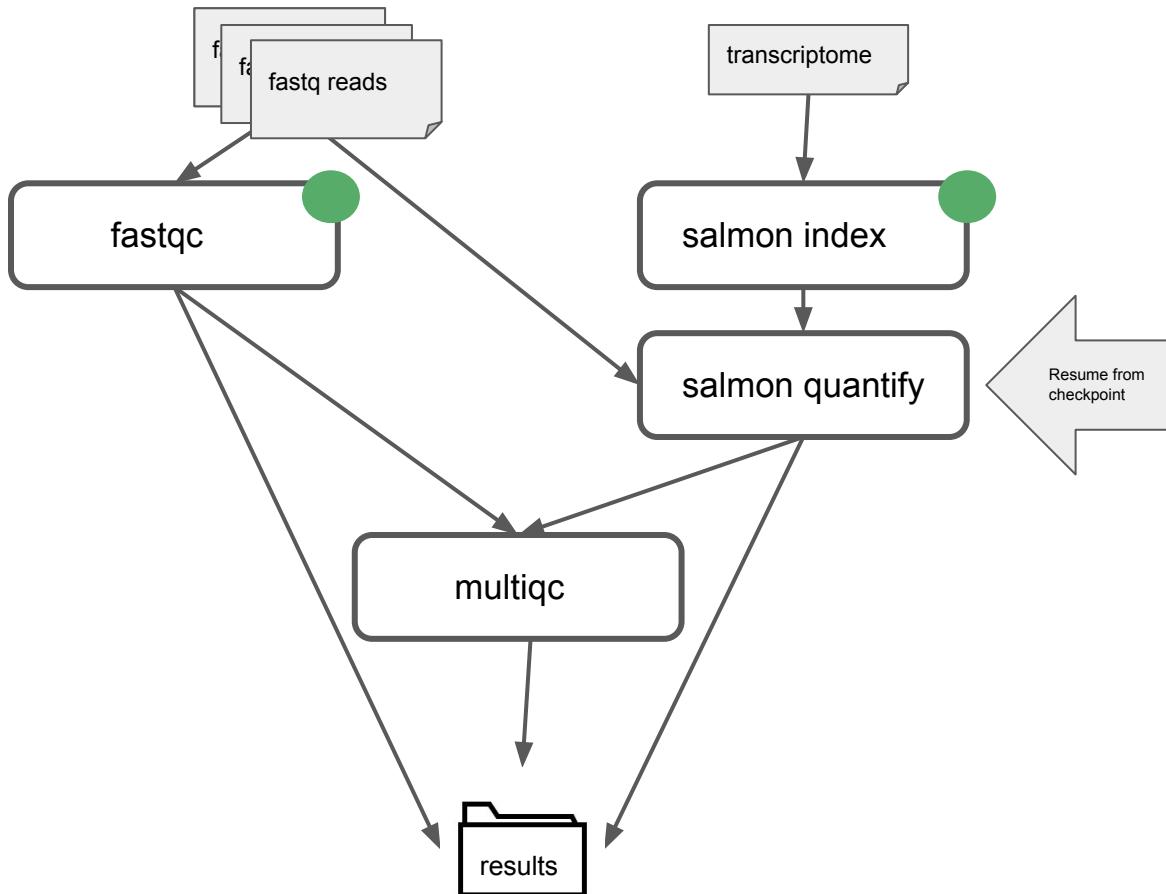
Checkpointing and Caching: Re-entrancy: Error



Checkpointing and Caching: Re-entrancy

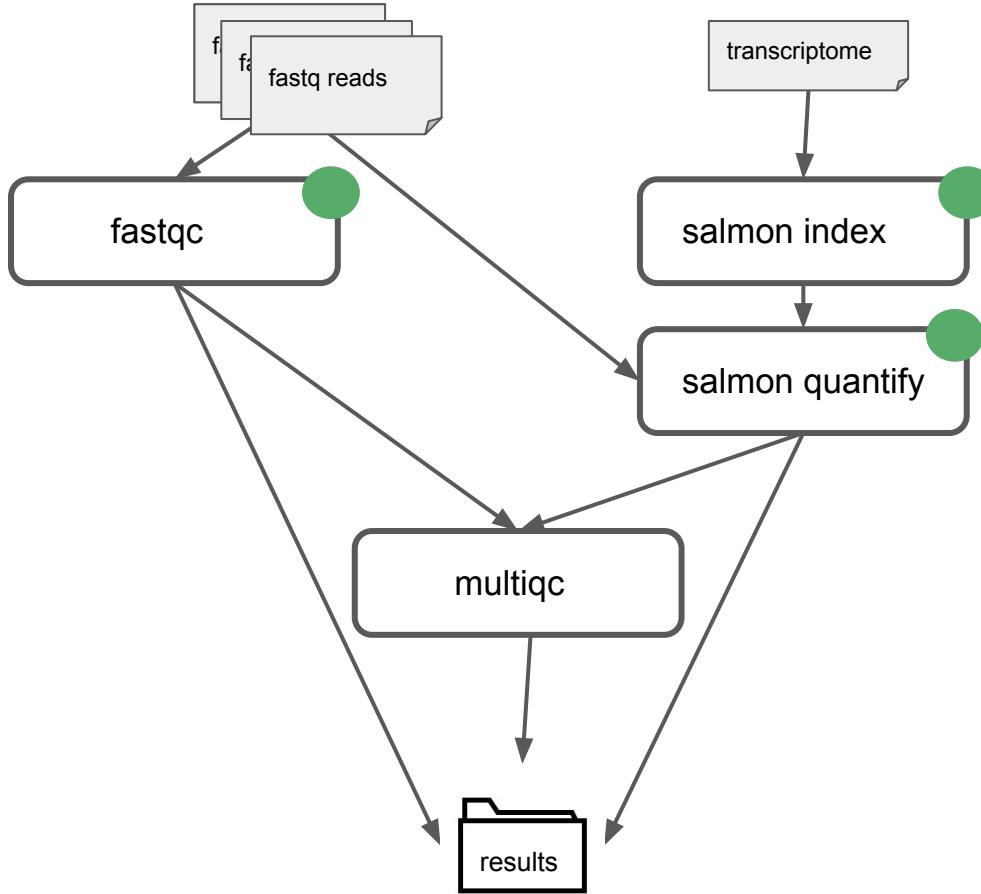


Checkpointing and Caching: Re-entrancy - resume

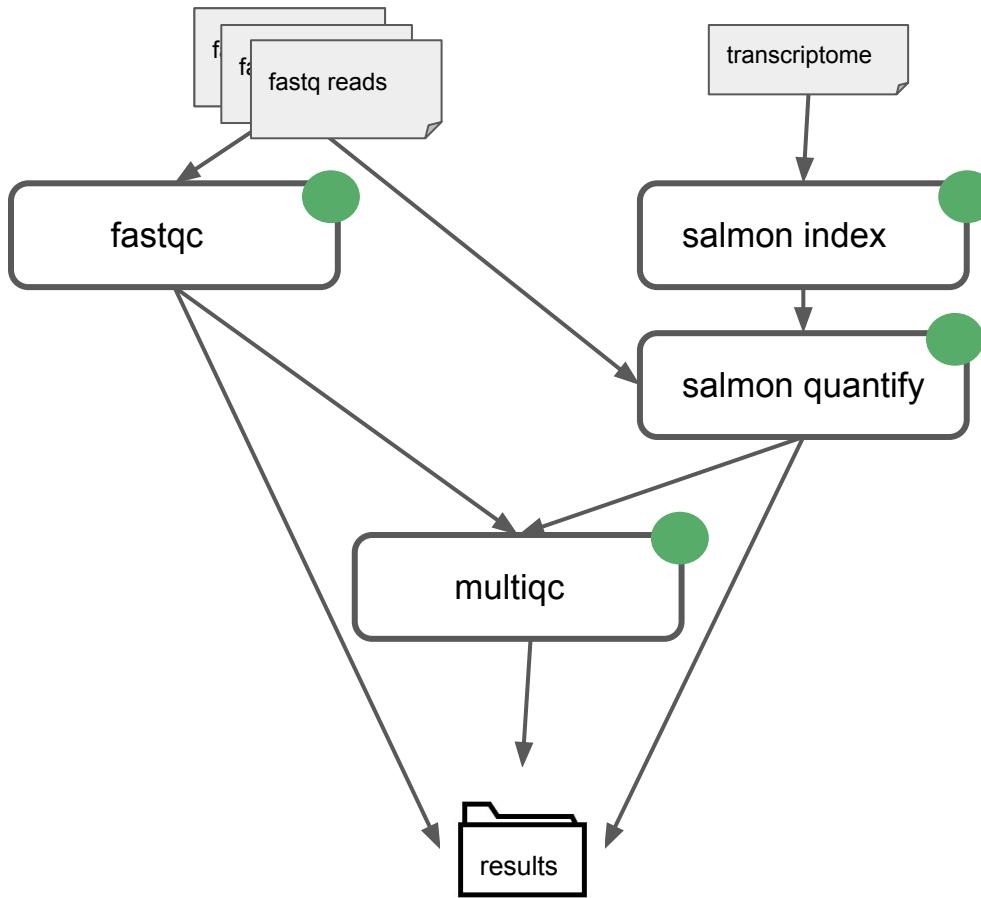


```
$ nextflow run rnaseq-pipe.nf --reads "data/*_{1,2}.fq" -resume
```

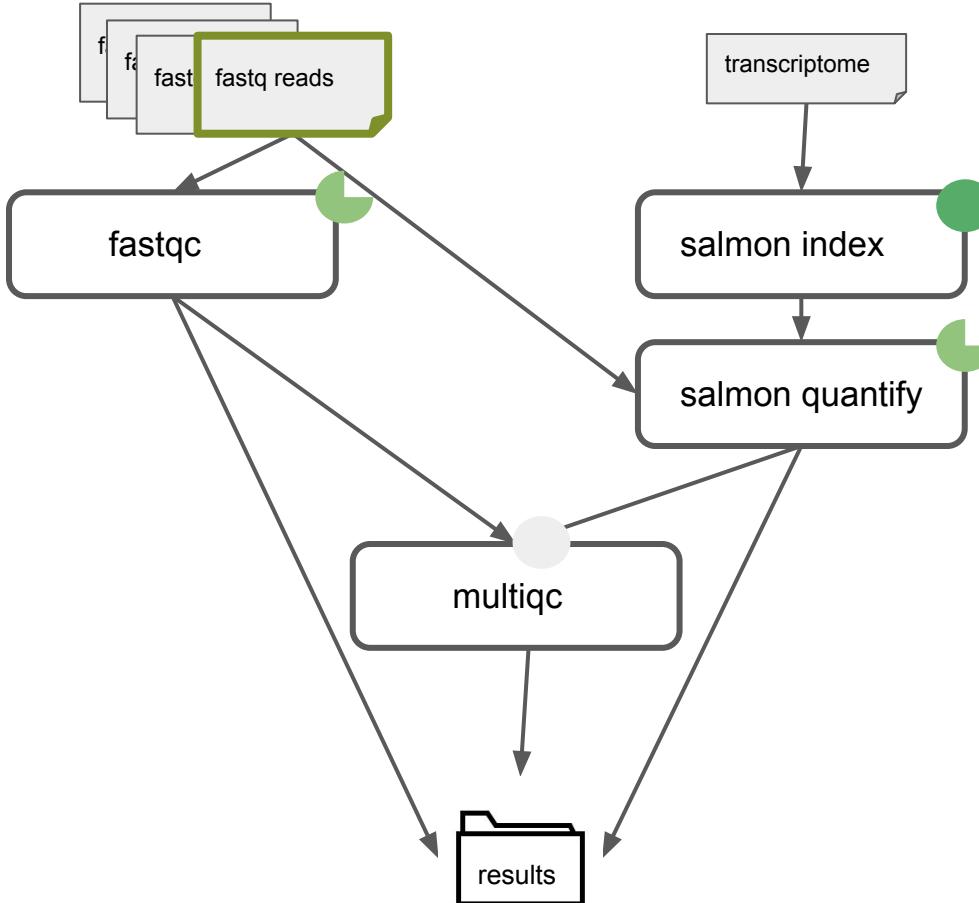
Checkpointing and Caching: Re-entrancy



Checkpointing and Caching: Re-entrancy

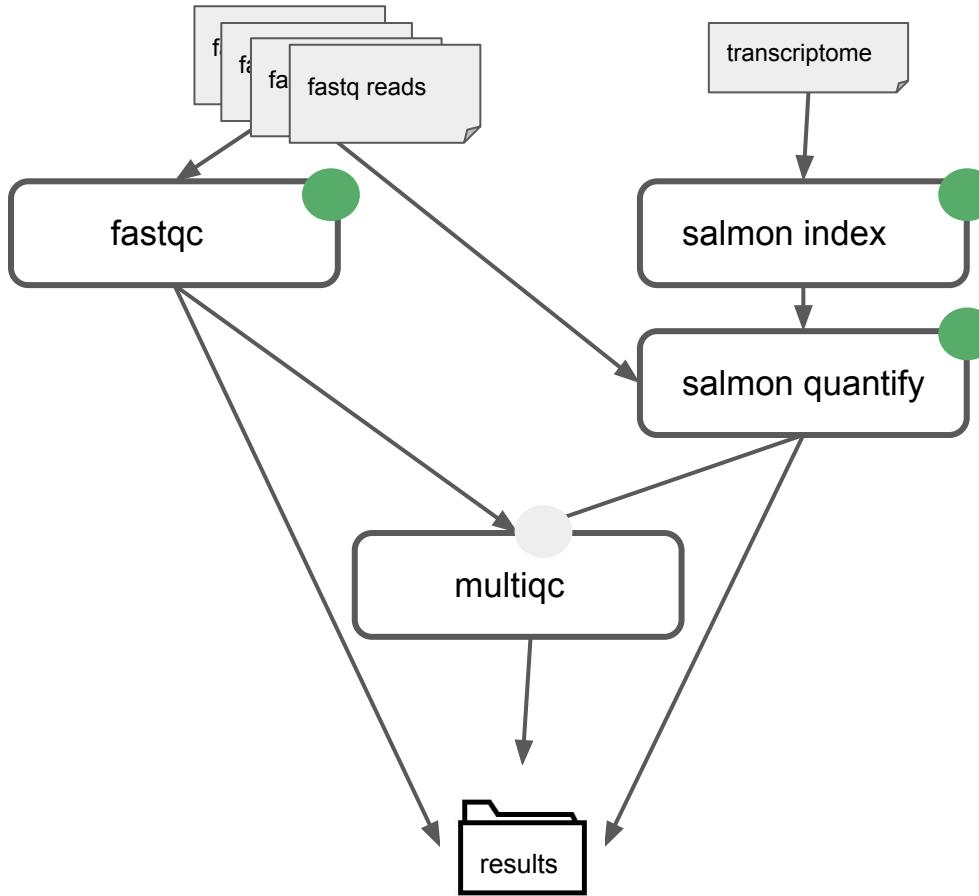


Checkpointing and Caching: Re-entrancy: Adding data

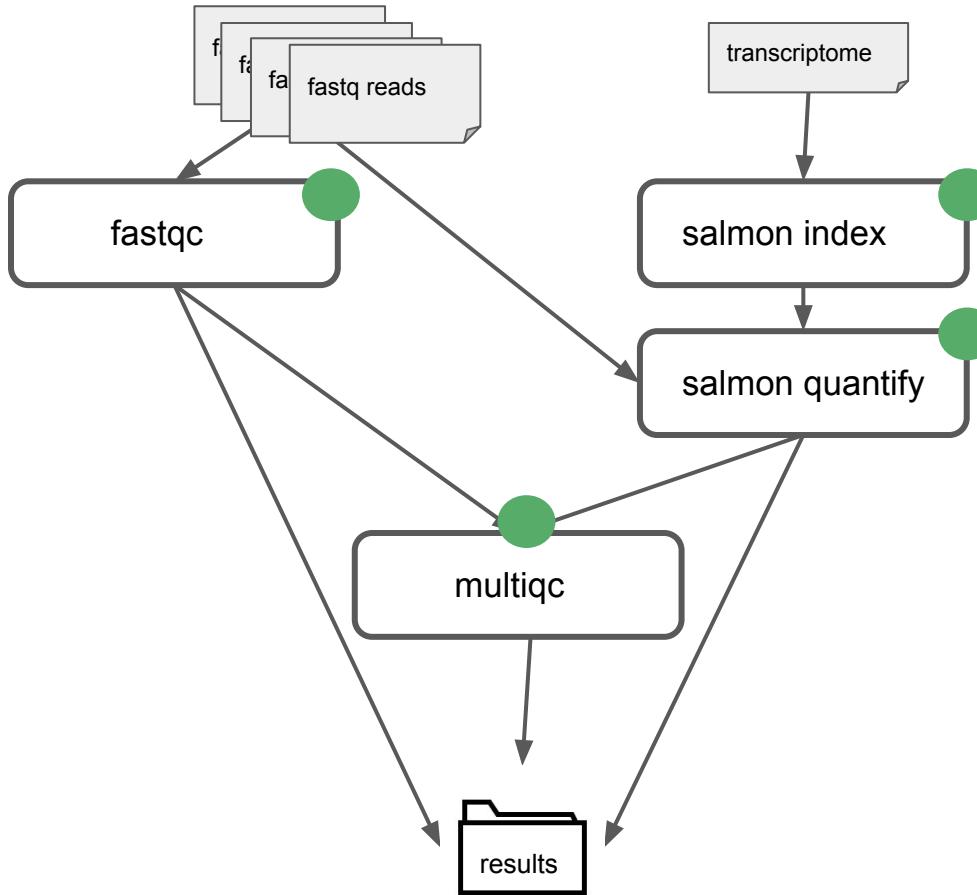


```
$ nextflow run rnaseq-pipe.nf --reads "data/*_{1,2}.fq" -resume
```

Checkpointing and Caching: Re-entrancy: New data



Checkpointing and Caching: Re-entrancy: New data



Example, wc .nf

- Counts the number of lines in each fastq file

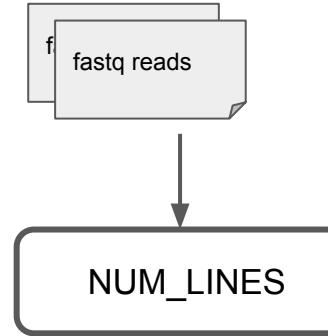
Exercise: wc.nf

Resume a pipeline

Resume the Nextflow script `wc.nf` by re-running the command and adding the parameter `-resume` and the parameter `--input 'data/yeast/reads/temp33*'`:

Solution

How does -resume work?

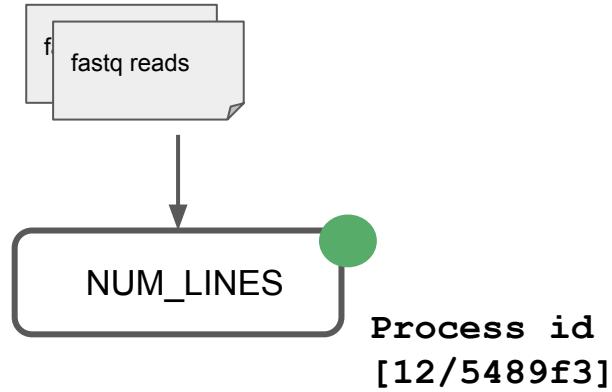


```
$ nextflow run wc.nf --input 'data/yeast/reads/ref1*.fq.gz'
```

How does -resume work?

```
work/
└── 12
    └── 5489f3c7dbd521c0e43f43b4c1f352
        ├── .command.begin
        ├── .command.err
        ├── .command.log
        ├── .command.out
        ├── .command.run
        ├── .command.sh
        ├── .exitcode
        └── temp33_1_2.fq.gz -> /home/training/data/yeast/reads/temp33_1_2.fq.gz
```

Unique id

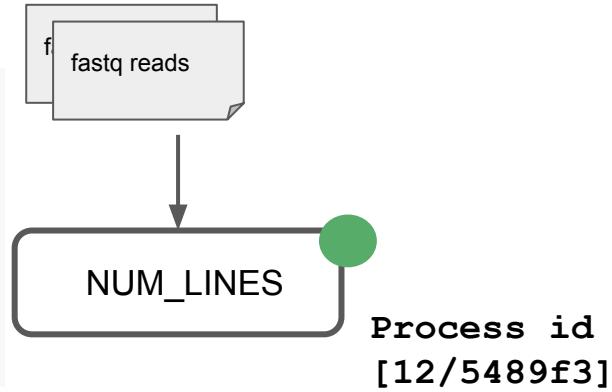


```
$ nextflow run wc.nf --input 'data/yeast/reads/ref1*.fq.gz'
```

How does -resume work?

```
work/
└── 12
    └── 5489f3c7dbd521c0e43f43b4c1f352
        ├── .command.begin
        ├── .command.err
        ├── .command.log
        ├── .command.out
        ├── .command.run
        ├── .command.sh
        ├── .exitcode
        └── temp33_1_2.fq.gz -> /home/training/data/yeast/reads/temp33_1_2.fq.gz
```

Unique id
↳



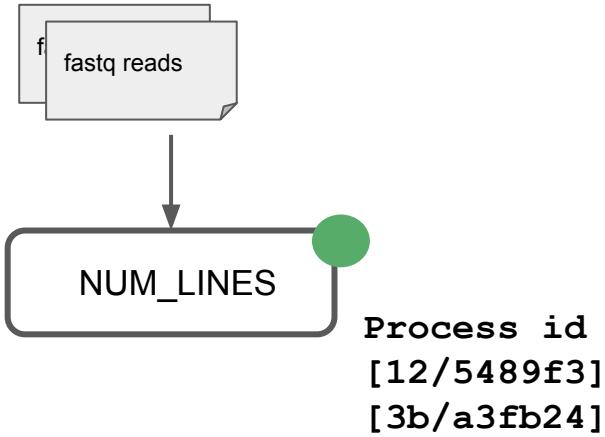
Unique ids made up of

- Inputs values
- Input files
- Command line string
- Container ID
- Conda environment
- Environment modules
- Any executed scripts in the bin directory

```
$ nextflow run wc.nf --input 'data/yeast/reads/ref1*.fq.gz'
```

How does -resume work?

```
work/
└── 12
    └── 5489f3c7dbd521c0e43f43b4c1f352
        ├── .command.begin
        ├── .command.err
        ├── .command.log
        ├── .command.out
        ├── .command.run
        ├── .command.sh
        └── .exitcode
            temp33_1_2.fq.gz -> /home/training/data/yeast/reads/temp33_1_2.fq.gz
└── 3b
    └── a3fb24ad3242e4cc8e5aa0c24d174b
        ├── .command.begin
        ├── .command.err
        ├── .command.log
        ├── .command.out
        ├── .command.run
        ├── .command.sh
        └── .exitcode
            temp33_2_1.fq.gz -> /home/training/data/yeast/reads/temp33_2_1.fq.gz
```



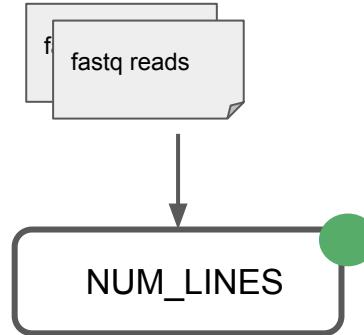
```
$ nextflow run wc.nf --input 'data/yeast/reads/ref1*.fq.gz'
```

How does -resume work?

```
work/
  12
    └── 5489f3c7dbd521c0e-
        ├── .command.begin
        ├── .command.err
        ├── .command.log
        ├── .command.out
        ├── .command.run
        ├── .command.sh
        └── .exitcode
        temp33_1_2.fq.gz -> /home/training/data/yeast/reads/temp33_1_2.fq.gz

  3b
    └── a3fb24ad3242e4cc8e5aa0c24d174b
        ├── .command.begin
        ├── .command.err
        ├── .command.log
        ├── .command.out
        ├── .command.run
        ├── .command.sh
        └── .exitcode
        temp33_2_1.fq.gz -> /home/training/data/yeast/reads/temp33_2_1.fq.gz
```

1. The working directory exists
2. It contains a valid command exit status



```
$ nextflow run wc.nf --input 'data/yeast/reads/ref1*.fq.gz' -resume
```

How does -resume work? Exit Code

- 
1. The working directory exists
 2. It contains a valid command exit status

- Every Linux or Unix command executed by the shell script or user, has an exit status.
- The exit status is an integer number.
- For the bash shell's, a command which exits with a zero (0) exit status has succeeded.
- A non-zero (1-255) exit status indicates failure.
- E.g. If a command is not found, the child process created to execute it returns a status of 127

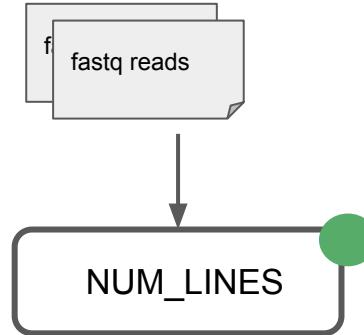
```
$ nextflow run wc.nf --input 'data/yeast/reads/ref1*.fq.gz' -resume
```

How does -resume work?

```
work/
  12
    └── 5489f3c7dbd521c0e
        ├── .command.begin
        ├── .command.err
        ├── .command.log
        ├── .command.out
        ├── .command.run
        ├── .command.sh
        └── .exitcode
            temp33_1_2.fq.gz -> /home/training/data/yeast/reads/temp33_1_2.fq.gz

  3b
    └── a3fb24ad3242e4cc8e5aa0c24d174b
        ├── .command.begin
        ├── .command.err
        ├── .command.log
        ├── .command.out
        ├── .command.run
        ├── .command.sh
        └── .exitcode
            temp33_2_1.fq.gz -> /home/training/data/yeast/reads/temp33_2_1.fq.gz
```

1. The working directory exists
2. It contains a valid command exit status
3. It contains the expected output files.

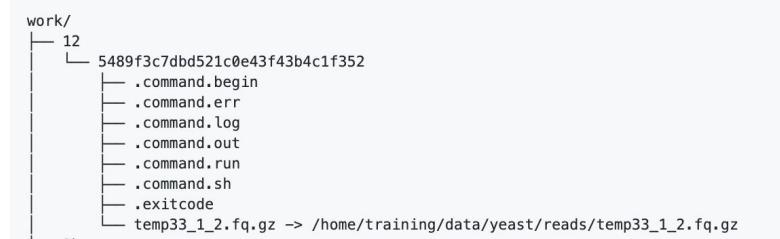


```
$ nextflow run wc.nf --input 'data/yeast/reads/ref1*.fq.gz' -resume
```

What is in Task execution directory ?

The task execution directory contains:

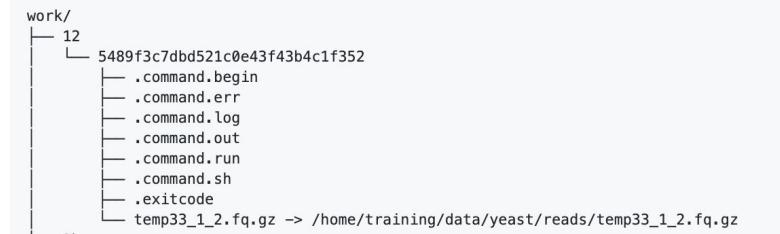
- `.command.sh`: The command script.



What is in Task execution directory ?

The task execution directory contains:

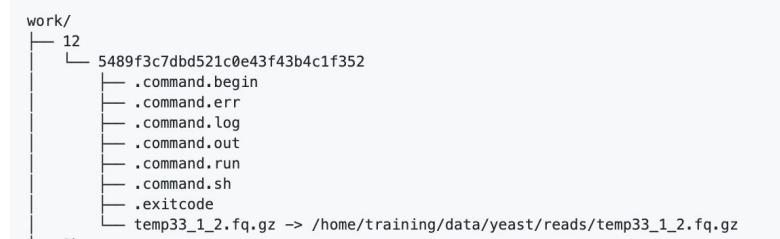
- `.command.sh`: The command script.
- `.command.run`: The command wrapped used to run the job.



What is in Task execution directory ?

The task execution directory contains:

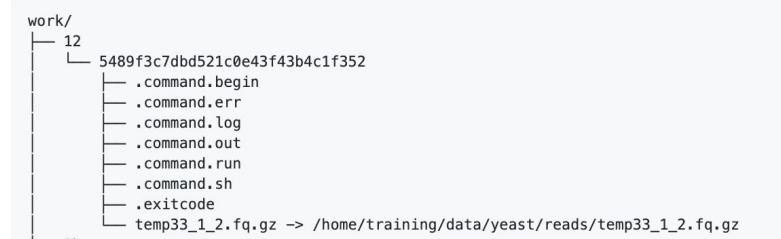
- `.command.sh`: The command script.
- `.command.run`: The command wrapped used to run the job.
- `.command.out`: The complete job standard output.
- `.command.err`: The complete job standard error.



What is in Task execution directory ?

The task execution directory contains:

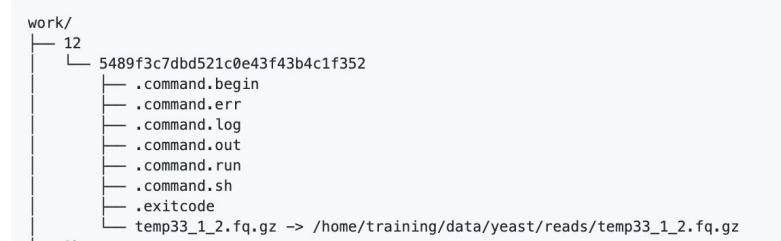
- `.command.sh`: The command script.
- `.command.run`: The command wrapped used to run the job.
- `.command.out`: The complete job standard output.
- `.command.err`: The complete job standard error.
- `.command.log`: The wrapper execution output.



What is in Task execution directory ?

The task execution directory contains:

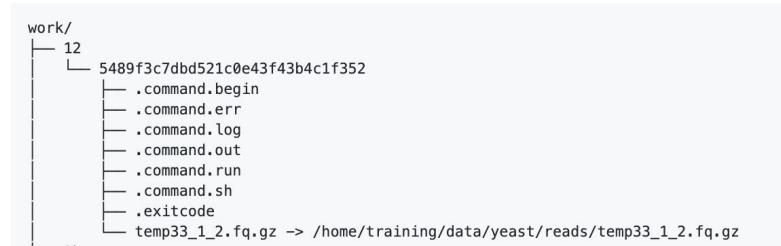
- `.command.sh`: The command script.
- `.command.run`: The command wrapped used to run the job.
- `.command.out`: The complete job standard output.
- `.command.err`: The complete job standard error.
- `.command.log`: The wrapper execution output.
- `.command.begin`: A file created as soon as the job is launched.



What is in Task execution directory ?

The task execution directory contains:

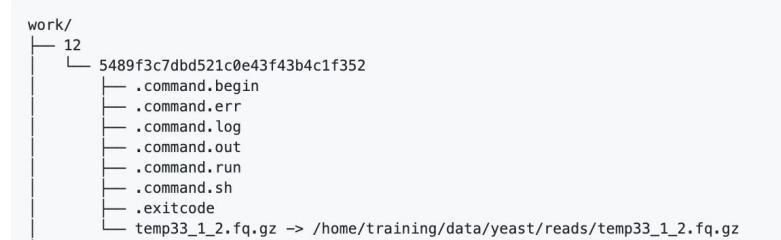
- `.command.sh`: The command script.
- `.command.run`: The command wrapped used to run the job.
- `.command.out`: The complete job standard output.
- `.command.err`: The complete job standard error.
- `.command.log`: The wrapper execution output.
- `.command.begin`: A file created as soon as the job is launched.
- `.exitcode`: A file containing the task exit code.



What is in Task execution directory ?

The task execution directory contains:

- `.command.sh`: The command script.
- `.command.run`: The command wrapped used to run the job.
- `.command.out`: The complete job standard output.
- `.command.err`: The complete job standard error.
- `.command.log`: The wrapper execution output.
- `.command.begin`: A file created as soon as the job is launched.
- `.exitcode`: A file containing the task exit code.
- **Task input files (symlinks)**
- **Task output files**





Modify Nextflow script and re-run.

Alter the timestamp on the file temp33_3_2.fq.gz using the UNIX `touch` command.

Bash

```
$ touch data/yeast/reads/temp33_3_2.fq.gz
```

Run command below.

Code

```
$ nextflow run wc.nf --input 'data/yeast/reads/temp33*' -resume
```

How many processes will be cached and how many will run ?

Another work directory **-w**

- By default nextflow uses the folder **work** in the directory where you call the script from.
- We can change this using the **-w** option and specifying another location

```
$ nextflow run wc.nf --input 'data/yeast/reads/temp33*' -w second work dir  
-resume
```

Cleaning the cache

- The work directory can become quite large
- To remove intermediate files we can clean the cache

```
$ nextflow clean [run name|session id] [options]
```

Cleaning the cache

- The work directory can become quite large
- To remove intermediate files we can clean the cache

```
$ nextflow clean [run name|session id] [options]
```

- Options
 - **-f, -force** force remove files
 - **-n, -dry-run** dry-run Print names of file to be removed without deleting them
 - **-k, keep-logs** Removes only temporary files but retains execution log entries and Metadata
 - **-after** Clean up runs executed before the specified one
 - **-before** Clean up runs executed before the specified one

Exercise



Remove a Nextflow run.

Remove the last nextflow run using the command `nextflow clean`. First use the option `-dry-run` to see which files would be deleted and then re-run removing the run and associated files.

Solution



Checkpointing and Caching: Key Points

- Nextflow automatically keeps track of all the processes executed in your pipeline via checkpointing.
- Nextflow caches intermediate data in task directories within the work directory.
- Nextflow caching and checkpointing allows re-entrancy into a workflow after a pipeline error or using new data, skipping steps that have been successfully executed.
- Re-entrancy is enabled using the `-resume` option.

Simple RNA-Seq pipeline

https://carpentries-incubator.github.io/workflows-nextflow/09-Simple_Rna-Seq_pipeline/index.html

Simple RNA-Seq pipeline

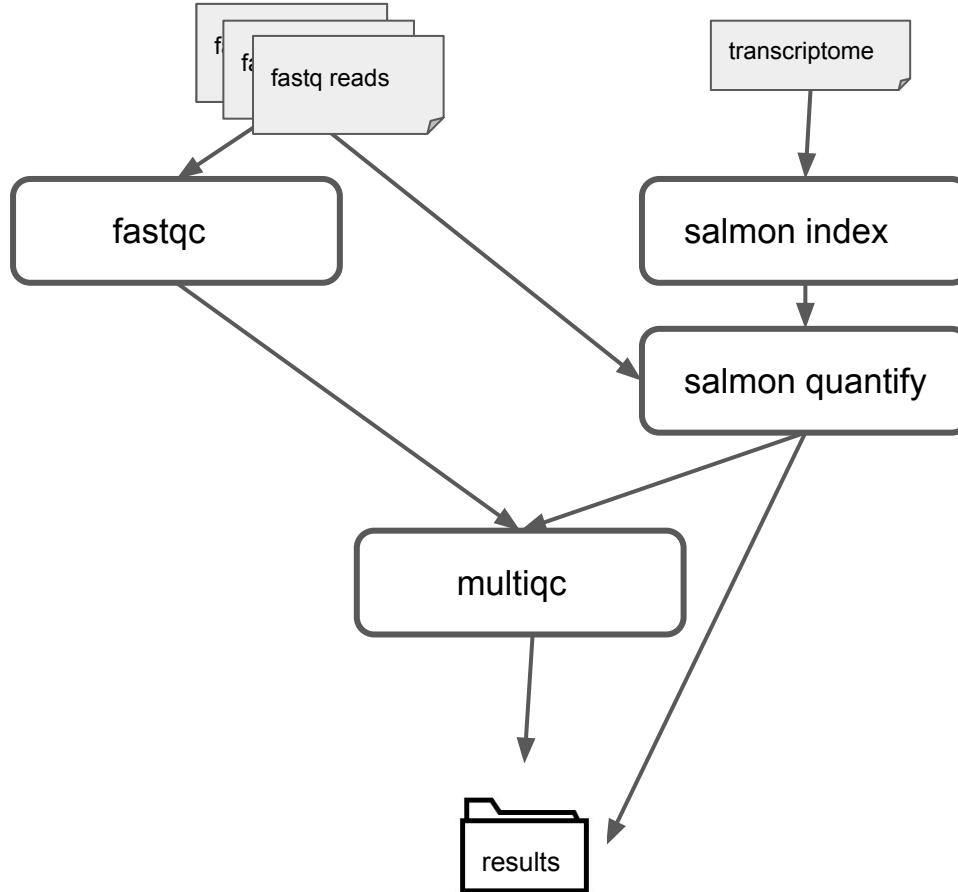
How can I create a RNA-Seq pipeline?

How do I print all the pipeline parameters by using a single command?

How can I use conda with my pipeline?

How do I know when my pipeline has finished?

How do I see runtime metrics and execution information?



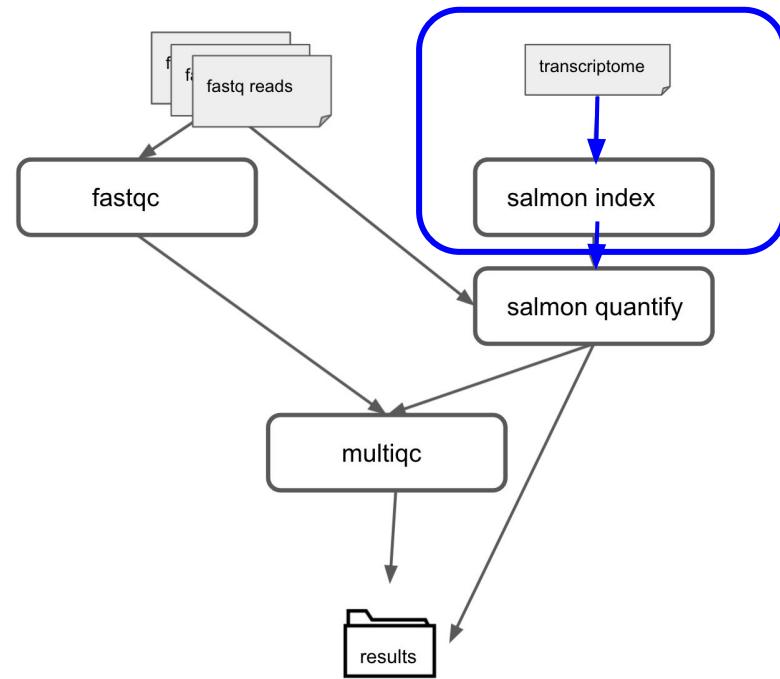
Steps: Index Transcriptome

We are finally ready to implement a simple RNA-Seq pipeline in Nextflow. This pipeline will have 4 processes that:

- indexes a transcriptome file.

Bash

```
$ salmon index --threads $task.cpus -t $transcriptome -i index
```



```
$ salmon index --threads $task.cpus -t $transcriptome -i index
```

Steps: FASTQC

We are finally ready to implement a simple RNA-Seq pipeline in Nextflow. This pipeline will have 4 processes that:

- indexes a transcriptome file.

Bash

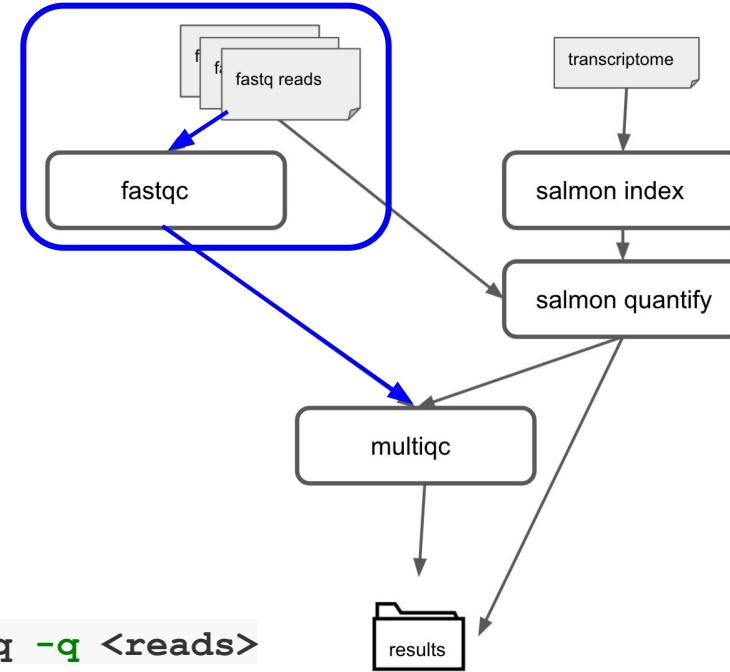
```
$ salmon index --threads $task.cpus -t $transcriptome -i index
```

- Performs quality controls

Bash

```
$ mkdir fastqc_<sample_id>_logs  
$ fastqc -o fastqc_<sample_id>_logs -f fastq -q <reads>
```

```
$ mkdir fastqc_<sample_id>_logs  
$ fastqc -o fastqc_<sample_id>_logs -f fastq -q <reads>
```



Steps: Salmon Quant

We are finally ready to implement a simple RNA-Seq pipeline in Nextflow. This pipeline will have 4 processes that:

- indexes a transcriptome file.

Bash

```
$ salmon index --threads $task.cpus -t $transcriptome -i index
```

- Performs quality controls

Bash

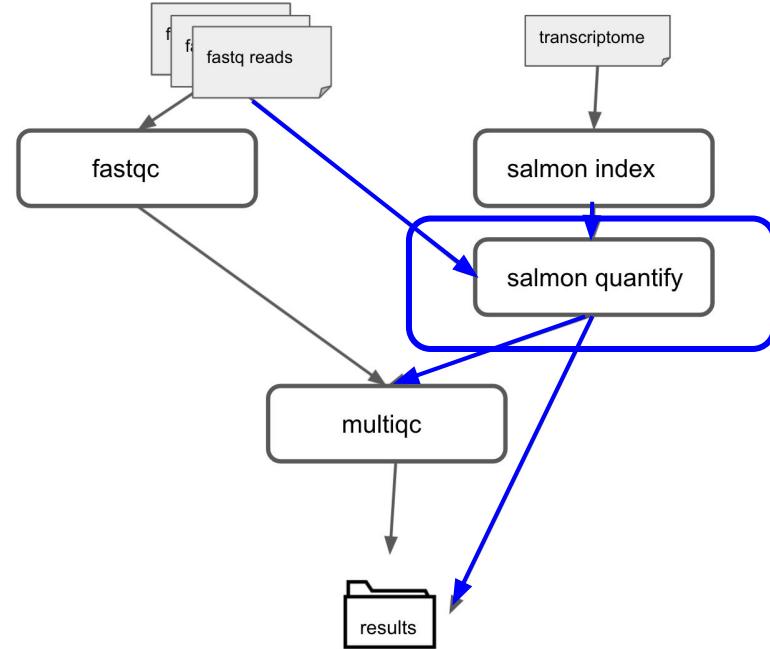
```
$ mkdir fastqc_<sample_id>_logs  
$ fastqc -o fastqc_<sample_id>_logs -f fastq -q <reads>
```

- Performs transcript level quantification.

Bash

```
$ salmon quant --threads <cpus> --libType=U -i <index> -1 <read1> -2 <read2> -o <pair_id>
```

```
$ salmon quant --threads <cpus> --libType=U -i <index> -1 <read1> -2 <read2> -o <pair_id>
```



Steps: MultiQC

We are finally ready to implement a simple RNA-Seq pipeline in Nextflow. This pipeline will have 4 processes that:

- indexes a transcriptome file.

Bash

```
$ salmon index --threads $task.cpus -t $transcriptome -i index
```

- Performs quality controls

Bash

```
$ mkdir fastqc_<sample_id>_logs  
$ fastqc -o fastqc_<sample_id>_logs -f fastq -q <reads>
```

- Performs transcript level quantification.

Bash

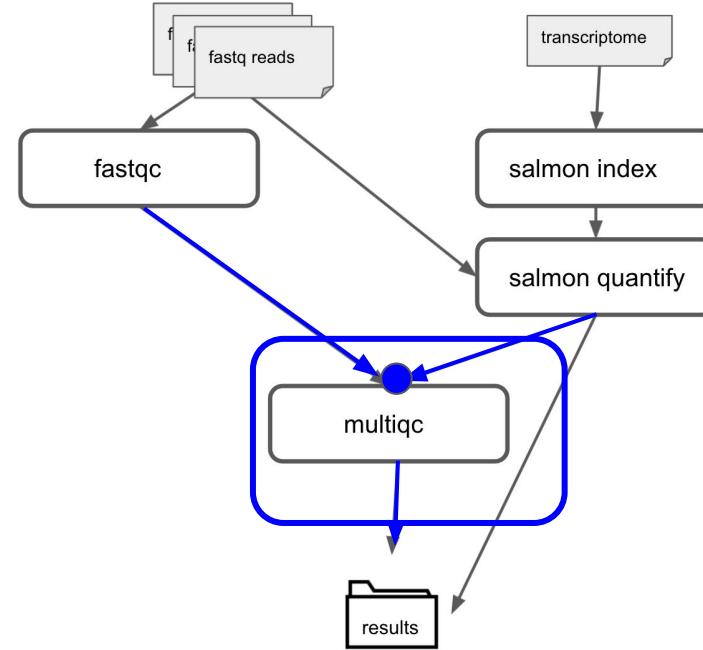
```
$ salmon quant --threads <cpus> --libType=U -i <index> -1 <read1> -2 <read2> -o <pair_id>
```

- Create a MultiQC report from the FastQC and salmon results.

Bash

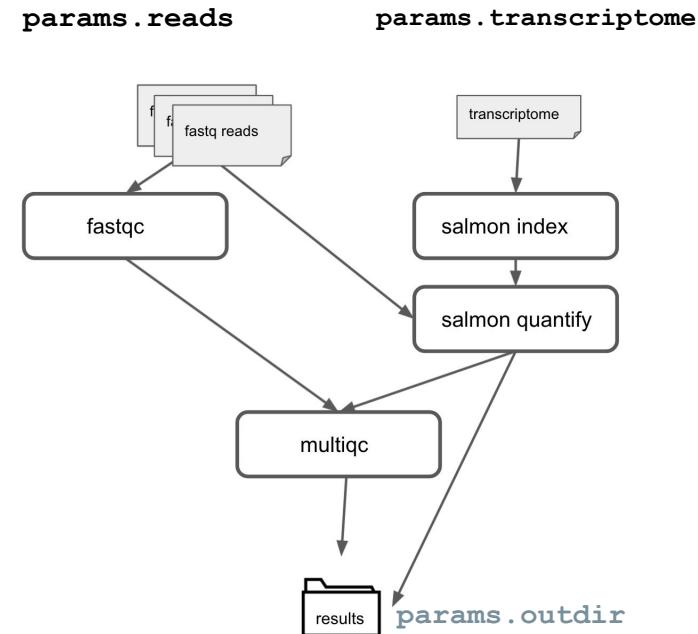
```
$ multiqc .
```

\$ multiqc .



Steps

1. Define the pipeline parameters
2. Create a transcriptome index file
3. Collect read files by pairs
4. Perform expression quantification
5. Quality control
6. MultiQC report
7. Handle completion event
8. Metrics and reports



script1.nf

script1.nf: Parameters

The first thing we want to do when writing a pipeline is define the pipeline parameters. The script script1.nf defines the pipeline input parameters.

```
//script1.nf
params.reads = "data/yeast/reads/*_{1,2}.fq.gz"
params.transcriptome = "data/yeast/transcriptome/*.fa.gz"

println "reads: $params.reads"

$ nextflow run script1.nf
```

script1.nf: Parameters

The first thing we want to do when writing a pipeline is define the pipeline parameters. The script script1.nf defines the pipeline input parameters.

```
//script1.nf
params.reads = "data/yeast/reads/*_{1,2}.fq.gz"
params.transcriptome = "data/yeast/transcriptome/*.fa.gz"

println "reads: $params.reads"
```

← Use \${variable_name} for String interpolation

```
$ nextflow run script1.nf
```

script1.nf: Parameters --params

We can specify a different input parameter using the `--<params>` option, for example :

```
//script1.nf
params.reads = "data/yeast/reads/*_{1,2}.fq.gz"
params.transcriptome = "data/yeast/transcriptome/*.fa.gz"

println "reads: $params.reads"

$ nextflow run script1.nf --reads "data/yeast/reads/ref1*_{1,2}.fq.gz"
```

Exercise: script1.nf

 Add parameter

Modify the `script1.nf` adding a fourth parameter named `outdir` and set it to a default path that will be used as the pipeline output directory.

 Solution ▾

script1.nf: log.info

It can be useful to print the pipeline parameters to the screen.

This can be done using the the `log.info` command and a multiline string statement.

```
log.info """\
    transcriptome: ${params.transcriptome}
"""
.stripIndent()
```

```
$ nextflow run script1.nf
```

script1.nf: log.info

It can be useful to print the pipeline parameters to the screen.

This can be done using the the `log.info` command and a multiline string statement.

The string method `.stripIndent()` command is used to remove the indentation on multi-line strings.
`log.info` also saves the output to the log execution file `.nextflow.log`.

```
log.info """\
    transcriptome: ${params.transcriptome}
"""
.stripIndent()
```

```
$ nextflow run script1.nf
```

Exercise

log.info

Modify the `script1.nf` to print all the pipeline parameters by using a single `log.info` command and a multiline string statement. See an example [here](#).

Bash

```
$ nextflow run script1.nf
```

Look at the output log `.nextflow.log`.

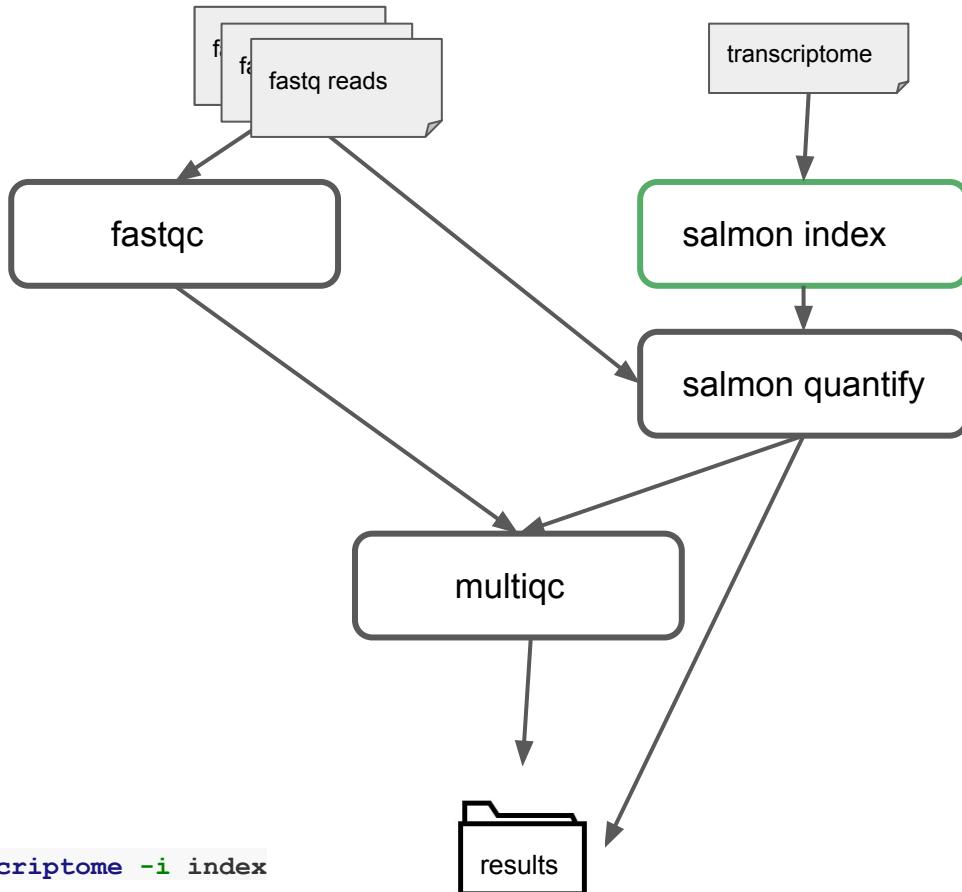
Hint: <https://github.com/nextflow-io/rnaseq-nf/blob/3b5b49f/main.nf#L41-L48>

Recap: Define the pipeline parameters

- How to define parameters in your pipeline script.
- How to pass parameters by using the command line.
- The use of `$var` and `${var}` variable placeholders.
- How to use multiline strings.
- How to use `log.info` to print information and save it in the log execution file.

Steps

1. Define the pipeline parameters
2. Create a transcriptome index file
3. Collect read files by pairs
4. Perform expression quantification
5. Quality control
6. MultiQC report
7. Handle completion event
8. Metrics and reports



```
$ salmon index --threads $task.cpus -t $transcriptome -i index
```

script2.nf: Create a transcriptome index file

```
//script2.nf
nextflow.enable.dsl=2

/*
 * pipeline input parameters
 */
params.reads = "data/yeast/reads/*_{1,2}.fq.gz"
params.transcriptome = "data/yeast/transcriptome/Saccharomyce
params.outdir = "results"

println """\
RNASEQ-NF PIPELINE
=====
transcriptome: ${params.transcriptome}
reads      : ${params.reads}
outdir     : ${params.outdir}
"""
.stripIndent()
```

```
/*
 * define the `INDEX` process that create a binary index
 * given the transcriptome file
 */
process INDEX {

    input:
    path transcriptome

    output:
    path 'index'

    script:
    """
    salmon index --threads $task.cpus -t $transcriptome -i index
    """

}

transcriptome_ch = channel.fromPath(params.transcriptome)

workflow {
    INDEX(transcriptome_ch)
}
```

```
$ nextflow run script2.nf
```

script2.nf: Create a transcriptome index file: Conda

The execution will fail because the program `salmon` is not available in your environment.

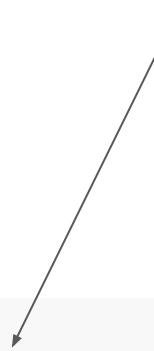
```
//nextflow.config
profiles {
    conda {
        process.conda = '/home/training/miniconda3/envs/nf'
    }
}
```

```
$ nextflow run script2.nf -profile conda
```

script2.nf: view operator

To view the output of the `INDEX` process use the `.view()` operator

```
workflow {
    INDEX(transcriptome_ch)
}
```



```
$ nextflow run script2.nf -profile conda
```

Exercises: script2.nf

✍ Enable conda by default

Enable the conda execution by removing the profile block in the `nextflow.config` file.

👁 Solution

Use the `.view()` operator to view the output of the `INDEX` process

```
workflow {  
    INDEX(transcriptome_ch)  
}
```

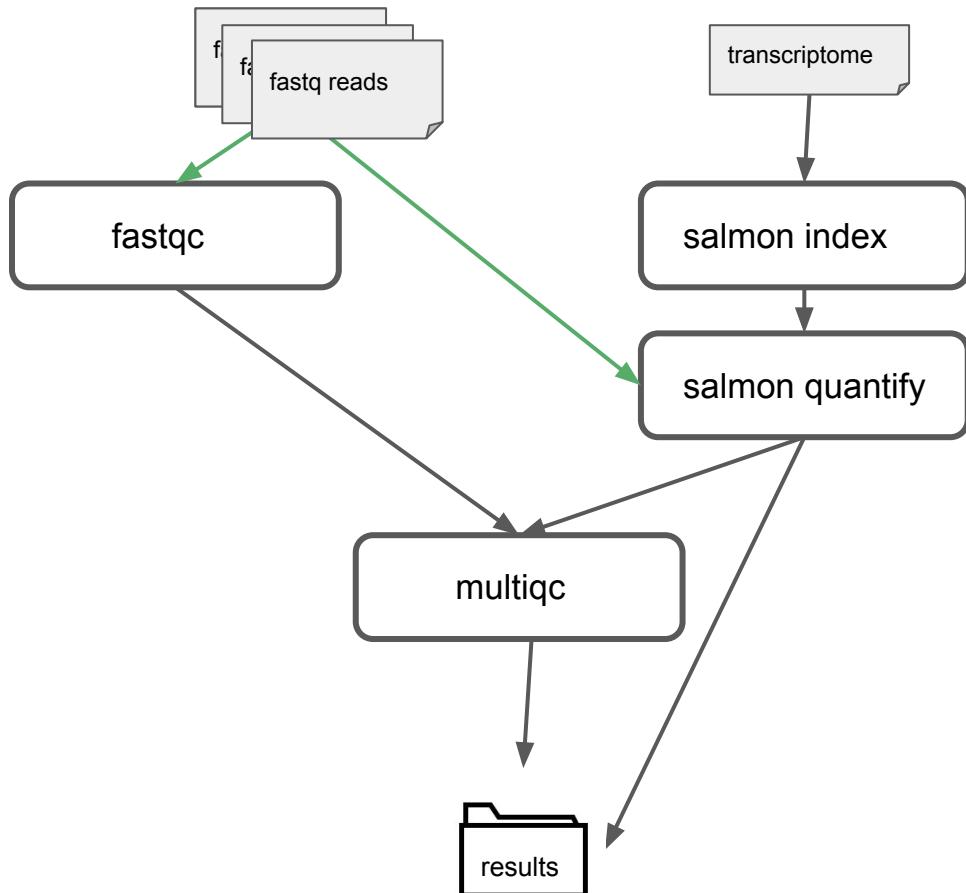


Recap: Create a transcriptome index file

- How to define a process executing a custom command
- How process inputs are declared
- How process outputs are declared
- How to use a nextflow configuration file to define and enable a `conda` environment.
- How to print the content of a channel `view()`

Steps

1. Define the pipeline parameters
2. Create a transcriptome index file
3. Collect read files by pairs
4. Perform expression quantification
5. Quality control
6. MultiQC report
7. Handle completion event
8. Metrics and reports



script3.nf: Collect read files by pairs

This step shows how to match **read** files into pairs, so they can be mapped by salmon.

The script `script3.nf` adds a line to create a channel, `read_pairs_ch`, containing fastq read pair files using the `fromFilePairs` channel factory.

```
params.reads = "data/yeast/reads/ref1_{1,2}.fq.gz"
```

```
read_pairs_ch = Channel.fromFilePairs( params.reads )
```

script3.nf: Collect read files by pairs

This step shows how to match **read** files into pairs, so they can be mapped by salmon.

The script `script3.nf` adds a line to create a channel, `read_pairs_ch`, containing fastq read pair files using the `fromFilePairs` channel factory.

```
read_pairs_ch = Channel.fromFilePairs( params.reads )
```

```
read_pairs_ch.view()
```

```
$ nextflow run script3.nf
```

script3.nf: Tuples

An output similar to the one shown below that shows how the `read_pairs_ch` channel emits a tuple.

The tuple is composed of two elements, where the first is the pattern matched by the glob pattern
`data/yeast/reads/ref1_{1,2}.fq.g`, defined by the variable `params.reads`, and the second is a list representing the actual files.

```
[ref1, [data/yeast/reads/ref1_1.fq.gz,data/yeast/reads/ref1_2.fq.gz]]
```

script3.nf: --params

To read in other read pairs we can specify a different glob pattern in the params.reads variable by using --reads options on the command line

```
params.reads = "data/yeast/reads/ref1_{1,2}.fq.gz"
```

```
$ nextflow run script3.nf --reads 'data/yeast/reads/ref*_1,2}.fq.gz'
```

script3.nf: Collect read files by pairs options

We can also add a argument, `checkIfExists: true` , to the `fromFilePairs` channel factory to return an message if the file doesn't exist.

```
read_pairs_ch = Channel.fromFilePairs( params.reads, checkIfExists: true)

$ nextflow run script3.nf --reads 'data/yeast/reads/*_1,2}.fq.gz'
```

script3.nf: Collect read files by pairs options

We can also add a argument, `checkIfExists: true` , to the `fromFilePairs` channel factory to return an message if the file doesn't exist.

```
[..truncated..]  
No such file: data/yeast/reads/*_1,2}.fq.gz
```

```
read_pairs_ch = Channel.fromFilePairs( params.reads, checkIfExists: true)
```

```
$ nextflow run script3.nf --reads 'data/yeast/reads/*_1,2}.fq.gz'
```

Exercises: script3.nf

Read in all read pairs

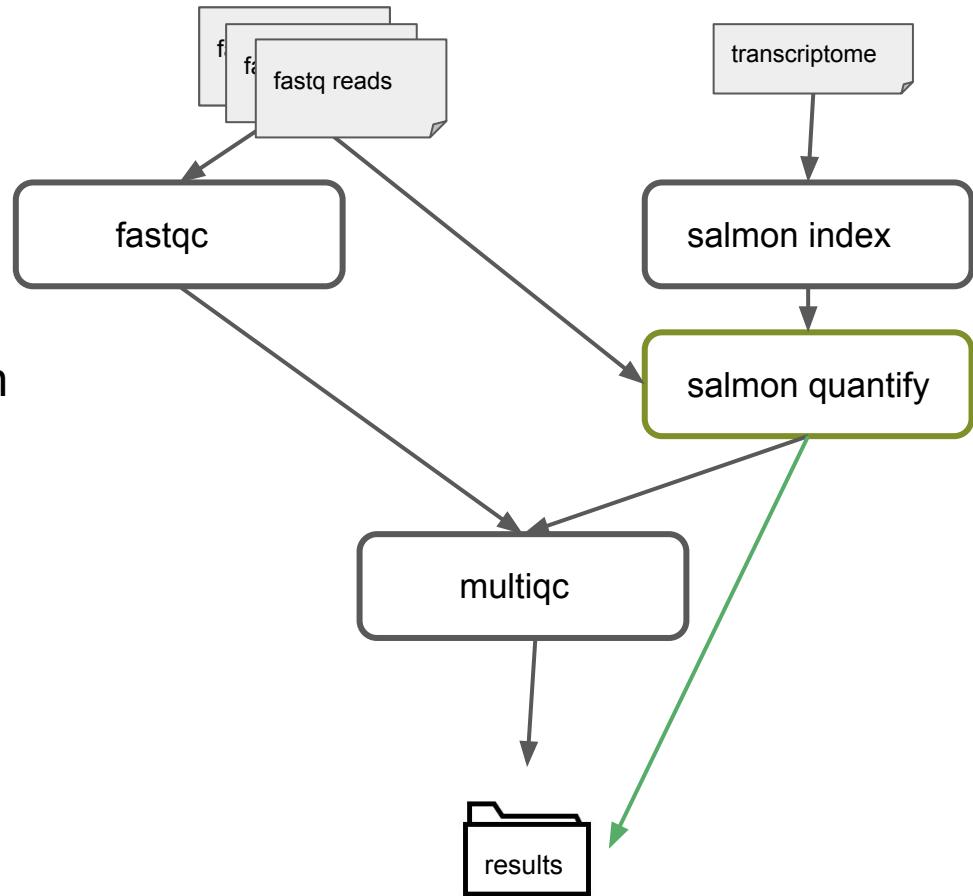
1. Add the `checkIfExists: true` argument to the `fromFilePairs` channel factory in `script3.nf`.
2. Using the command line parameter `--reads`, add a glob pattern to read in *all* the read pairs files from the `data/yeast/reads` directory.

Recap: Collect read files by pairs

- How to use `fromFilePairs` to handle read pair files
- How to use the `checkIfExists` option to check input file existence

Steps

1. Define the pipeline parameters
2. Create a transcriptome index file
3. Collect read files by pairs
4. Perform expression quantification
5. Quality control
6. MultiQC report
7. Handle completion event
8. Metrics and reports



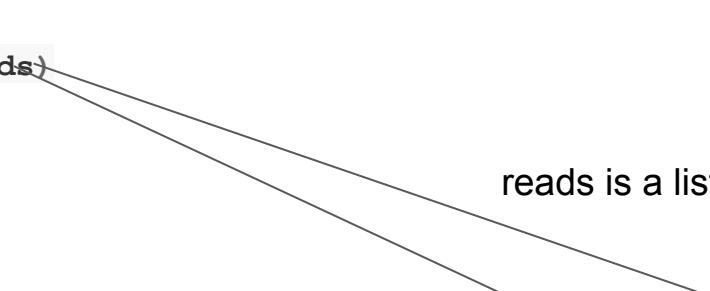
script4.nf

script4.nf: Perform expression quantification

The script `script4.nf`:

1. Adds the quantification process, `QUANT`.

```
process QUANT {  
  
    input:  
    path index  
    tuple val(pair_id),path(reads)  
  
    output:  
    path(pair_id)  
  
    script:  
    """  
    salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id  
    """  
}
```



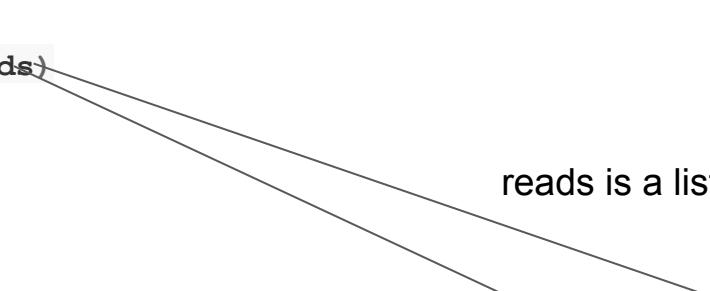
reads is a list

script4.nf: Perform expression quantification

The script `script4.nf`:

1. Adds the quantification process, `QUANT`.

```
process QUANT {  
  
    input:  
    each index  
        tuple val(pair_id),path(reads)  
  
    output:  
        path(pair_id)  
  
    script:  
    """  
        salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id  
    """  
}
```



reads is a list

script4.nf: Perform expression quantification

The script `script4.nf`:

1. Adds the quantification process, `QUANT`.
2. Calls the `QUANT` process in the workflow block.

```
process QUANT {  
    inp  
    pat  
    tup  
    workflow {  
        out  read_pairs_ch = Channel.fromFilePairs( params.reads, checkIfExists:true )  
        pat   transcriptome_ch = Channel.fromPath( params.transcriptome, checkIfExists:true )  
    }  
    sci  
    """  
    sal   index_ch=INDEX(transcriptome_ch)  
    """  
    quant_ch=QUANT(index_ch,read_pairs_ch)  
}  
}  
}
```

Connection two process via output and input channel

script4.nf: Perform expression quantification

```
$ nextflow run script4.nf
```

You will see the execution of the index and quantification process.

How many times does it run?

script4.nf: Perform expression quantification

```
$ nextflow run script4.nf
```

You will see the execution of the index and quantification process.

How many times does it run?

The QUANT process runs 1 times.

```
params.reads = "data/yeast/reads/ref1_{1,2}.fq.gz"
```

script4.nf: Perform expression quantification: resume

```
$ nextflow run script4.nf -resume
```

The `-resume` option cause the execution of any step that has been already processed to be skipped.

How many times does the QUANT process run ?

script4.nf: Perform expression quantification: resume

```
$ nextflow run script4.nf -resume
```

The `-resume` option cause the execution of any step that has been already processed to be skipped.

How many times does the QUANT process run ?

The QUANT process is skipped data is retrieved from cache

script4.nf: Perform expression quantification: resume

```
$ nextflow run script4.nf --resume --reads 'data/yeast/reads/ref*_1,2.fq.gz'
```

Adding more read files

How many time does the QUANT process run ?

You will notice that the INDEX step and one of the QUANT steps has been cached, and the quantification process is executed more than one time.

script4.nf: Perform expression quantification: tagging

```
$ nextflow run script4.nf --resume --reads 'data/yeast/reads/ref*_1,2.fq.gz'
```

When your input channel contains multiple data items Nextflow, where possible, parallelises the execution of your pipeline.

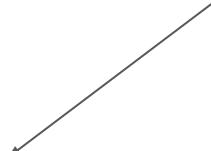
In these situations it is useful to add a `tag` directive to add some descriptive text to instance of the process being run.

```
process QUANT {
    tag "some useful info"
    input:
    each index
    tuple val(pair_id), path(reads)

    output:
    path(pair_id)

    script:
    """
    salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id
    """
}

}
```



Exercise: script4.nf

Add a tag directive

Add a `tag` directive to the `QUANT` process of `script4.nf` to provide a more readable execution log.

Solution

script4.nf: Perform expression quantification: publishDir

Data produced by the workflow during a process will be saved in the working directory, by default a directory named `work`.

The working directory should be considered a temporary storage space and any data you wish to save at the end of the workflow should be specified in the process output with the final storage location defined in the `publishDir` directive.

```
process QUANT {
    publishDir "some_location", mode:"copy"
    input:
    each index
    tuple val(pair_id),path(reads)

    output:
    path(pair_id)

    script:
    """
    salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id
    """
}


```

script4.nf: Perform expression quantification: publishDir

Data produced by the workflow during a process will be saved in the working directory, by default a directory named `work`.

The working directory should be considered a temporary storage space and any data you wish to save at the end of the workflow should be specified in the process output with the final storage location defined in the `publishDir` directive.

```
process QUANT {
    publishDir "some_location", mode:"copy"
    input:
    each index
        tuple val(pair_id), path(reads)

    output:
    path(pair_id)

    script:
    """
    salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]} -o $pair_id
    """
}
```

Note: by default the `publishDir` directive creates a symbolic link to the files in the working this behaviour can be changed using the `mode` parameter.

Exercise: script4.nf

Add a publishDir directive

Add a `publishDir` directive to the quantification process of `script4.nf` to store the process results into folder specified by the `params.outdir` Nextflow variable. Include the `publishDir mode` option to copy the output.

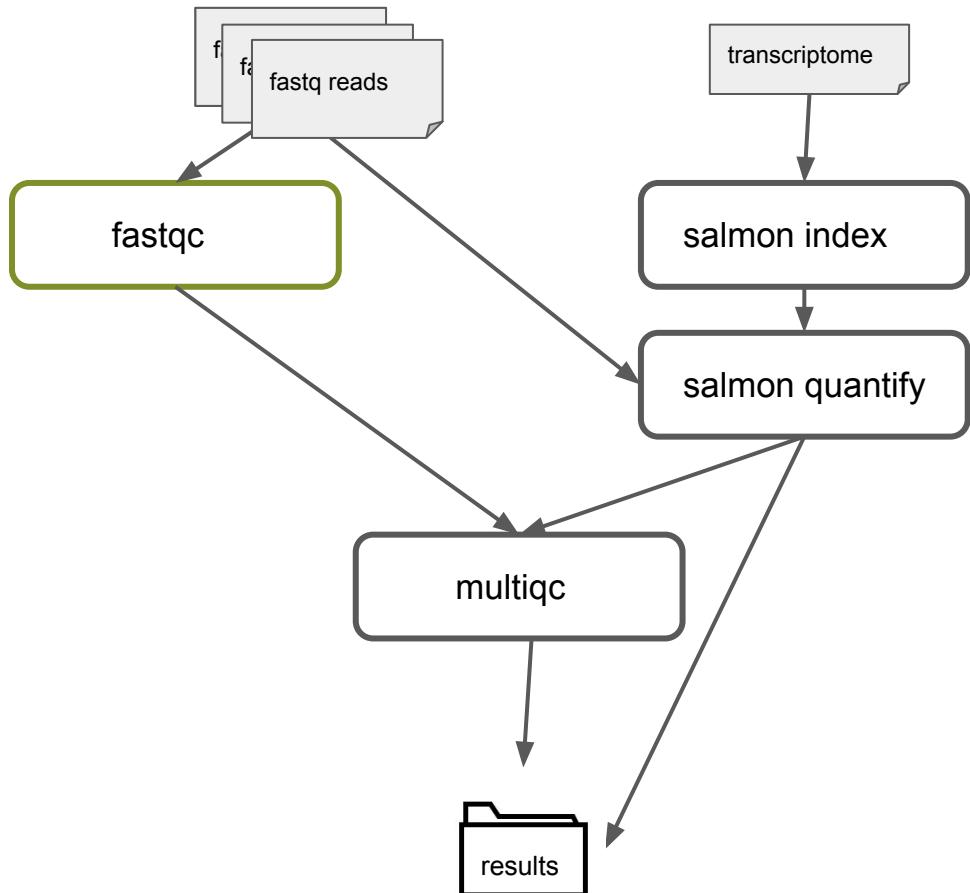
Solution

Recap: Perform expression quantification

- How to connect two processes by using the channel declarations.
- How to resume the script execution skipping already computed steps.
- How to use the `tag` directive to provide a more readable execution output.
- How to use the `publishDir` to store a process results in a path of your choice.

Steps

1. Define the pipeline parameters
2. Create a transcriptome index file
3. Collect read files by pairs
4. Perform expression quantification
5. Quality control
6. MultiQC report
7. Handle completion event
8. Metrics and reports



script5.nf: Quality Control

This step implements a quality control step for your input reads.

```
process FASTQC {
    tag "FASTQC on ${sample_id}"
    cpus 1

    input:
        tuple val(sample_id), path(reads)

    output:
        path("fastqc_${sample_id}_logs")

    script:
        """
        mkdir fastqc_${sample_id}_logs
        fastqc -o fastqc_${sample_id}_logs -f fastq -q ${reads} -t ${task.cpus}
        """
}
```

script5.nf: Quality Control

This step implements a quality control step for your input reads.

The input to the `FASTQC` process is the same `read_pairs_ch` that is provided as input to the quantification process `QUANT`.

```
process FASTQC {
    tag "FASTQC on ${sample_id}"
    cpus 1

    input:
    tuple val(sample_id), path(reads)

    output:
    path("fastqc_${sample_id}_logs")

    script:
    """
    mkdir fastqc_${sample_id}_logs
    fastqc -o fastqc_${sample_id}_logs -f fastq -q ${reads} -t ${task.cpus}
    """
}

$ nextflow run script5.nf -resume
```

script5.nf: Quality Control

The `FASTQC` process will not run as the process has not been declared in the workflow scope..

```
process FASTQC {
    #> FASTQC -> QC_report
}

workflow {
    read_pairs_ch = Channel.fromFilePairs( params.reads, checkIfExists:true )
    transcriptome_ch = Channel.fromPath( params.transcriptome, checkIfExists:true )

    index_ch=INDEX(transcriptome_ch)
    quant_ch=QUANT(index_ch,read_pairs_ch)
}

}
```

Exercise: script5.nf

Add FASTQC process

Add the `FASTQC` process to the `workflow scope` of `script5.nf` adding the `read_pairs_ch` channel as an input. Run the nextflow script using the `-resume` option.

Bash

```
$ nextflow run script5.nf -resume
```

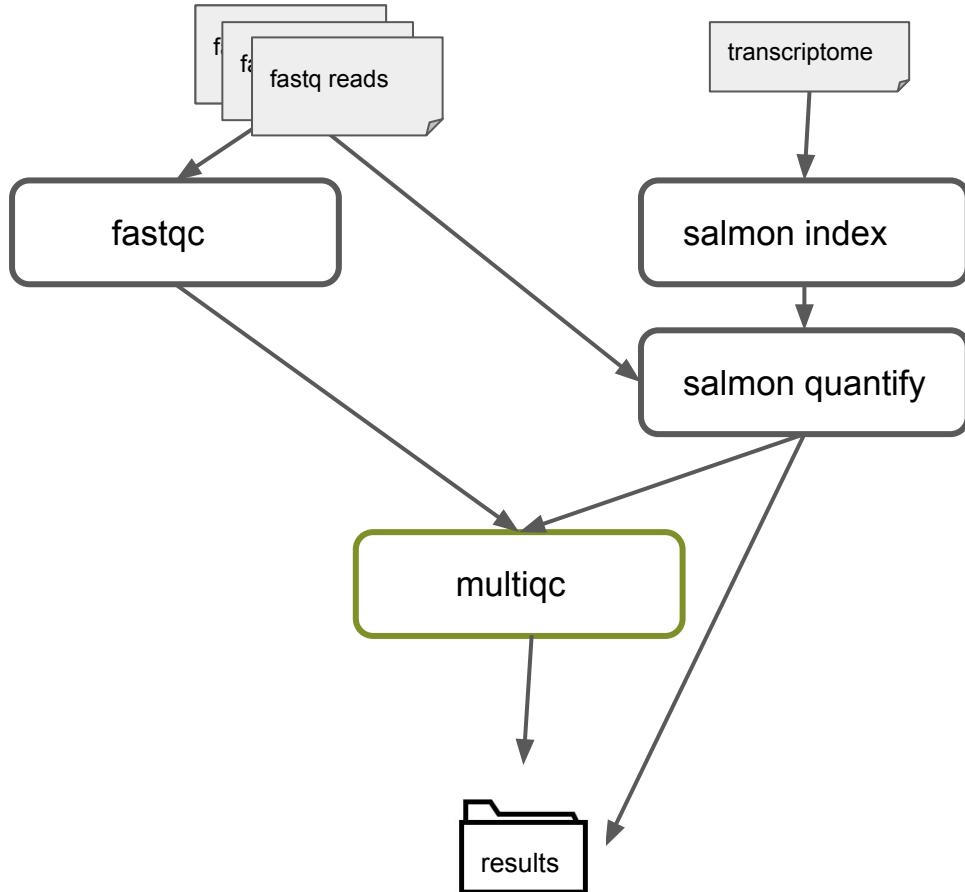
Solution

Recap: Quality Control

- How to add a process to the workflow scope.
- How to add a channel as input to a process.

Steps

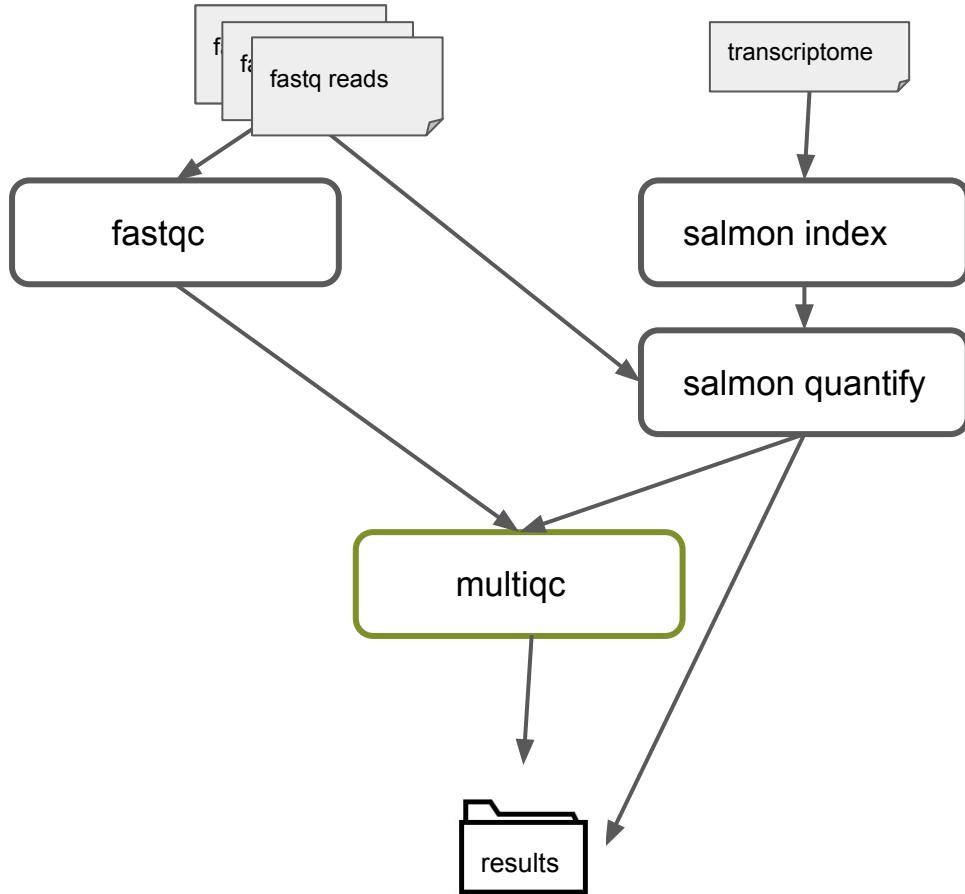
1. Define the pipeline parameters
2. Create a transcriptome index file
3. Collect read files by pairs
4. Perform expression quantification
5. Quality control
6. MultiQC report
7. Handle completion event
8. Metrics and reports



script6.nf: multiqc

- Aggregate results from bioinformatics analyses across many samples into a single report
- Runs **once** on all output of FASTQC and QUANT processes

```
$ multiqc .
```



Recap: collect and mix operators

```
ch1 = channel.of('A','B','C')
```

'A'	'B'	'C'
-----	-----	-----

```
ch2 = channel.of(1,2)
```

1	2
---	---

Recap: collect and mix operators

```
ch1 = channel.of('A','B','C')
```

'A'	'B'	'C'
-----	-----	-----

```
ch2 = channel.of(1,2)
```

1	2
---	---

What we want

['A' , 'B' , 'C' , 1 , 2]

mix() combining channels

ch1

'A'	'B'	'C'
-----	-----	-----

ch2

1	2
---	---

ch1.**mix(ch2)**

'A'	'B'	'C'	1	2
-----	-----	-----	---	---

`collect()` , collecting items into a single List

ch1

'A'	'B'	'C'
-----	-----	-----

`ch1.collect()`

['A', 'B', 'C']

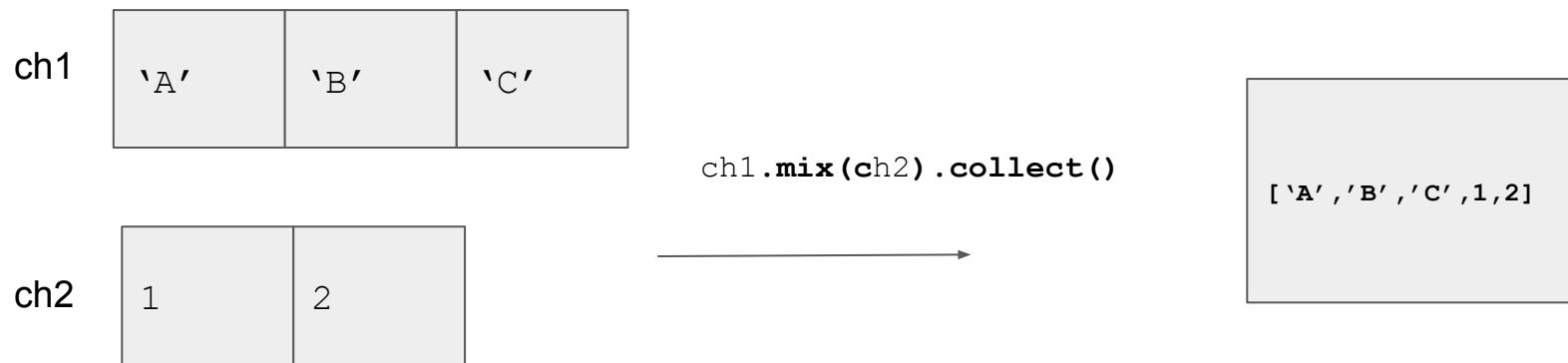
ch2

1	2
---	---

`ch2.collect()`

[1, 2]

mix() and collect()



Exercise

Combing operators

Which is the correct way to combined `mix` and `collect` operators so that you have a single channel with one List item?

1. `quant_ch.mix(fastqc_ch).collect()`
2. `quant_ch.collect(fastqc_ch).mix()`
3. `fastqc_ch.mix(quant_ch).collect()`
4. `fastqc_ch.collect(quant_ch).mix()`

Solution ▾

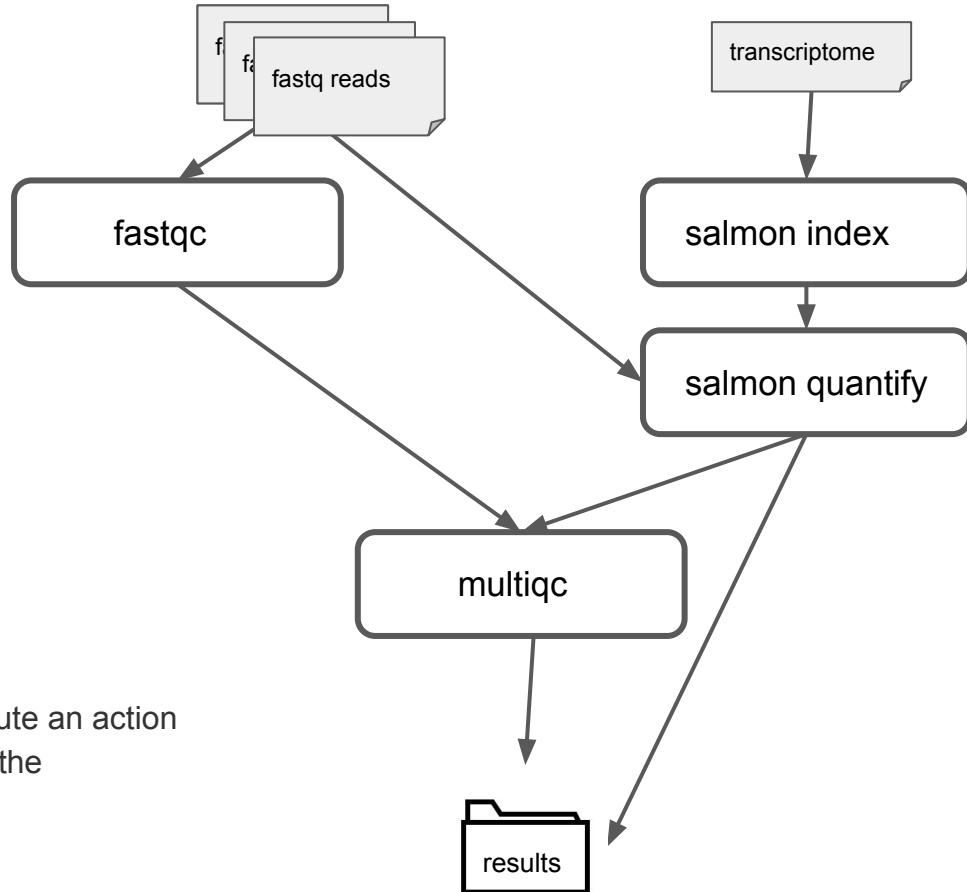
Recap: MultiQC

- How to collect many outputs to a single input with the `collect` operator
- How to mix two channels in a single channel using the `mix` operator.
- How to chain two or more operators together using the `.` operator

Steps

1. Define the pipeline parameters
2. Create a transcriptome index file
3. Collect read files by pairs
4. Perform expression quantification
5. Quality control
6. MultiQC report
7. Handle completion event
8. Metrics and reports

This step shows how to execute an action when the pipeline completes the execution.



script7.nf

Print a confirmation message when the script completes.

script7.nf: Handle completion event

The script `script7..nf` uses the `workflow.onComplete` event handler to print a confirmation message when the script completes.

```
workflow.onComplete { log.info("\nDone! Open the following report in your browser  
--> $params.outdir/multiqc/multiqc_report.html\n") }
```

script7.nf: Workflow introspection

We can use the `workflow` object `success` property to reports if the execution completed successfully.

```
workflow.success
```

script7.nf: ternary operator

Then we can use a groovy ternary expression equivalent to an if/else branch assigning some value to a variable.

```
workflow.success
```



```
If expression is true? "set value to a" : "else set value to b"
```

script7.nf: Handle completion event

Now message prints success or failure message depending in workflow.success parameter

```
workflow.onComplete {
    log.info ( workflow.success ? "\nDone! Open the following report in your
browser --> $params.outdir/multiqc/multiqc_report.html\n" : "Oops .. something
went wrong" )
}
```

```
$ nextflow run script7.nf -resume --reads 'data/yeast/reads/*_{1,2}.fq.gz'
```

Recap: Handle completion event

We can use the `workflow.onComplete` event handler to print a confirmation message when the script completes

Steps

1. Define the pipeline parameters
2. Create a transcriptome index file
3. Collect read files by pairs
4. Perform expression quantification
5. Quality control
6. MultiQC report
7. Handle completion event
8. Metrics and reports

Metrics and Reports

Nextflow is able to produce multiple reports and charts providing several runtime metrics and execution information.

- The `-with-report` option enables the creation of the workflow execution report.

Metrics and Reports

Nextflow is able to produce multiple reports and charts providing several runtime metrics and execution information.

- The `-with-report` option enables the creation of the workflow execution report.
- The `-with-trace` option enables the creation of a tab separated file containing runtime information for each executed task, including: submission time, start time, completion time, cpu and memory used..

Metrics and Reports

Nextflow is able to produce multiple reports and charts providing several runtime metrics and execution information.

- The `-with-report` option enables the creation of the workflow execution report.
- The `-with-trace` option enables the creation of a tab separated file containing runtime information for each executed task, including: submission time, start time, completion time, cpu and memory used..
- The `-with-timeline` option enables the creation of the workflow timeline report showing how processes were executed along time. This may be useful to identify most time consuming tasks and bottlenecks. See an example at this [link](#).

Metrics and Reports

Nextflow is able to produce multiple reports and charts providing several runtime metrics and execution information.

- The `-with-report` option enables the creation of the workflow execution report.
- The `-with-trace` option enables the creation of a tab separated file containing runtime information for each executed task, including: submission time, start time, completion time, cpu and memory used..
- The `-with-timeline` option enables the creation of the workflow timeline report showing how processes were executed along time. This may be useful to identify most time consuming tasks and bottlenecks. See an example at this [link](#).
- The `-with-dag` option enables rendering of the workflow execution direct acyclic graph representation. **Note:** this feature requires the installation of [Graphviz](#), an open source graph visualization software, in your system.

Exercises

Metrics and reports

Run the script7.nf with the reporting options as shown below:

Bash

```
$ nextflow run script7.nf -resume -with-report -with-trace -with-timeline -with-dag dag.png
```

1. Open the file `report.html` with a browser to see the report created with the above command.
2. Check the content of the file `trace.txt` or view `timeline.html` to find the longest running process.
3. View the `dag.png`

Solution ▾

Recap: Metrics and reports

Nextflow is able to produce multiple reports and charts providing several runtime metrics and execution information

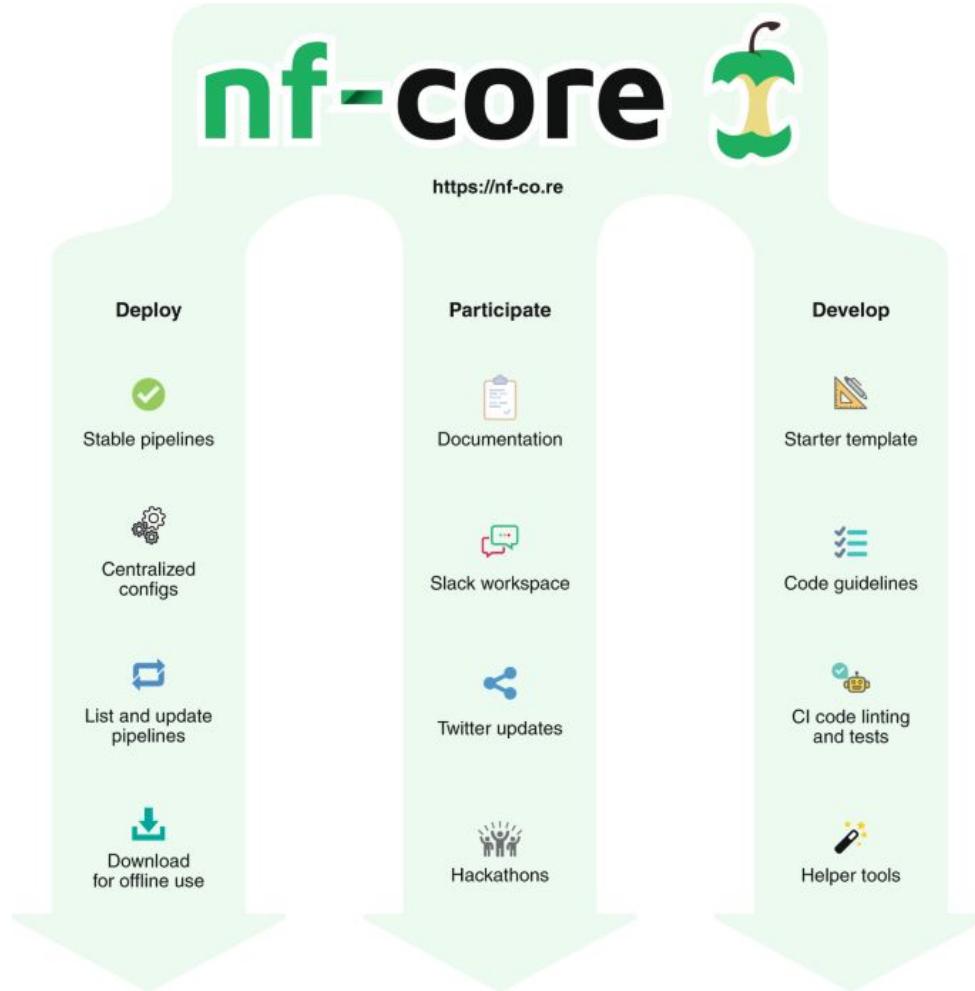
- `-with-report`
- `-with-trace`
- `-with-timeline`
- `-with-dag`

nf-core

nf-core

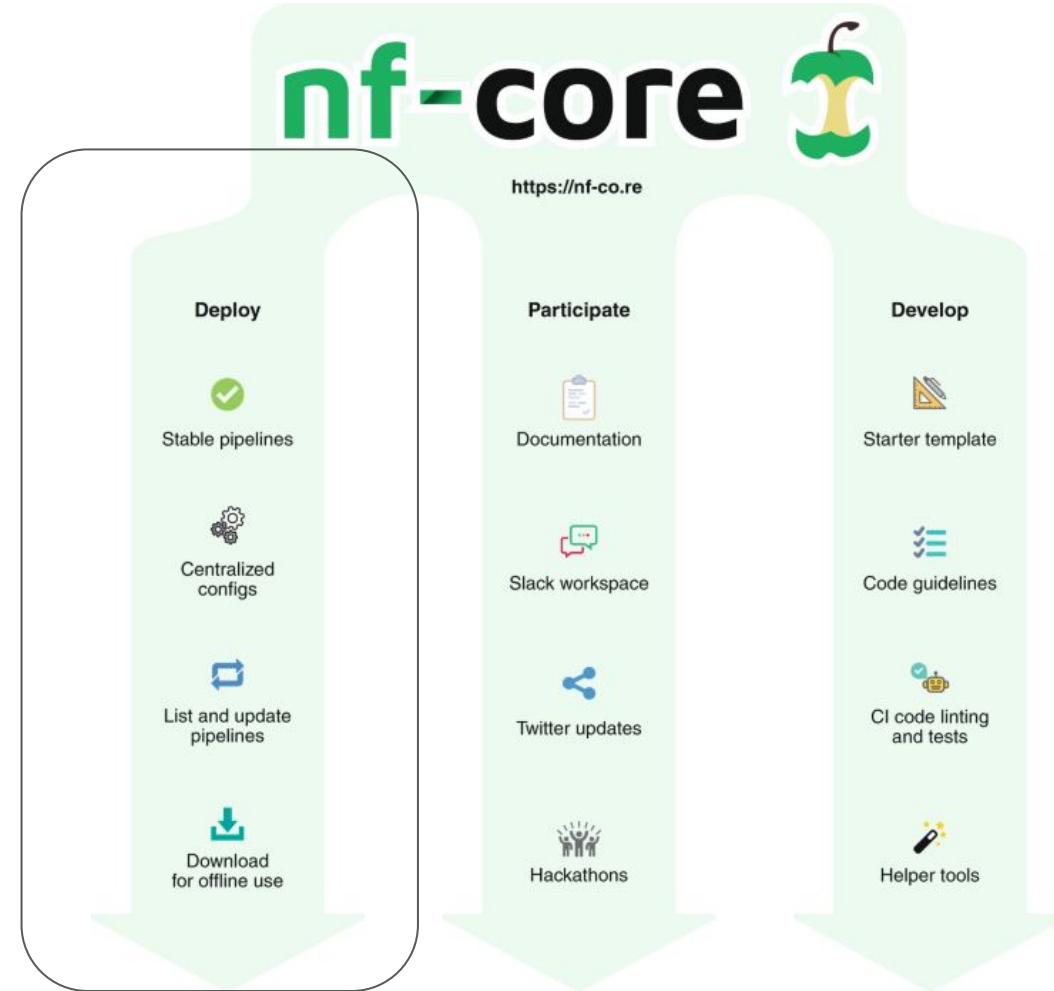
nf-core is a community-led project to develop a set of best-practice pipelines built using Nextflow workflow system.

Pipelines are governed by a set of guidelines, enforced by community code reviews and automatic code testing.



nf-core

In this episode we will cover deploying nf-core pipelines



nf-core: pipelines



Available Pipelines

Can you think of another pipeline that would fit in well? Let us know!

Search keywords Filter Released 28 Under development 12 Archived 4 Sort: Last Release Alphabetical Stars Display

nf-core/eager ✓ 46 adna ancient-dna-analysis ancientna genome metagenomics pathogen-genomics population-genetics A fully reproducible and state-of-the-art ancient DNA analysis pipeline Version 2.3.2 Published 1 week ago	nf-core/diaproteomics ✓ 5 data-independent-proteomics dia-proteomics openms proteomics Automated quantitative analysis of DNA proteomics mass spectrometry measurements. Version 1.2.2 Published 4 weeks ago
nf-core/dualrnaseq ✓ 3 dualrna-seq host-pathogen quantification remapping rna-seq Analysis of Dual RNA-seq data - an experimental method for interrogating host-pathogen interactions through simultaneous RNA-seq. Version 1.0.0 Published 1 month ago	nf-core/mag ✓ 42 annotation assembly binning long-read-sequencing metagenomes metagenomics nanopore nanopore-sequencing Assembly and binning of metagenomes Version 1.2.0 Published 1 month ago
nf-core/ampliseq ✓ 55 16s amplicon-sequencing metagenomics qime rna 16S RNA amplicon sequencing analysis workflow using QIIME2 Version 1.2.0 Published 2 months ago	nf-core/sarek ✓ 95 annotation cancer gatk4 genomics germline pre-processing somatic variant-calling Analysis pipeline to detect germline or somatic variants (pre-processing, variant calling and annotation) from WGS / targeted sequencing Version 2.7 Published 2 months ago
nf-core/cageseq ✓ 3 cage cagedata cageseq-data gene-expression rna CAGE-sequencing analysis pipeline with trimming, alignment and counting of CAGE tags. Version 1.0.2 Published 2 months ago	nf-core/rnaseq ✓ 308 rna rna-seq RNA sequencing analysis pipeline using STAR, RSEM, HISAT2 or Salmon with gene/exon counts and extensive quality control. Version 3.0 Published 3 months ago
nf-core/nanoseq ✓ 25 alignment demultiplexing nanopore qc Nanopore demultiplexing, QC and alignment pipeline Version 1.1.0 Published 5 months ago	nf-core/bacass ✓ 19 assembly bacterial-genomes denovo denovo-assembly genome-assembly hybrid-assembly nanopore nanopore-sequencing Simple bacterial assembly and annotation pipeline Version 1.1.1 Published 5 months ago
nf-core/epitopeprediction ✓ 10 epitope epitope-prediction mhc-binding-prediction A bioinformatics best-practice analysis pipeline for epitope prediction and annotation Version 1.0.0 Published 5 months ago	nf-core/proteomicslfq ✓ 15 label-free-quantification openms proteomics Proteomics label-free quantification (LFQ) analysis pipeline Version 1.0.0 Published 5 months ago

<https://nf-co.re/pipelines>

nf-core tools

nf-core provides a suite of helper tools aim to help people run and develop pipelines.

The nf-core tools package is written in Python and can run from the command line or imported and used within other packages.

```
$ nf-core
```

nf-core tools sub-commands

You can use the `--help` option to see the range of nf-core tools sub-commands.

We will be covering the

list

launch

download

sub-commands which aid in the finding and deployment of the nf-core pipelines.

```
$ nf-core --help
```

nfcore: Filtering available nf-core pipelines

If you supply additional keywords after the list sub-command, the listed pipeline will be filtered.

This searches more than just the displayed output, including keywords and description text.

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

nfcore: Sorting available nf-core pipelines

You can sort the results by adding the option `--sort` followed by a keyword.

- `--sort release`
- `--sort pulled`
- `--sort name`
- `--sort stars`

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

Exercise

Exercise: listing nf-core pipelines

1. Use the `--help` flag to print the list command usage.
2. Sort all pipelines by popularity (stars) and find out which is the most popular?.
3. Filter pipelines for those that work with RNA and find out how many there are?

nfcore: Running nf-core pipelines: software

nf-core pipeline software dependencies are specified using either Docker, Singularity or Conda.

It is Nextflow that handles the downloading of containers and creation of conda environments.

In theory it is possible to run the pipelines with software installed by other methods (e.g. environment modules, or manual installation), but this is not recommended.

nfcore: Fetching pipeline code

Unless you are actively developing pipeline code, you should use Nextflow's [built-in functionality](#) to fetch nf-core pipelines.

To pull the latest version of a remote workflow from the nf-core github site

```
$ nextflow pull nf-core/<PIPELINE>
```

nfcore: Fetching pipeline code

Nextflow will also automatically fetch the pipeline code when you run

```
$ nextflow run nf-core/<PIPELINE>
```

nfcore: Fetching pipeline code: revision

For the best reproducibility, it is good to explicitly reference the pipeline version number that you wish to use with the `-revision/-r` flag.

In the example below we are pulling the rnaseq pipeline version 3.0

```
$ nextflow pull nf-core/rnaseq -revision 3.0
```

Exercise

Exercise: Fetch the latest RNA-Seq pipeline

1. Use the `nextflow pull` command to download the latest `nf-core/rnaseq` pipeline
2. Use the `nf-core list` command to see if you have the latest version of the pipeline

Solution

nfcore: Usage instructions and documentation

You can find general documentation and instructions for Nextflow and nf-core on the [nf-core website](#).

Pipeline-specific documentation is bundled with each pipeline in the /docs folder. This can be read either locally, on GitHub, or on the nf-core website.

Each pipeline has its own webpage at <https://nf-co.re/> e.g.

<https://nf-co.re/rnaseq/usage>

nfcore: Usage instructions and documentation

In addition to this documentation, each pipeline comes with basic command line reference. This can be seen by running the pipeline with the parameter `--help` , for example:

```
$ nextflow run -r 3.4 nf-core/rnaseq --help
```

nfcore: The nf-core launch command

To make it easier to launch pipelines, these parameters are described in a JSON file, `nextflow_schema.json` bundled with the pipeline.

The `nf-core launch` command uses this to build an interactive command-line wizard which walks through the different options with descriptions of each, showing the default value and prompting for values.

nfcore: The nf-core launch command

Once all prompts have been answered, non-default values are saved to a `params.json` file which can be supplied to Nextflow using the `-params-file` option.

Optionally, the Nextflow command can be launched there and then.

To use the launch feature, just specify the pipeline name:

```
$ nf-core launch -r 3.0 rnaseq
```

Exercise

Exercise : nf-core launch rnaseq

Use the launch feature to create a params file named `nf-params.json`.

nfcore: Config files

nf-core pipelines make use of Nextflow's configuration files to specify how the pipelines run

For example,

- Define custom parameters
- software management system
 - Docker
 - Singularity
 - conda

nfcore: Config files: base.config

Nextflow can load pipeline configurations from multiple locations.

nf-core pipelines load configuration in the following order:



Always loaded. Contains pipeline-specific parameters and "sensible defaults" for things like computational requirements

nfcore: Config files: Core

nextflow.config



Default 'base' config ([always loaded](#))



Core profiles (e.g. docker, conda, test)

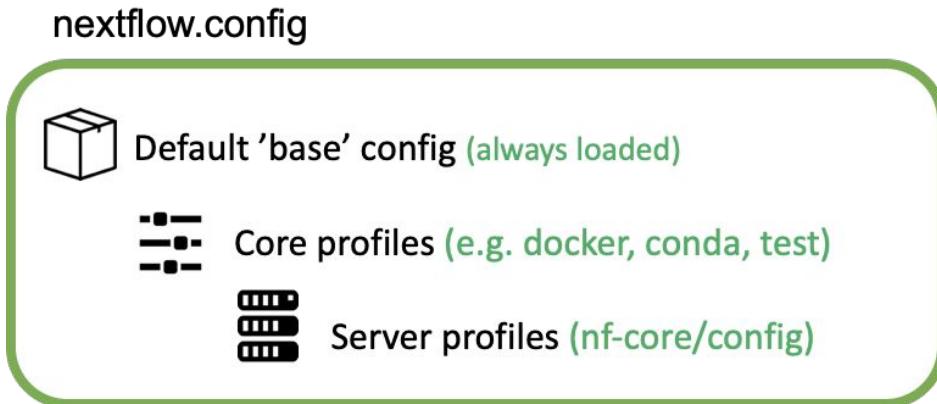
All nf-core pipelines come with some generic config profiles.

The most commonly used ones are for software packaging: docker, singularity and conda

* Other core profiles are debug and two test profiles. There two test profile, a small test profile (nf-core/test-datasets) for quick test and a full test profile which provides the path to full sized data from public repositories.

nfcore: Config files: Server Profiles nf-core/configs

Nextflow can load pipeline configurations from multiple locations. nf-core pipelines load configuration in the following order:



At run time, nf-core pipelines fetch configuration profiles from the configs remote repository

The profiles here are specific to clusters at different institutions.

nf-core/ configs

```
//Profile config names for nf-core/configs
params {
    config_profile_description = 'University of Edinburgh (eddie) cluster profile provided by nf-core/configs.'
    config_profile_contact = 'Alison Meynert (@ameynert)'
    config_profile_url = 'https://www.ed.ac.uk/information-services/research-support/research-computing/ecdf/high-performance-computing'
}

executor {
    name = "sge"
    queueSize = "100"
}

process {
    clusterOptions = { task.memory ? "-l h_vmem=${task.memory.bytes/task.cpus}" : null }
    scratch = true
    penv = { task.cpus > 1 ? "sharedmem" : null }

    // common SGE error statuses
    errorStrategy = {task.exitStatus in [143,137,104,134,139,140] ? 'retry' : 'finish'}
    maxErrors = '-1'
    maxRetries = 3

    beforeScript =
    """
    . /etc/profile.d/modules.sh
    module load 'roslin/singularity/3.5.3'
    export SINGULARITY_TMPDIR="$TMPDIR"
    """
}

params {
    saveReference = true
    // iGenomes reference base
    igenomes_base = '/exports/igmm/eddie/NextGenResources/igenomes'
    max_memory = 384.GB
    max_cpus = 32
    max_time = 240.h
}

env {
    MALLOC_ARENA_MAX=1
}

singularity {
    envWhitelist = "SINGULARITY_TMPDIR"
    runOptions = '-p'
    enabled = true
    autoMounts = true
}
```

<https://github.com/ameynert/configs/blob/master/conf/eddie.config>

Nfcore: Configuration: Profiles

nf-core makes use of Nextflow configuration *profiles* to make it easy to apply a group of options on the command line.

Configuration files can contain the definition of one or more profiles.

Common profiles are conda, singularity and docker that specify which software manager to use.

```
$ nextflow run nf-core/rnaseq -r 3.0 -profile test,conda
```

Nfcore: Configuration: Profiles

Adding institutional profile



```
$ nextflow run nf-core/rnaseq -r 3.0 -profile <inst_cfg_profile>, test, conda
```

```
$ nextflow run nf-core/rnaseq -r 3.0 -profile eddie, test, conda
```

nfcore: Config files: custom

Local config files given to Nextflow with the ` -c` flag

nextflow.config



Default 'base' config (always loaded)



Core profiles (e.g. docker, conda, test)



Server profiles (nf-core/config)



Your local config files

- \$HOME/.nextflow.config
- -c custom.config

Local config files given to Nextflow with the ` -c` flag

nfcore: Config files: custom

Local config files given to Nextflow with the `-c` flag

```
$ nextflow run nf-core/rnaseq -r 3.0 -c mylocal.config
```

nfcore: Config files: command line params

Command line configuration: pipeline parameters can be passed on the command line using the
--<parameter> syntax.

```
$ nextflow run nf-core/rnaseq -r 3.0 --email "my@email.com"
```

Exercise

Exercise create a custom config

Add the `params.email` to a file called `nfcore-custom.config`

Solution

nf-core: profile: test

The nf-core config profile *test* is a of a special case.

It specifies URLs for test data and all required parameters.

nf-core: profile: test

```
/*
=====
 Nextflow config file for running minimal tests
=====
 Defines input files and everything required to run a fast and simple pipeline test.

 Use as follows:
 nextflow run nf-core/rnaseq --profile test,<docker/singularity>

=====
 */

params {
    config_profile_name      = 'Test profile'
    config_profile_description = 'Minimal test dataset to check pipeline function'

    // Limit resources so that this can run on GitHub Actions
    max_cpus    = 2
    max_memory  = 6.GB
    max_time    = 6.h

    // Input data
    input       = 'https://raw.githubusercontent.com/nf-core/test-datasets/rnaseq/samplesheet/v3.4/samplesheet_test.csv'

    // Genome references
    fasta       = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/genome.fa'
    gtf         = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/genes.gtf.gz'
    gff         = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/genes.gff.gz'
    transcript_fasta = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/transcriptome.fasta'
    additional_fasta = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/gfp.fa.gz'

    bbsplit_fasta_list = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/bbsplit_fasta_list.txt'
    hisat2_index       = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/hisat2.tar.gz'
    star_index         = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/star.tar.gz'
    salmon_index       = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/salmon.tar.gz'
    rsem_index         = 'https://github.com/nf-core/test-datasets/raw/rnaseq/reference/rsem.tar.gz'

    // Other parameters
    skip_bbsplit      = false
    pseudo_aligner    = 'salmon'
    umitools_bc_pattern = 'NNNN'

    // When using RSEM, remove warning from STAR whilst building tiny indices
    modules {
        'rsem_preparereference' {
            args2 = "--genomeSAindexNbases 7"
        }
    }
}
```

<https://github.com/nf-core/rnaseq/blob/master/conf/test.config>

nf-core: profile: test

You can test any nf-core pipeline with the following command:

```
$ nextflow run nf-core/<pipeline_name> -profile test
```

Exercise

Exercise Run a test nf-core pipeline

Run the nf-core/hlatyping pipeline release 1.2.0 with the provided test data using the profile `test` and `conda`. Add the parameter `--max_memory 3G` on the command line. Include the config file, `nfcore-custom.config`, from the previous exercise using the option `-c`, to send an email when your pipeline finishes.

Code

```
$ nextflow run nf-core/hlatyping -r 1.2.0 -profile test,conda --max_memory 3G -c nfcore-custom.config
```

The pipeline does next-generation sequencing-based Human Leukozyte Antigen (HLA) typing and should run quickly.

```
$ nextflow run nf-core/hlatyping -r 1.2.0 -profile test,conda --max_memory 3G -c nfcore-custom.config
```

nf-core: pipelines offline

Many of the techniques and resources described above require an active internet connection at run time

pipeline files,

- configuration profiles
- software containers

dynamically fetched when the pipeline is launched.

This can be a problem for people using secure computing resources that do not have connections to the internet.

nf-core: pipelines offline

To help with this, the `nf-core download` command automates the fetching of required files for running nf-core pipelines offline.

The command can download a specific release of a pipeline with `-r/--release`

```
nf-core download nf-core/rnaseq -r 3.4
```

nf-core: pipelines offline

By default, the pipeline will download the pipeline code and the institutional nf-core/configs files.

If you specify the flag `--singularity`, it will also download any singularity image files that are required

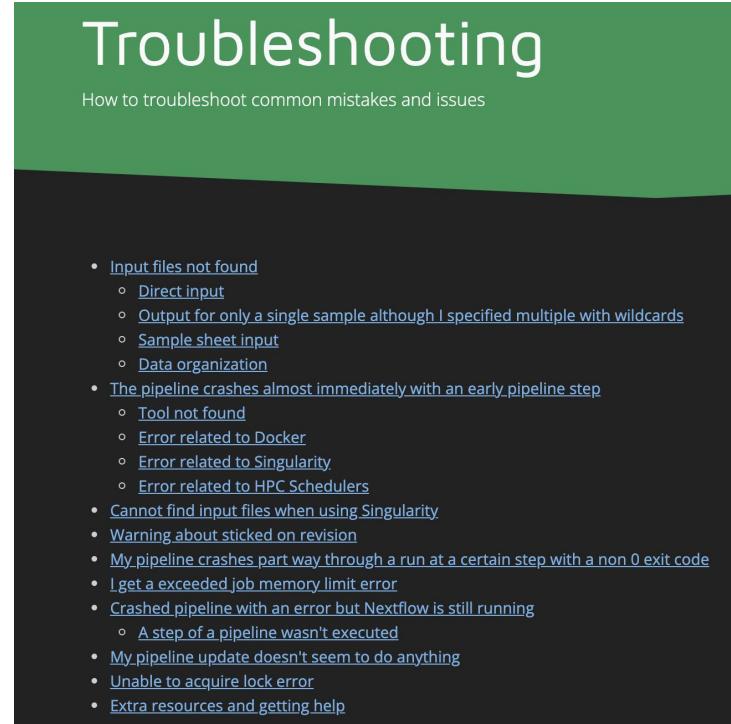
This needs Singularity to be installed.

All files are saved to a single directory, ready to be transferred to the cluster where the pipeline will be executed.

```
nf-core download nf-core/rnaseq -r 3.4 --singularity
```

nf-core: Troubleshooting

If you run into issues running your pipeline you can you the nf-core website to troubleshoot common mistakes and issues



The image shows a screenshot of a web page titled "Troubleshooting" with a green header. Below the title, it says "How to troubleshoot common mistakes and issues". The main content area is black and lists various troubleshooting topics:

- [Input files not found](#)
 - [Direct input](#)
 - [Output for only a single sample although I specified multiple with wildcards](#)
 - [Sample sheet input](#)
 - [Data organization](#)
- [The pipeline crashes almost immediately with an early pipeline step](#)
 - [Tool not found](#)
 - [Error related to Docker](#)
 - [Error related to Singularity](#)
 - [Error related to HPC Schedulers](#)
- [Cannot find input files when using Singularity](#)
- [Warning about stucked on revision](#)
- [My pipeline crashes part way through a run at a certain step with a non 0 exit code](#)
- [I get a exceeded job memory limit error](#)
- [Crashed pipeline with an error but Nextflow is still running](#)
 - [A step of a pipeline wasn't executed](#)
- [My pipeline update doesn't seem to do anything](#)
- [Unable to acquire lock error](#)
- [Extra resources and getting help](#)

<https://nf-co.re/usage/troubleshooting>

nf-core: Troubleshooting

Slack

💡 If you would like help with running nf-core pipelines, Slack is the best place to start.

Slack is a real-time messaging tool, with discussion split into channels and groups. We use it to provide help to people running nf-core pipelines, as well as discussing development ideas.

Once you have registered, you can access the nf-core slack at <https://nfcore.slack.com/> (NB: No hyphen!)

 Get invite to nf-core Slack



The nf-core Slack organisation has channels dedicated for each pipeline, as well as specific topics (eg. `#help`, `#pipelines`, `#tools`, `#configs` and much more).

<https://nf-co.re/join>

nf-core: Referencing a pipeline

Each release of an nf-core pipeline has a digital object identifiers (DOIs) for easy referencing in literature. The DOIs are generated by Zenodo from the pipeline's github repository.

Citations

If you use nf-core/rnaseq for your analysis, please cite it using the following doi: [10.5281/zenodo.1400710](https://doi.org/10.5281/zenodo.1400710)

An extensive list of references for the tools used by the pipeline can be found in the [CITATIONS.md](#) file.

You can cite the [nf-core](#) publication as follows:

The screenshot shows a Zenodo record page for the nf-core/rnaseq pipeline version 3.4. The page has a blue header with the Zenodo logo, search bar, and navigation links. Below the header, the title is "nf-core/rnaseq: nf-core/rnaseq v3.4 - Aluminium Aardvark". The page lists several contributors and their roles. A detailed list of changes (Changelog) is provided, showing enhancements and fixes. At the bottom, there are sections for parameters, dependencies, and a note about parameter updates.

October 5, 2021

nf-core/rnaseq: nf-core/rnaseq v3.4 - Aluminium Aardvark

Harshil Patel; Phil Ewels; Alexander Peltzer; Rickard Hammaren; Olga Botvinic; Gregor Sturm; Denis Moreno; Pranith Venuri; silviamorinos; Lorena Pantano; Gavin Kelly; FriederikeHansen; Maxime U. Garcia; nf-core bot; Chris Cheshire; rfenouil; marchocroissant; Peng Zhou; Gisela Gabernet; Jose Espinosa-Carrasco; Christian Mertes; Daniel Straub; Matthias Hörtenhuber; Paolo Di Tommaso; Sven F. George Hall; Senthilkumar Panneerselvam; Denis O'Meally; jun-wan; Alice Mayer

[3.4] - 2021-10-05 Enhancements & fixes

- Software version(s) will now be reported for every module imported during a given pipeline execution
- Added `python3` shebang to appropriate scripts in `bin/` directory
- [#4071] - Filter mouse reads from PDX samples
- [#5710] - Update ScrtMeRNA to use SilvaDB 138 (for commercial use)
- [#6991] - Error with post-trimmed read 2 sample names from FastQC in MultiQC report
- [#6993] - Cutadapt version missing from MultiQC report
- [#6997] - `pipeline_report.xhtml` missing from `pipeline.info` directory
- [#7053] - Sample sheet error check false positive

Parameters Old parameter New parameter `--bbsplit_fasta_list` `--bbsplit_index` `--save_bbsplit_reads` `--skip_bbsplit`

NB: Parameter has been **updated** if both old and new parameter information is present. **NB:** Parameter has been **added** if just the new parameter information is present. **NB:** Parameter has been **removed** if parameter information isn't present.

Software dependencies

nf-core : Key Points

- nf-core is a community-led project to develop a set of best-practice pipelines built using the Nextflow workflow management system.
- nf-core tools is a suite of helper tools that aims to help people run and develop pipelines.
- nf-core pipelines can found using the nf-core helper tool `--list` option or from the nf-core website.
- nf-core pipelines can be run using `nextflow run nf-core/<pipeline>` syntax, or launched and parameters configured using the nf-core helper tool launch option.
- nf-core pipelines can be configured by modifying nextflow config files and/or adding command line parameters.