Nextflow cheatsheet: creating input channels

channel creation	channel content	use it in the process
`Channel.from( "A" )`	A	`input: val(x)` `""" echo $x """`

- Channel.from( ) will not alter anything

channel creation	channel content	use it in the process
`Channel.fromPath( "A" )` `Channel.fromPath( "/path/A" )`	/path/A	`input: path(x)` `""" cat $x """` `input: path("x.txt")` `""" cat x.txt """`

- If the full file path is absent, .fromPath( ) will prefix current folder as path
- So the resulting channels always carry a full path
- both work
- input: path("x.txt") will create a **symlink** in the working directory with the name "x.txt" (can use any name) pointing to /path/A

• From text files

input text file

```
A
B
C
```

channel creation

```
Channel.fromPath( "file.txt" )
    .splitText { it.strip( ) }
```

- .splitText will read in the file.txt
- it.strip( ) will remove the blank items

channel content

A
B
C

use it in the process

`input: val(x)`

- 1 channel, 3 items
- 3 parallel executions of process

```
Channel.fromPath( "file.txt" )
    .splitText { it.strip( ) }
    .map { it -> file(it) }
```

/path/A
/path/B
/path/C

`input: file(x)`

- file( ) adds current folder as path unless there is already full path in the item.

A	/path/A.bam	/path/A.bam.bai
B	/path/B.bam	/path/B.bam.bai
C	/path/C.bam	/path/C.bam.bai

```
Channel.fromPath( "file.tsv" )
   .splitCsv( sep: "\t" )
   .map { row -> [ row[0], file(row[1]), file(row[2]) ] }
```

[A, /path/A.bam, /path/A.bam.bai]
[B, /path/B.bam, /path/B.bam.bai]
[C, /path/C.bam, /path/C.bam.bai]

`input: tuple val(x), file(bam), file(bai)`

- .map{ } doesn't output of this example, but it is very useful to select columns and specify channel structure.

strain	bam
A	A.bam
B	B.bam
C	C.bam

```
Channel.fromPath( "file.tsv" )
   .splitCsv( header:true, sep: "\t" )
   .map { row ->
                if ( params.bam_path != "" ) {
                    row.bam = "${params.bam_path}/${row.bam}"
                    }
            [ row.strain, file("${row.bam}"), file("${row.bam}.bai") ] }
```

- If params.bam_path doesn't add a full path, file( ) will.

• Automatically from a list of files

file list

```
A_R1.fq
A_R2.fq
B_R1.fq
B_R2.fq
```

```
Channel.fromPath(  "*.fq"  )
```

/path/A_R1.fq
/path/A_R2.fq
/path/B_R1.fq
/path/B_R2.fq

`input: path(fq)`

- 1 channel, 4 items, 4 parallel executions of process

```
Channel.fromFilePairs( "/path/*_R{1,2}.fq" )
```

[A, [/path/A_R1.fq, /path/A_R2.fq]]
[B, [/path/B_R1.fq, /path/B_R2.fq]]

`input: tuple val(x), path(R1R2))`

```
Channel.fromFilePairs( "/path/*_R{1,2}.fq", flat:true )
```

[A, /path/A_R1.fq, /path/A_R2.fq]
[B, /path/B_R1.fq, /path/B_R2.fq]

`input: tuple val(x), path(R1), path(R2)`

```
A.bam
A.bam.bai
B.bam
B.bam.bai
```

```
Channel.fromFilePairs( "/path/*.{bam, bam.bai}", flat:true )
```

[A, /path/A.bam, /path/A.bam.bai]
[B, /path/B.bam, /path/B.bam,bai]

`input: tuple val(x), path(bam), path(bai)`

- First item "A" came from stripping the path and common pattern ".{bam, bam.bai}" as specified in Channel.fromFilePairs

Nextflow cheatsheet: combine inputs into 1 channel

examples of 1 channel:

Channel.from("A.fa", "B.fa", "C.fa" **)**

A.fa → • Each item (row) has 1 parallel execution of process
B.fa → • 1 channel, 3 items
C.fa → • 3 parallel executions of process

Channel.from(["A.fa", "B.fa", "C.fa" **])** → [A.fa, B.fa, C.fa] → • 1 channel, 1 item, 1 execution

Channel.from(["I.vcf", "I.vcf.tbi" **] ,**
["II.vcf", "II.vcf.tbi" **] ,**
["III.vcf", "III.vcf.tbi" **])**

[I.vcf, I.vcf.tbi] → • 1 channel, 3 items
[II.vcf, II.vcf.tbi] → • 3 parallel executions of process
[III.vcf, III.vcf.tbi] → • [ ] indicates a "set" "tuple" "ArrayList"

• With 1 input channel: change number of items and parallel execution of the process

input [A.fa, B.fa, C.fa] — input.**flatten**( ) → A.fa, B.fa, C.fa
• 1 channel, 3 parallel executions of the process.
• Note difference vs fromFilePairs(… flat:true)

input A.fa, B.fa, C.fa — input.**collect**( ) / input.**toSortedList**( ) → [A.fa, B.fa, C.fa]
• .collect( ) will maintain the order of items in the input channel, but if that channel is generated by a process, item order may be unpredictable.
• .toSortedList( ) will sort the items.

input [A.vcf, A.vcf.tbi], [B.vcf, B.vcf.tbi]
- input.**flatten**( ) → A.vcf, A.vcf.tbi, B.vcf, B.vcf.tbi
- input.**collect**( ) → [A.vcf, A.vcf.tbi, B.vcf, B.vcf.tbi]
- input.**toList**( ) / input.**toSortedList**( ) → **[[** A.vcf, A.vcf.tbi **] , [** B.vcf, B.vcf.tbi **]]**

.flatten( )

• .flatten( ) will break ALL layers of tuple
• .collect( ) and .combine( ) will break 1 layer of tuple
• .toList( ) and .concat( ) will keep all layers
• .map{ [it] } will add 1 layer to EACH item, aka put [ ] outside of each row

• With multiple input channels: combine them and change number of items

vcf [I.vcf]
index [I.vcf.tbi]
vcf.**combine**(index) → [I.vcf, I.vcf.tbi]
• Note how .combine( ) works when there are multiple items in each input channels (see bottom section).

vcf1 [I.vcf, I.vcf.tbi]
vcf2 [II.vcf, II.vcf.tbi]
- vcf1.**combine**(vcf2) → [I.vcf, I.vcf.tbi, II.vcf, II.vcf.tbi]
- vcf1.**concat**(vcf2) → [I.vcf, I.vcf.tbi] [II.vcf, II.vcf.tbi]

.collect( )
• .concat( ) and then .collect( ) sometimes gives the same result as .combine( ), but they are much more versatile for diverse input types

vcf [I.vcf, I.vcf,tbi] [II.vcf, II.vcf,tbi]
contig contigs.txt
vcf.**concat**(contig) → [I.vcf, I.vcf,tbi] [II.vcf, II.vcf,tbi] contigs.txt
.collect( ) → [I.vcf, I.vcf.tbi, II.vcf, II.vcf.tbi, contigs.txt]
• if all files, can be referred to as **input: path("*")**

vcf1 [I.vcf, I.vcf.tbi]
vcf1.**map { [it] }** → **[[** I.vcf, I.vcf.tbi **]]**
.combine(contig) → **[[** I.vcf, I.vcf.tbi **]**, contigs.txt **]**
• This is especially useful when number of items in vcf1 could vary. In process: **input: tuple path("*"), path(contig)**

• Other useful cases

vcf A.vcf, B.vcf
contig chr1, chr2
vcf.**combine**(contig) → [A.vcf, chr1] [A.vcf, chr2] [B.vcf, chr1] [B.vcf, chr2]
• combine = spread
• All combinations

input1 [A, 1] [B, 2]
• counting starts from 0
input1.**combine**(input2,**by:0**) → [A, 1, I] [B, 2, m] [B, 2, n]

input2 [A, I] [B, m] [B, n]
input1.**join**(input2) → [A, 1, I] [B, 2, m]
• .join( ) will keep only the first match

input2.**collect { it[1] }** → [I, m, n]
• "**it**" represents an item, which is used throughout the cheatsheet