Nextflow cheatsheet: creating input channels

Create a channel | **Channel content** | **Use the channel in the process**

Channel.of("A") → A → input: val(x)   """ echo $x """

• `Channel.of()` or `Channel.fromList()` will not alter anything

Channel.fromPath("A")
Channel.fromPath("/path/A") → /path/A →
input: path(x)   """ cat $x """
or 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

• input: `path("x.txt")` will create a **symlink** to /path/A in the working directory with the name "x.txt"

• Creating input channels from a text file

Input text file content | **Create a channel** | **Channel content** | **Use the channel in the process**

A
B
C

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

A
B
C
→ input: val(x)
• 1 channel, 3 items
• 3 parallel executions of process

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

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

/path/A
/path/B
/path/C
→ input: path(x)
• 1 channel, 3 items
• 3 parallel executions of process

• `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), path(bam), path(bai)

• `.map{ }` is very useful to select columns and specify channel structure. Here it converts the row to a tuple

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")] }

• A path is added with params.bam_path
• If "params.bam_path" doesn't add a full path, `file( )` will.
• [x, y] is the same as `tuple( x, y )`

• Creating input channels from a list of files

Files in the folder | **Create a channel** | **Channel content** | **Use the channel in the process**

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 →
• 1 channel, 4 items,
• 4 parallel executions of process
input: path(fq)

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.of("A.fa", "B.fa", "C.fa" **)**

A.fa →
B.fa →
C.fa →

- Each item (row) has 1 execution of process
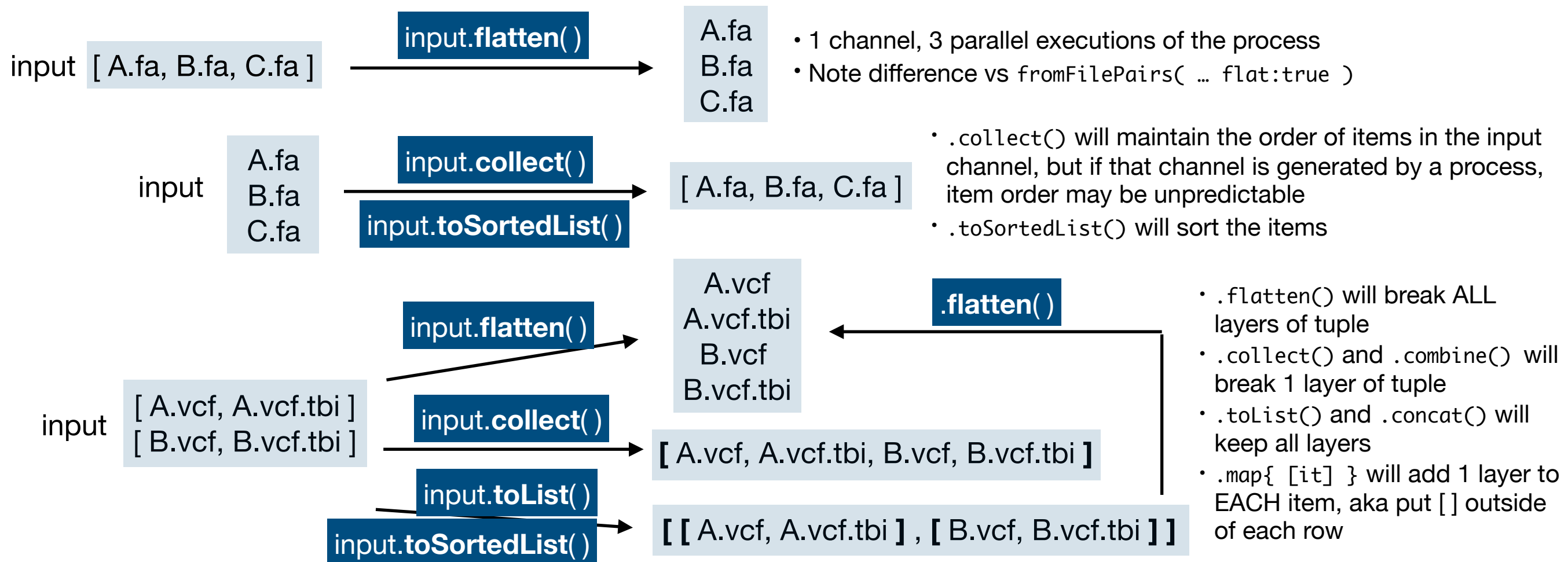- 1 channel, 3 items
- 3 parallel executions of the process

Channel.of(["A.fa", "B.fa", "C.fa" **])**

[A.fa, B.fa, C.fa] →
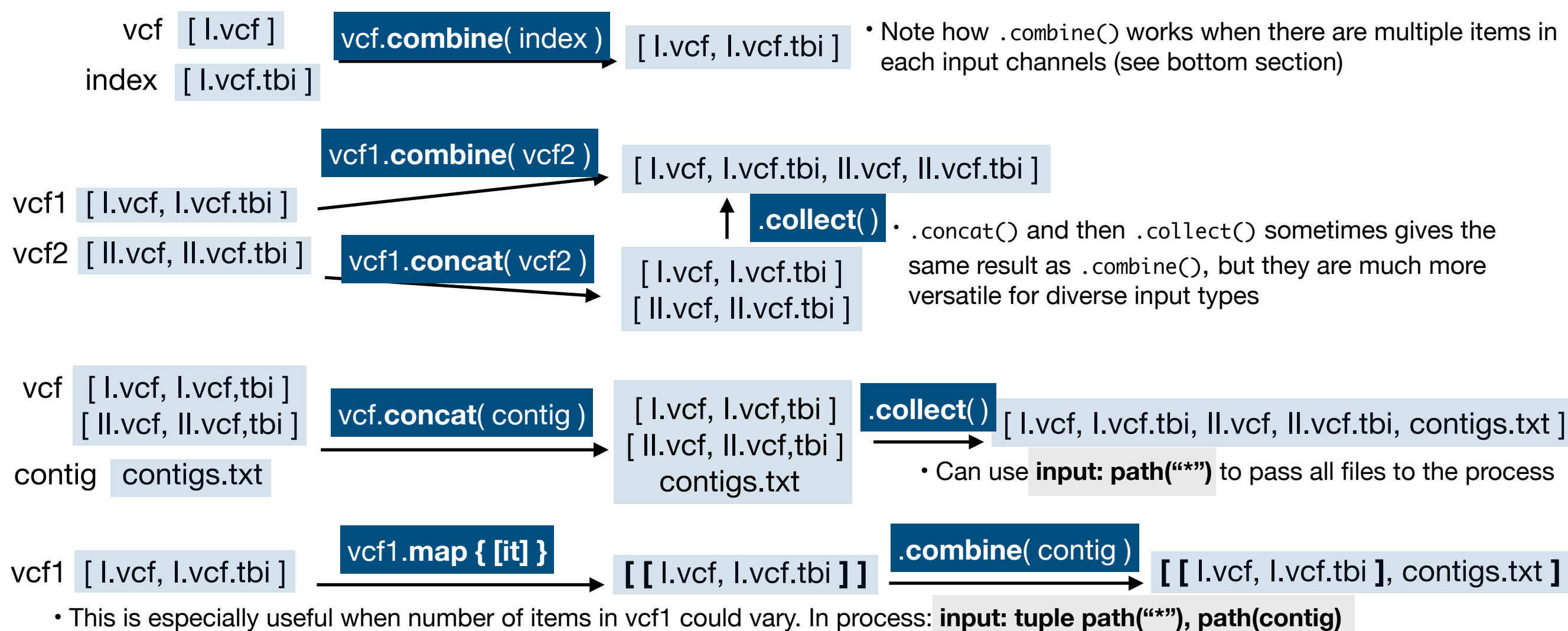
- 1 channel, 1 item, 1 execution

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

[I.vcf, I.vcf.tbi   ] →
[II.vcf, II.vcf.tbi] →
[III.vcf, III.vcf.tbi] →

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

← **.flatten( )**

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()` 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]

↑ **.collect( )**

vcf1.concat(vcf2) →

[I.vcf, I.vcf.tbi]
[II.vcf, II.vcf.tbi]

- `.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]

- Can use **input: path("*")** to pass all files to the process

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]

- All combinations

- counting starts from 0

input1

[A, 1]
[B, 2]

input1.combine(input2, by:0) →

[A, 1, I]
[B, 2, m]
[B, 2, n]

input2

[A, I]
[B, m]
[B, n]

input2.collect { it[1] } → [I, m, n]

- "**it**" represents an item, which is used throughout the cheatsheet

input1.join(input2) →

[A, 1, I]
[B, 2, m]

- `.join()` will keep only the first match