Nextflow Intro

Outline

1. Getting Started with Nextflow
2. Workflow parameterization
3. Channels
4. Processes
5. Workflows

Getting Started with Nextflow

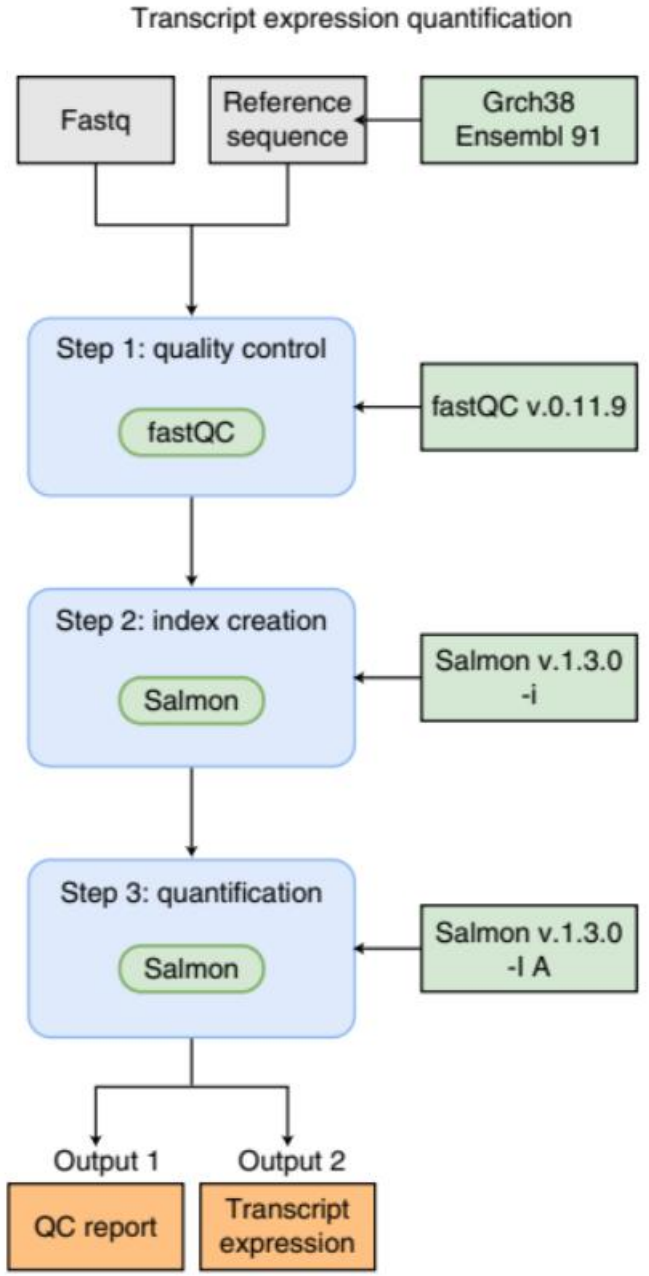
Objectives:

- Understand what a workflow management system is.
- Understand the benefits of using a workflow management system.
- Explain the benefits of using Nextflow as part of your bioinformatics workflow.
- Explain the components of a Nextflow script.
- Run a Nextflow script.

Workflows

- run a sequence of tasks
- often multiple software packages

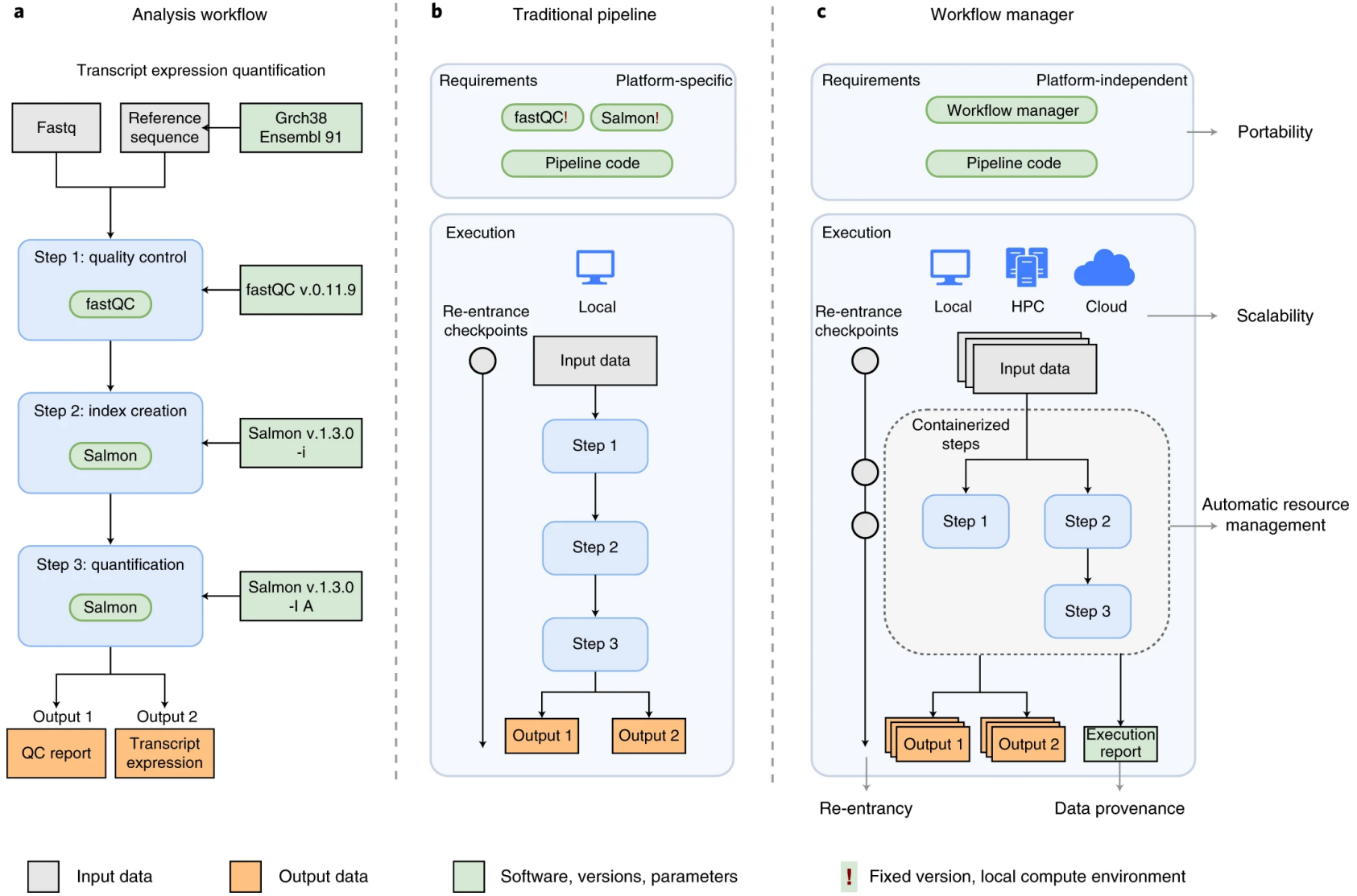
- as workflows become larger and more complex, the management of the programming logic and software becomes difficult
  - need a workflow language that handles this better than Bash or Python



Workflow management systems

- *Workflow Management Systems* (WfMS) such as Snakemake, Galaxy, and Nextflow have been developed specifically to manage computational data-analysis workflows in fields such as bioinformatics, imaging, physics, and chemistry.
- These systems contain multiple features that simplify the development, monitoring, execution and sharing of pipelines, such as:
  - Run time management
  - Software management
  - Portability & Interoperability
  - Reproducibility
  - Re-entrancy

Workflow management systems

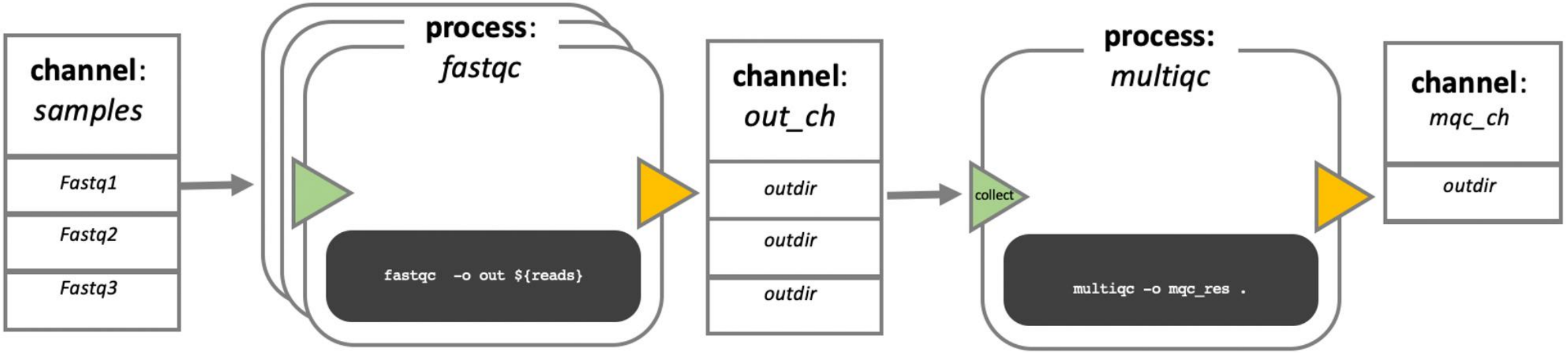


Wratten, L., Wilm, A. & Göke, J. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. Nat Methods 18, 1161–1168 (2021). https://doi.org/10.1038/s41592-021-01254-9

Processes, channels, and workflows

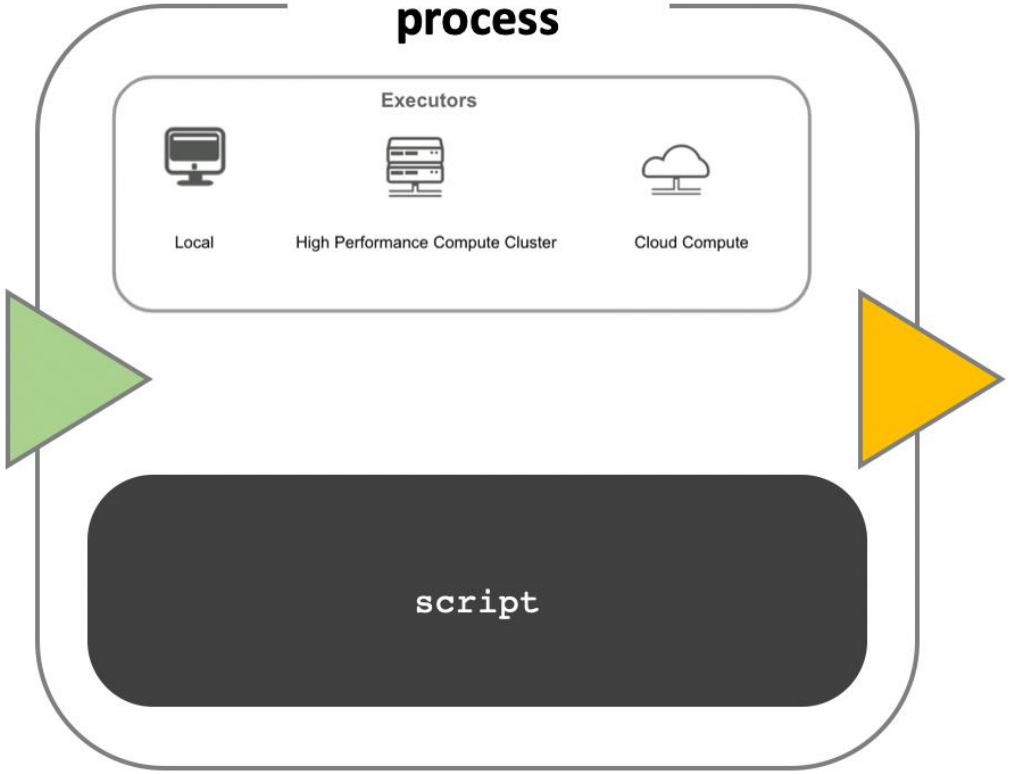
- Nextflow workflows have 3 main parts: ***processes***, ***channels***, and ***workflows***.
    1. ***Processes*** describe a task to be run. A process script can be written in any scripting language that can be executed by the Linux platform (Bash, Perl, Ruby, Python, R, etc.). Processes spawn a task for each complete input set. Each task is executed independently and cannot interact with other tasks. The only way data can be passed between process tasks is via asynchronous queues, called ***channels***.
    2. Processes define inputs and outputs for a task. *Channels* are then used to manipulate the flow of data from one process to the next.
    3. The interaction between processes, and ultimately the pipeline execution flow itself, is then explicitly defined in a ***workflow*** section.

Processes, channels, and workflows



Workflow execution

- While a process defines what command or script must be executed, the ***executor*** determines how that script is run in the target system.
- If not otherwise specified, processes are executed on the local computer. The local executor is very useful for pipeline development, testing, and small-scale workflows, but for large-scale computational pipelines, a High Performance Cluster (HPC) or Cloud platform is often required.
- Nextflow provides a separation between the pipeline's functional logic and the underlying execution platform. This makes it possible to write a pipeline once, and then run it on your computer, compute cluster, or the cloud, without modifying the workflow, by defining the target execution platform in a **configuration file**.
- Nextflow provides out-of-the-box support for major batch schedulers and cloud platforms such as Sun Grid Engine, SLURM job scheduler, AWS Batch service and Kubernetes.
  - A full list can be found here.



Your first script

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

workflow {
    SAY_HELLO()
    SAY_HELLO.out.view()
}

process SAY_HELLO {
    output:
    stdout  // Declares that the output will be sent to standard output
    script:
    """
    echo "Hello world"
    """
}
```

```
$ nextflow run ./hello_world.nf
 N E X T F L O W   ~  version 25.04.7

Launching `./hello_world.nf` [ridiculous_volta] DSL2 - revision: 573291b26b

executor >  local (1)
[83/2ee314] SAY_HELLO | 1 of 1 ✔
Hello world
```

Workflow parameterization

Objectives

- Use pipeline parameters to change the input to a workflow.
- Add a pipeline parameters to a Nextflow script.
- Understand how to create and use a parameter file.

Workflow parameterization

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


params.location = null


workflow {
    location_ch = Channel.of(params.location)
    SAY_HELLO(location_ch)
    SAY_HELLO.out.view()
}


process SAY_HELLO {
    input:
    val location


    output:
    stdout  // Declares that the output will be sent to standard output


    script:
    """
    echo "Hello ${location ?: 'World'}"
    """
}
```

```
$ nextflow run ./hello_location.nf
```

```
 N E X T F L O W   ~  version 25.04.7

Launching `./hello_location.nf` [chaotic_volta] DSL2 - revision: 994a7468c6

executor >  local (1)
[da/9f5213] SAY_HELLO (1) | 1 of 1 ✔
Hello World
```

Adding a parameter to a script

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


params.location = null


workflow {
    location_ch = Channel.of(params.location)
    SAY_HELLO(location_ch)
    SAY_HELLO.out.view()
}


process SAY_HELLO {
    input:
    val location


    output:
    stdout  // Declares that the output will be sent to standard output


    script:
    """
    echo "Hello ${location ?: 'World'}"
    """
}
```

```
$ nextflow run ./hello_location.nf --location Earth
```

```
 N E X T F L O W   ~  version 25.04.7

Launching `./hello_location.nf` [goofy_colden] DSL2 - revision: 0e88a68f12

executor >  local (1)
[2d/31682a] SAY_HELLO (1) | 1 of 1 ✔
Hello Earth
```

Create & use a parameter file (JSON)

```
{
  "location": "Georgia"
}
```

location-params.json

```
$ nextflow run ./hello_location.nf -params-file location-params.json

 N E X T F L O W   ~  version 25.04.7

Launching `./hello_location.nf` [trusting_mestorf] DSL2 - revision: 0e88a68f12

executor >  local (1)
[3b/0485b1] SAY_HELLO (1) | 1 of 1 ✔
Hello Georgia
```

Channels

Objectives

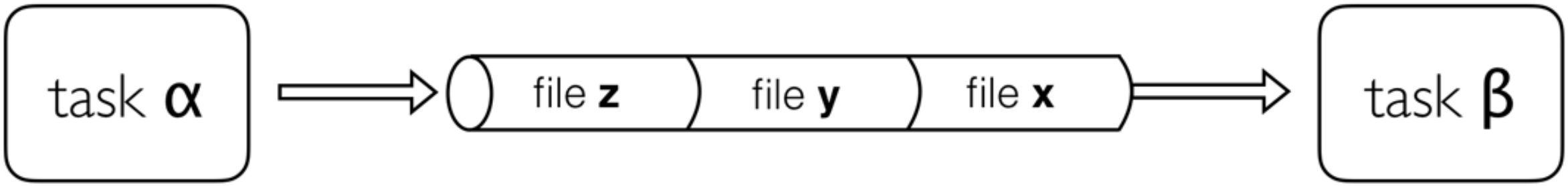
- Understand how Nextflow manages data using channels.
- Understand the different types of Nextflow channels.
- Create a value and queue channel using channel factory methods.
- Select files as input based on a glob pattern.
- Edit channel factory arguments to alter how data is read in.

Channels

- ***Channels*** are the way in which Nextflow sends data around a workflow.
- Channels connect processes via their inputs and outputs.
- Channels can store multiple items, such as files (e.g., fastq files) or values.
- When the process runs using one item from the input channel, we will call that run a ***task***.

Channels

- allows complex tasks to be split up, run in parallel
- asynchronous
- the first element into a channel queue is the first out of the queue (First in - First out).
- Channels only send data in one direction



Channel types

- 2 kinds of channels: **queue** channels and **value** channels

- **Queue** channels are a type of channel in which data is consumed (used up) to make input for a process/operator. Queue channels can be created in two ways:
  1. As the outputs of a process.
  2. Explicitly using channel factory methods such as Channel.of or Channel.fromPath.
- A **value** channel is bound to a **single** value. A value channel can be used an unlimited number times since its content is not consumed.

Creating Channels using Channel factories

- Channel factories are used to explicitly create channels.
  - In programming, factory methods (functions) are a programming design pattern used to create different types of objects (in this case, different types of channels).
- Channel factories are called using the Channel.<method> syntax, and return a specific instance of a Channel.

```
ch1 = Channel.value( 'GRCh38' )
ch2 = Channel.value( ['chr1', 'chr2', 'chr3', 'chr4', 'chr5'] )
ch3 = Channel.value( ['chr1' : 248956422, 'chr2' : 242193529, 'chr3' : 198295559] )
```

Queue channel factory

- Queue (consumable) channels can be created using the following channel factory methods.
  - Channel.of
  - Channel.fromList
  - Channel.fromPath
  - Channel.fromFilePairs
  - Channel.fromSRA

The ***of*** Channel factory

- 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()
```

```
chr1
chr3
chr5
chr7
```

The ***fromList*** Channel factory

- You can use the *Channel.fromList* method to create a queue channel from a list object.

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

queue_ch = Channel.fromList(ids)
value_ch = Channel.value(ids)
queue_ch.view()
value_ch.view()
```

```
ERR908507
ERR908506
ERR908505
[ERR908507, ERR908506, ERR908505]
```

The ***fromPath*** Channel factory

- The *fromPath* factory method creates a **queue channel** containing one or more files matching a file path.
- You can also use glob syntax to specify pattern-matching behavior for files.

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

```
data/yeast/reads/ref1_2.fq.gz
data/yeast/reads/etoh60_3_2.fq.gz
data/yeast/reads/temp33_1_2.fq.gz
data/yeast/reads/temp33_2_1.fq.gz
data/yeast/reads/ref2_1.fq.gz
data/yeast/reads/temp33_3_1.fq.gz
[..truncated..]
```

The ***fromFilePairs*** Channel factory

- creates a queue channel containing a tuple for every set of files matching a specific glob pattern (e.g., /path/to/*_{1,2}.fq.gz)

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

```
[etoh60_3, [data/yeast/reads/etoh60_3_1.fq.gz, data/yeast/reads/etoh60_3_2.fq.gz]]
[temp33_1, [data/yeast/reads/temp33_1_1.fq.gz, data/yeast/reads/temp33_1_2.fq.gz]]
[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]]
```

Processes

Objectives

- Understand how Nextflow uses processes to execute tasks.
- Create a Nextflow process.
- Define inputs and outputs to a process.
- Understand how to handle grouped input and output using the tuple qualifier.
- Understand how to use conditionals to control process execution.
- Use process directives to control execution of a process.
- Use the *publishDir* directive to save result files to a directory.

Processes

- We now know how to create and use *Channels* to send data around a workflow. We will now see how to run tasks within a workflow using ***processes***.

- A *process* is the way Nextflow executes commands you would run on the command line or custom scripts.

- a particular step in a workflow
- independent of each other

Processes

```
1  process COUNT_FILES {
2      script:
3      """
4      echo "Greetings!"
5      COUNTED_NEXTFLOW_FILES=\$(ls ${projectDir}/*.nf | wc -l)
6      echo "Number of .nf files found: \${COUNTED_NEXTFLOW_FILES}"
7      """
8  }
9
10 workflow {
11     COUNT_FILES()
12 }
```

```
$ nextflow run ./process_1.nf -process.debug
 N E X T F L O W  ~  version 25.04.7

Launching `./process_1.nf` [special_rosalind] DSL2 - revision: 5a534d16f5

executor >  local (1)
[3f/f47cf7] COUNT_FILES | 1 of 1 ✔
Greetings!
Number of .nf files found: 5
```

Processes

```
process COUNT_FILES {
    script:
    """
    #!/usr/bin/env python
    import os, sys
    import glob

    i=0
    for file in glob.glob("${projectDir}/*.nf"):
        i += 1
    print('file count: {}'.format(i))
    """
}

workflow {
    COUNT_FILES()
}
```

```
$ nextflow run ./process_python.nf -process.debug
 N E X T F L O W   ~  version 25.04.7

Launching `./process_python.nf` [fabulous_wright] DSL2 - revision: 248b2ac339

executor >  local (1)
[52/d48ab6] COUNT_FILES | 1 of 1 ✔
file count: 4
```

Processes

```
process COUNT_FILES {
    script:
    """
    count_files.py --add 7
    """
}

workflow {
    COUNT_FILES()
}
```

```
$ nextflow run ./process_associated_script.nf -process.debug
Launching `./process_associated_script.nf` [prickly_rosalind] DSL2 - revision: 18b6f32f5
b

executor >  local (1)
[f2/b88e0c] COUNT_FILES | 1 of 1 ✔
File count: 0
Added value: 7.0
Total: 7.0
```

Processes: Inputs

- The ***input*** block defines which channels the process is expecting to receive input from.
- The number of elements in input channels determines the process dependencies and the number of times a process executes.

- `val`: Access the input value by name in the process script.
- `path`: Handle the input value as a path, staging the file properly in the execution context.
- `env`: Use the input value to set an environment variable in the process script.
- `stdin`: Forward the input value to the process `stdin` special file.
- `tuple`: Handle a group of input values having any of the above qualifiers.
- `each`: Execute the process for each element in the input collection.

https://www.nextflow.io/docs/latest/process.html#inputs

Processes: Inputs

```groovy
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 process is executed only **twice**, because when a queue channel has no more data to be processed it stops the process execution.

```
$ nextflow run ./process_inputs.nf -process.debug
 N E X T F L O W   ~  version 25.04.7

Launching `./process_inputs.nf` [insane_coulomb] DSL2 - revision: f9b21fde48

executor >  local (2)
[5a/dd842e] COMBINE (1) | 2 of 2 ✔
1 and a

2 and b
```

Processes: Outputs

- Like the input, the type of ***output*** data is defined using type qualifiers.

- When an output file name contains a ***** or ***?***, it is interpreted as a pattern match.
  - Input files are not included in the list of possible matches.
  - Glob pattern matches against both files and directories path.
  - When a two stars pattern ****** is used to recurse through subdirectories, only file paths are matched i.e. directories are not included in the result list.

Processes: Directives

- Directive declarations allow the definition of optional settings, like the number of ***cpus*** and amount of ***memory***, that affect the execution of the current process without affecting the task itself.
- Directives are commonly used to define the amount of computing resources to be used or extra information for configuration (e.g., ***container***) or logging purpose such as (e.g., ***tag***).

- **Note:** You do not use **=** when assigning a value to a directive.

Processes: Directives

```
process ASSEMBLE_CONTIGS_SKESA {

    label "process_high"
    tag { "${meta.id}" }
    container "staphb/skesa@sha256:b520da51cd3929683c5eb94739bcd6c32045863dab16e777a4e02d2ff3802f20"

    input:
    tuple val(meta), path(cleaned_fastq_files)

    output:
    tuple val(meta), path("${meta.id}-${meta.assembler}.Raw_Assembly_File.tsv"), emit: qc_filecheck
    tuple val(meta), path("${meta.id}-SKESA_contigs.fasta")                    , emit: contigs
    path(".command.{out,err}")
    path("versions.yml")                                                       , emit: versions

```

Processes: Output Organizing

- 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. 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 in the process output block and then the output directory specified using the directive ***publishDir***. More information here.

Processes: Output Organizing

- 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. 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 in the process output block and then the output directory specified using the directive ***publishDir***. More information here.

Processes: Output Organizing

```
process COMBINE_FILES {
    publishDir "results/merged_files", mode: 'copy'

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

    output:
    path("${sample_id}.merged.txt")

    script:
    """
    cat ${reads} > ${sample_id}.merged.txt
    """
}

reads_ch = Channel.fromFilePairs('hello_{location,world}.nf')

workflow {
    COMBINE_FILES(reads_ch)
}
```

```
$ nextflow run ./process_publishDir.nf
 N E X T F L O W   ~  version 25.04.7

Launching `./process_publishDir.nf` [hopeful_pasteur] DSL2 - revision: 5c37ab2596

executor >  local (1)
[1e/b7536e] COMBINE_FILES (1) | 1 of 1 ✔
```

Workflows

Objectives

- Create a Nextflow ***workflow*** joining multiple processes.
- Understand how to connect processes via their inputs and outputs within a ***workflow***.

Workflows

Objectives

- Create a Nextflow ***workflow*** joining multiple processes.
- Understand how to to connect processes via their inputs and outputs within a ***workflow***.

Workflows

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

- In contrast to processes, the ***workflow*** definition in Nextflow does not require a name.
  - A named workflow is a ***subworkflow*** that can be invoked from other workflows. More information can be found in the official documentation here.

- A ***process*** is invoked as a function in the ***workflow*** scope, passing the expected ***input*** channels as arguments.

Workflows

```
process COMBINE_FILES {
    publishDir "results/merged_files", mode: 'copy'

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

    output:
    val(sample_id), emit: name
    path("${sample_id}.merged.txt"), emit: merged_files

    script:
    """
    cat ${reads} > ${sample_id}.merged.txt
    """
}
```

```
process COMPRESS_FILES {
    publishDir "results/merged_files", mode: 'copy'

    input:
    val(sample_id)
    path(text_file)

    output:
    path("${sample_id}.merged.txt.gz")

    script:
    """
    gzip -f ${text_file} > ${sample_id}.merged.txt.gz
    """
}

workflow {
    reads_ch = Channel.fromFilePairs('hello_{location,world}.nf')
    COMBINE_FILES(reads_ch)
    COMPRESS_FILES(
                  COMBINE_FILES.out.name,
                  COMBINE_FILES.out.merged_files
                  )
}
```

```
$ nextflow run ./workflow.nf
 N E X T F L O W   ~  version 25.04.7

Launching `./workflow.nf` [lonely_ekeblad] DSL2 - revision: db996b4ae6

executor >  local (2)
[13/72c867] COMBINE_FILES (1)   | 1 of 1 ✔
[92/124284] COMPRESS_FILES (1)  | 1 of 1 ✔
```

More Learning Resources

- https://training.nextflow.io/latest/
- https://nf-co.re/docs/
- https://carpentries-incubator.github.io/workflows-nextflow/
- https://github.com/bacterial-genomics/nextflow-intro