tardis.py

Contents

Description

tardis.py *conditions* a command for execution on a cluster. *Conditioning* a command means re-writing it to refer to conditioned input and output files. Conditioning an input file, means splitting it into chunks (and may include optional inline filtering and editing of records, before they are written to chunks) - each chunk is a conditioned input file, and yields a conditioned output file when the conditioned command is executed on the cluster. Once all of the conditioned commands have completed, the conditioned output is *unconditioned*, which means that it is merged together to yield the output file that was referred to by the original unconditioned command.

To indicate how a command should be conditioned, it is marked up with conditioning directives. For example, a command which runs a script to process a fastq input file and generate a fastq output file, such as

```
Perl split_pairEnds.pl L01_BWA_Trimmed.fastq L01_BWA_Trimmed_pair1.fastq
```

might be marked up as

```
Perl split_pairEnds.pl _condition_fastq_input_L01_BWA_Trimmed.fastq
_condition_fastq_output_L01_BWA_Trimmed_pair1.fastq
```

and run on the cluster using tardis

```
tardis.py -w -c 100000 Perl split_pairEnds.pl _condition_fastq_input_L01_BWA_Trimmed.fastq
_condition_fastq_output_L01_BWA_Trimmed_pair1.fastq
```

tardis has been developed using a condor based cluster.

Availability

Tardis is available from https://bitbucket.org/agr-bifo/tardis

Usage

Passing run parameters via Command-line

tardis.py [-w] [-c Chunksize] [-s SampleRate] [-from record_number] [-to record_number] [-d workingRootPath] [-k] [-v] any command (with optional conditioning directives)

-w Run the command as part of a workflow. After launching all of the jobs, tardis waits for all outputs, which are then collated and merged into a single output file, as specified by the output file path in the original command; all of the temporary input files (for example chunks of uncompressed fastq) are deleted provided all prior steps completed without error (if there was an error they are left there to assist with debugging). Without this option, the program exits immediately after launching all of the jobs, and output is left un-collated in the scratch folder created by this script, and no cleanup is done.

-c ChunkSize When conditioning the input file(s), split into files each containing Chunksize logical records. (A logical record for a sequence file is a complete sequence. For a text file it is a line of text). (If the -s option is used to sample the inputs, the chunksize relates to the full un-sampled file – so the same chunk-size can be used whether random sampling or not. For example a chunksize of 1,000,000 is specified in combination with a sampling rate of .0001, then each chunk would contain 100 sequences – i.e. you should not adjust the chunk-size, for the sampling rate. Note that to avoid a race-condition that could be caused by a very small chunk-size resulting in launching a very large number of jobs, tardis will throw an exception if the chunk-size used would result in launching more than MAX_DIMENSION jobs (currently 5000))

-from record_number When conditioning the input file(s), only use records from the input file after or including record_number (where that is logical record number – e.g. in a fastq file, start from record number N means start from sequence N). By combining this option with -to, you can process slices of a file. Note that this option has no affect when processing a list-file.

-to record_number When conditioning the input file(s), only use records up to and including the record record_number (where that is logical record number – e.g. in a fastq file, process up to record number N means process up to and including sequence N). By combining this option with -from, you can process slices of a file. Note that this option has no affect when processing a list-file.

-s SampleRate Rather than process the entire input file(s), a random sample of the records is processed. SampleRate is the probability that a given record will be sampled. For example -s .001 will result in roughly 1 in every 1000 logical records being sampled. When the -s option is specified, tardis does not clean up the conditioned input and output – e.g. all of the uncompressed fastq sample fragments would be retained. These are retained to assist with the Q/C work that is normally associated with a sampled run. Paired fastq input files are sampled in lock-step, provided the paired fastq conditioning directive is used for both files.

-d workingRootPath create the tardis working folder under workingRootPath. If no working root is specified, a default location is used.

-v validate the run by doing a dry run. This means that the chunks, job scripts and job files etc. are all generated but the jobs are not launched. The user can start then kill (CTRL-C) the run, inspect the script and job files that were generated to check that their command has been conditioned as envisaged.

-k keep the conditioned input and output - i.e. the input and output fragments. Normally in workflow mode these are deleted after the output is successfully "unconditioned" - i.e. joined back together

-t filename optionally supply a condor job template - tardis will read the contents of filename and use this as the job template.

-hpctype value optionally indicate the hpc environment. Currently the only supported values are :

default which results in tardis attempting to set up and launch condor jobs;

local which results in each job being launched by tardis itself on the local machine, using the native python sub-process API. The maximum number of processes it will run at a time is controlled by a global variable in the script MAX_PROCESSES, which is initially 10.

```
If the -hpctype option is not used, tardis assumes default and will try to set up and launch
condor jobs.

-batonfile value         if you supply a "baton file" name, tardis will write the process exit
code to this file after all processing has completed. This can be useful to preserve synchronous
execution of a workflow, even if tardis is started in the background - the workflow can test the
existence of the batonfile - if it exists then the corresponding tardis processing step has
completed - i.e. this is another way of a tardis based step in a workflow "passing the baton" to
the next step.

-h                    print usage and exit.
```

Passing run parameters via .tardishrc configuration file

You can also pass run-time parameters by specifying these in a .tardishrc file. Tardis will search for a configuration file in the current working folder, and also in /etc/tardis

Example:

```
[tardis_engine]
# filter sequences - minimum length 200
seqlength_min=200

# take 1 in 1000 random sample of the input
samplerate=.001

# job template to use
job_template_name=condor_job

#..or specify a filename, containing a custom job definition
jobtemplatefile=/somewhere/my_template.txt

#specify a shell template that uses bash
shell_template_name=condor_bash_shell

# you can supply a list of valid command patterns to restrict which commands
# the engine will execute - e.g. if tardis is running as a server
valid_command_patterns=cat awk [t]*blast[nxp] bwa bowtie flexbar

# by default tardis creates uniquely named working folders under the
# root folder specified by the -d argument. In some cases you might
# want to specify the working folder name yourself. To do that set
# workdir_is_rootdir to True - tardis will then use the named
# folder specified by the -d argument as its working folder
workdir_is_rootdir=False

# by default tardis uses a fast C based program (kseq_split) to condition
# fasta and fastq files. To use the slower biopython based method
#( for example if you want to include a sequence length filter)
# use the following configuration
[tardis_engine]
fast_sequence_input_conditioning=False
```

Conditioning Directives

Input conditioning

Input conditioning directives are prefixed to command arguments corresponding to input filenames. Input files that are prefixed with conditioning directives will

- If necessary be uncompressed (e.g. fastq.gz → fastq)
- If necessary be sampled randomly
- If necessary be format-converted (e.g. fastq → fasta) as implied by the directive
- Split into chunks

and the original command will be rewritten to refer to the conditioned files containing the conditioned data.

The following input conditioning directives are available

`_condition_fastq_input_`	optionally uncompress and then split a fastq file into fastq fragments
`_condition_fasta_input_`	optionally uncompress and then split a fasta file into fasta fragments
`_condition_fastq2fasta_input_`	optionally uncompress and then split a fastq file into fasta fragments
`_condition_paired_fastq_input_`	optionally uncompress and then split one of two pair-end matched fastq files into pair-end matched fastq fragments, enforcing integrity of pair-end matching by name
`_condition_text_input_`	optionally uncompress and then split a text file

Output conditioning

Output conditioning directives are prefixed to command arguments corresponding to output filenames. Output filenames that are prefixed with conditioning directives will be re-written into the conditioned command as conditioned filenames, so that conditioned input files yield output that is written to conditioned output filenames. Conditioned output filenames are obtained by adding chunk numbers to the original output filename.

At the conclusion of the run – i.e. when all of the conditioned commands have finished executing on the cluster – tardis will "uncondition" the output, which means joining all of the output fragments together, to yield the single output file referred to by the original command. This is also usually then automatically compressed.

The following output conditioning directives are available:

`_condition_fastq_output_`	
`_condition_fasta_output_`	
`_condition_uncompressedfastq_output_`	as above but don't compress the concatenated output
`_condition_uncompressedfasta_output_`	as above but don't compress the concatenated output
`_condition_text_output_`	generate conditioned output filenames corresponding to the prefixed output file. At the end of the run concatenate the output fragments into the prefixed filename and compress. (All of these directives result in simple text

	concatenation of the output and could be used interchangeably at the moment)
_condition_uncompressedtext_output_	as above but don't compress the concatenated output
_condition_sam_output_	generate conditioned output filenames corresponding to the prefixed output file. At the end of the run compress each SAM output file chunk to sorted (by sequence name) BAM and merge all of the resulting BAM fragments into a single output BAM file
_condition_uncompressedsam_output_	as for _condition_sam_output_ but final output is SAM not BAM
_condition_headlesssam_output_	as for _condition_sam_output_ but final output does not include the SAM header. Note that the unconditioned SAM output generated by the command when run on each chunk should include the SAM header, as this is used to help check the integrity of the job. For example the header of each SAM output chunk is compared, and an error flagged if they are not identical (excluding the @PG record, which may be different for different chunks).
_condition_pdf_output_	generate conditioned output filenames corresponding to the prefixed output file. At the end of the run combine the pdf fragments (using ghostscript), and compress the final results
_condition_blastxml_output_	generate conditioned output filenames corresponding to the prefixed output file. At the end of the run combine the blast XML fragments into a single valid blast XML output
_condition_output_	generate conditioned output command line tokens, but don't attempt to uncondition any output. These tokens can be referenced by product conditioning mark-up – see the flexbar example below.
_condition_wait_output_	this causes tardis to wait for the requested command to complete, and check that the output exists, but no further processing is done. This option can be used for example if you simply want to use tardis to launch a command on a cluster and wait for the job to finish.

Product conditioning

Some applications do not let you specify output filename(s), or generate output that you can redirect, and write their output to files with auto-generated names. These types of output are here termed "products" and are handled by product conditioning directives. Product conditioning directives are appended as new arguments to the command to be executed, rather than appended to output filenames as are output conditioning directives. Product conditioning directives specify how any products of the command are to be unconditioned - i.e. collated and merged – at the end of the run. (). Product conditioning directives include sub-expressions which are used by tardis to find the conditioned products, and to determine the name of the final target file into which the conditioned products from each chunk are to be merged.

There is a product conditioning directive corresponding to each output conditioning directive.

See examples of product conditioning below

Throughput conditioning

Throughput conditioning directives are used where a chain of commands is conditioned, and output from one command is input to the next. Output that is used as input to a subsequent command in a chain is here termed throughput.

There is a throughput conditioning directive corresponding to each output conditioning directive.

See examples of throughput conditioning below

Wait conditioning

Wait conditioning directives are used where you do not necessarily want to split up your input, but rather you want to launch a command on a cluster and then wait for it to complete – for example as part of a workflow

See examples of wait conditioning below.

Filtering and editing records

There are often situations where you want to filter or edit records before they are processed. For example reject short records (filtering); reformatting fastq sequence names for compatibility with parsing requirements of a tool (editing).

Rather than create filtered or edited copies of the original data files, you can configure tardis to filter and edit data as part of conditioning.

See examples of filtering and editing as part of conditioning, below

Conditioning Examples

Each example shows first the original command, then the command marked up with conditioning directives and typical tardis options.

No conditioning

```
ls -l  /dataset/resequencing/data
```

```
tardis.py ls -l  /dataset/resequencing/data
```

(This could be used as a convenient way to run a command on a single node of the cluster)

Executing a custom fastq processing script

```
perl split_pairEnds_3_files.pl 1049L_both_with_no_primers_or_adapters_BWA_Trimmed.fastq
1049L_both_with_no_primers_or_adapters_BWA_Trimmed_pair1.fastq
1049L_both_with_no_primers_or_adapters_BWA_Trimmed_pair2.fastq
1049L_both_with_no_primers_or_adapters_BWA_Trimmed_single.fastq
```

```
tardis.py -w -c 100000 -d /dataset/RNA-Seq/scratch perl split_pairEnds_3_files.pl
_condition_fastq_input_1049L_both_with_no_primers_or_adapters_BWA_Trimmed.fastq
_condition_fastq_output_1049L_both_with_no_primers_or_adapters_BWA_Trimmed_pair1.fastq
_condition_fastq_output_1049L_both_with_no_primers_or_adapters_BWA_Trimmed_pair2.fastq
```

```
_condition_fastq_output_1049L_both_with_no_primers_or_adapters_BWA_Trimmed_single.fastq
```

Executing a blast of a random sample of sequences from a compressed fastq file

first would need to uncompress and sample the input file manually – then execute something like

```
blastn -query L01.fastq.sampled.fa -num_threads 2 -db nt  -evalue 1.0e-10 -max_target_seqs 1 -
outfmt 7 -out contamination_check.txt
```

tardis handles uncompression and sampling the original compressed file

```
tardis.py -w -c 1000000 -s .0001 blastn -query _condition_fastq2fasta_input_L01.fastq.gz  -
num threads 2 -db nt  -evalue 1.0e-10 -max_target_seqs 1 -outfmt 7 -out
_condition_text_output_contamination_check.txt
```

Executing a blast of a random sample of sequences from a compressed fastq file, generating XML output for import into MEGAN

first would need to uncompress and sample the input file manually – then execute something like

```
blastn -query /dataset/JHI_High_Low_Sequencing_Data/L01_filtered_sampled.fastq -query_gencode 11
-db nt -outfmt 5 -evalue .00001 -out _condition_blastxml_output_L01_sample.xml
```

tardis handles uncompression, sampling the original compressed file and merging the XML chunks

```
tardis.py -c 1500000  -s .0001 -w blastn -query
_condition_fastq2fasta_input_/dataset/JHI_High_Low_Sequencing_Data/L01_filtered.fastq.gz -
query gencode 11 -db nt -outfmt 5 -evalue .00001 -out
_condition_blastxml_output_/dataset/JHI_High_Low_Sequencing_Data/scratch/ L01_sample.xml
```

Executing a blast with customised tabular output – escaping quotes

This uses a custom outfmt string

```
blastx -query
/dataset/Deer_Transcriptomics_Resources/scratch/deerTranscript/deerTranscript_and_mRNA-
seq_contigs.fa -num_threads 4 -db uniprot_sprot.fa -evalue 1.0e-6 -outfmt "7 qseqid qstart qend
qframe sacc stitle" -out deer.out
```

Note the need to escape the quotes

```
tardis.py  -w -c 1000 -d /dataset/Deer_Transcriptomics_Resources/scratch/tardis blastx -query
_condition_fasta_input_/dataset/Deer_Transcriptomics_Resources/scratch/deerTranscript/deerTransc
ript_and_mRNA-seq_contigs.fa -num_threads 4 -db uniprot_sprot.fa -evalue 1.0e-6 -outfmt \"7
qseqid qstart qend qframe sacc stitle\" -out _condition_text_output_deer.out
```

Launching a command file on the cluster or local machiner

A command file `commands.txt` is prepared containing commands to be run on the cluster

```
  tardis.py -w -d /dataset/GBS_Rumen_Metagenomes/scratch -c 1 source
 condition text input my_commands.txt  1\> _condition_text_output_my_commands.stdout 2\>
_condition_text_output_my_commands.stderr

or launch multiple processes on the local machine, tardis acts as scheduler / load balancer

tardis.py -w -hpctype local -d /dataset/GBS_Rumen_Metagenomes/scratch -c 1 source
 condition text input my_commands.txt  1\> _condition_text_output_my_commands.stdout 2\>
_condition_text_output_my_commands.stderr
```

Running hmmscan, using a custom job template

Command has two outputs

```
hmmscan -o vel98.21.hmm.tout --tblout vel98.21.tblout --noali Pfam-A.hmm
/dataset/Sand_Scarab_Gut_Transcriptome/active/rRNA/vel98.21.prot
```

Multiple outputs are fine. Note again that we can start with compressed sequence data. Here we include a custom job template (for example we may want to exclude a specific machine for some reason)

```
tardis.py -c 50 -d /dataset/Sand_Scarab_Gut_Transcriptome/scratch -w -t custom_job_template.dat
hmmscan -o _condition_text_output_vel98.21.hmm.tout --tblout
_condition_text_output_vel98.21.tblout --noali Pfam-A.hmm
_condition_fasta_input_/dataset/Sand_Scarab_Gut_Transcriptome/active/rRNA/vel98.21.prot.gz
```

Quickly Sub-sampling a fasta file using a list of sequence id's

The original command uses a simple bio-python script to filter a fasta file, to output only those sequences whose id's are in a list file.

```
cat allseqs.fasta  | fasta_yank.py gg_13_5_taxonomy.nonarchaea.names > selectedseqs.fasta
```

Where fasta_yank.py is simply :

```
#!/usr/bin/python2.6
# filters a fasta file (from standard input), pulling out sequences by name
#

from Bio import SeqIOimport sys
usage = "usage :  cat some_file.fa | fasta_yank.py file_of_names "

if len(sys.argv) != 2:
   print usage
   sys.exit(1)
```

```
SeqIO.write ((r for r in SeqIO.parse(sys.stdin, "fasta") if r.name in [name.strip() for name in
open(sys.argv[1],"r")]) , sys.stdout, "fasta")
```

….and `gg_13_5_taxonomy.nonarchaea.names` just contains sequence ids – e.g.

```
228054
844608
178780
198479
187280
etc
```

The original is very slow. The following marked up command splits the list file (not the sequence file), and launches jobs to filter the entire sequence file through chunks of the list file. This is much faster.

```
tardis.py -w -c 100 -d /dataset/scratch/tmp cat allseqs.fasta \| fasta_yank.py
_condition_text_input_gg_13_5_taxonomy.nonarchaea.names \>
_condition_fasta_output_selectedseqs.fasta
```

Product conditioning

This script writes 3 output files, with names constructed from the input file, as well as writing some stats to stdout

```
/usr/bin/DynamicTrim.pl -probcutoff 0.05 AD1AAFACXX_L001_R2_001.fastq
```

There are no DynamicTrim command-line options available to specify the filenames or redirect the output, so output conditioning is not possible. Product conditioning directives include sub-expressions which are used by tardis to find and collate the conditioned output. Here the sub-expressions indicate that the conditioned product filenames are obtained by appending the indicated suffices to the conditioned input filenames, and that these are to be collated into the filename that appears after the comma in the sub-expression. The conditioning directive prefixed to the redirected stdout means that the stdout of each chunk will be written to conditioned filenames AD1AAFACXX_L001_R2_001_trimming_stdout.1, AD1AAFACXX_L001_R2_001_trimming_stdout.2, etc, and then these will be concatenated to yield a final stdout file AD1AAFACXX_L001_R2_001_trimming_stdout.gz (merged output is compressed by default). Note that the shell redirect character needs to be escaped, so that it is passed through to tardis to be included in the command conditioning (rather than interpreted by the shell being used to run the tardis command).

```
tardis.py -w -c 500000 /usr/bin/DynamicTrim.pl -probcutoff 0.05
_condition_fastq_input_AD1AAFACXX_L001_R2_001.fastq \>_condition_text_output_
AD1AAFACXX_L001_R2_001_trimming_stdout
_condition_uncompressedfastq_product_.trimmed,AD1AAFACXX_L001_R2_001_trimmed.fastq
_condition_uncompressedtext_product_.trimmed_segments,AD1AAFACXX_L001_R2_001_segments.txt
_condition_uncompressedpdf_product_.trimmed_segments.hist.pdf,
AD1AAFACXX_L001_R2_001_segments_histogram.pdf
```

Another example of product conditioning

```
/usr/bin/bismark -q --phred33-quals -n 1 -l 50 -I 50 -X 350 --path_to_bowtie /usr/bin/
/bifo/active/Run938_5/Rachael/oarv3_conversion -1
/bifo/active/Run938_5/Rachael/C279BACXX_NZGL00461_423_250_NoIndex_L003_R1.fastq_Qtrimmed -2
/bifo/active/Run938_5/Rachael/C279BACXX_NZGL00461_423_250_NoIndex_L003_R2.fastq_Qtrimmed
```

In this case bismark would generate output filenames itself, as follows

```
C279BACXX_NZGL00461_423_250_NoIndex_L003_R1.fastq_Qtrimmed_bismark_pe.sam
C279BACXX_NZGL00461_423_250_NoIndex_L003_R1.fastq_Qtrimmed_Bismark_paired-end_mapping_report.txt
```

Product conditioning directives are used, so that tardis can find the output files for each chunk and join them back together to yield target filenames. As in the previous example, the product conditioning is of the form _condition_outputtype_product_***expected suffix,output product file name***. This directs tardis to look for output products from each input chunk named *input_chunk_name.****expected suffix***, and join these together to produce ***output product file name.*** Where, as in this case, there is more than one input file, then (for this product conditioning directive), tardis expects the output product file names to be based

on the chunks arising from the first input file encountered in the command .

```
tardis.py -w -c 500000 /usr/bin/bismark -q --phred33-quals -n 1 -l 50 -I 50 -X 350 --
path_to_bowtie /usr/bin/ /bifo/active/Run938_5/Rachael/oarv3_conversion -1
_condition_paired_fastq_input_/bifo/active/Run938_5/Rachael/C279BACXX_NZGL00461_423_250_NoIndex_
L003_R1.fastq_Qtrimmed -2
_condition_paired_fastq_input_/bifo/active/Run938_5/Rachael/C279BACXX_NZGL00461_423_250_NoIndex_
L003_R2.fastq_Qtrimmed _condition_text_product__Bismark_paired-end_mapping_report.txt,
C279BACXX_NZGL00461_423_250_NoIndex_L003_Bismark_paired-end_mapping_report.txt
_condition_uncompressedsam_product__bismark_pe.sam,
C279BACXX_NZGL00461_423_250_NoIndex_L003_bismark_pe.sam
```

- so that the final unconditioned (i.e. joined-back-together) output from this command will be

```
C279BACXX_NZGL00461_423_250_NoIndex_L003_Bismark_paired-end_mapping_report.txt
C279BACXX_NZGL00461_423_250_NoIndex_L003_bismark_pe.sam
```

Product conditioning with output basename specified by a command line arg

This utility (used to trim sequence reads) allows you to specify an output base-name which is then used to construct a number of output file names all with that base-name, but with different suffixes. For example

```
flexbar --threads 4 --reads /home/bloggs/fbproject/reads1.fastq --reads2 /home/bloggs/fbproject
/reads2.fastq --format sanger --max-uncalled 2 --min-read-length 35 --pre-trim-phred 20 --
/home/bloggs/adapters/adapters.txt --adapter-trim-end ANY --adapter-min-overlap 4 --adapter-
threshold 3 --single-reads --length-dist --target flexbartag  > fbrun.stdout
```

this tells the utility to use flexbartag as the basename. For these options, the utility will thus generate the following output files :

```
flexbartag_1.fastq
flexbartag_1_single.fastq
flexbartag_1.fastq.lengthdist
flexbartag_2.fastq
flexbartag_2_single.fastq
flexbartag_2.fastq.lengthdist
fbrun.stdout
```

The conditioning directive `_condition_output_somelabel` , will cause conditioned command-line arguments `somelabel.1` `somelabel.2` etc to appear in each conditioned command, however no output processing will be invoked. These conditioned labels can then be referenced by product conditioning directives like

```
_condition_text_product_{}.suffix,finalfile.dat
```

When used together with the `_condition_output_` directive, this directive means that at the end of the run tardis will expect to find conditioned output files `somelabel.1.suffix, somelabel.2.suffix` etc, and will join these together to yield a final unconditioned output file `finalfile.dat`

The marked-up version of the above command is thus for example

```
tardis.py -w -c 1000000 -d /data/working/myproject flexbar --threads 4 --reads
_condition_paired_fastq_input_/home/bloggs/fbproject/reads1.fastq --reads2
_condition_paired_fastq_input_/home/bloggs/fbproject/reads2.fastq --format sanger --max-uncalled
2 --min-read-length 35 --pre-trim-phred 20 --adapters /home/bloggs/adapters/adapters.txt --
adapter-trim-end ANY --adapter-min-overlap 4 --adapter-threshold 3 --single-reads --length-dist
--target _condition_output_flexbartag  \>
```

```
 condition uncompressedtext output /home/bloggs/fbproject/output/fbrun.stdout
 condition fastq product {} 1.fastq, /home/bloggs/fbproject/output/reads 1.fastq
 condition fastq product {} 2.fastq, /home/bloggs/fbproject/output/reads 2.fastq
_condition_fastq_product_{}_1_single.fastq, /home/bloggs/fbproject/output/reads_1_single.fastq
_condition_fastq_product_{}_2_single.fastq, /home/bloggs/fbproject/output/reads_2_single.fastq
_condition_uncompressedtext_product_{}_1.fastq.lengthdist,
/home/bloggs/fbproject/output/reads_1.fastq.lengthdist
 condition uncompressedtext product {} 2.fastq.lengthdist,
/home/bloggs/fbproject/output/reads_2.fastq.lengthdist
```

Product conditioning with an output folder specified by a command line arg

Tophat2 allows you to specify an output folder, into which it writes a number of output files (if you don't specify a folder, it uses a default of "tophat_out"). Example:

```
tophat2 -o th2temp --num-threads 4 -r 300 --mate-std-dev=20
/dataset/datacache/oar31/bowtie2_index/oar31 dataset_9767.fastq  dataset_9768.fastq
```

(where oar31 is the location of a reference genome, and the two fastq arguments are paired-end files)

This will write the following output files to the sub-folder `th2temp`:

```
accepted_hits.sam
align_summary.txt
deletions.bed
insertions.bed
junctions.bed
logs
prep_reads.info
unmapped.bam
```

As above, the conditioning directive `_condition_output_somelabel`, will cause conditioned command-line arguments `somelabel.1 somelabel.2` etc to appear in each conditioned command, however no output processing will be invoked. These conditioned labels can then be referenced by product conditioning directives like

```
_condition_text_product_{}/outputfile.dat,finalfile.dat
```

When used together with the `_condition_output_` directive, this directive means that at the end of the run tardis will expect to find conditioned sub-folder names `somelabel.1, somelabel.2 etc,` each containing a file `outputfile.dat`, and will join these together to yield a final unconditioned output file `finalfile.dat`

The marked up version of the above command is thus for example

```
tardis.py -w -c 1000000 -d /data/working/myproject tophat2 -o _condition_output_th2temp --num-
threads 4 -r 300 --mate-std-dev=20 /dataset/datacache/oar31/bowtie2_index/oar31
_condition_paired_fastq_input_dataset_9767.fastq
_condition_paired_fastq_input_dataset_9768.fastq
_condition_bam_product_{}/accepted_hits.bam,accepted_hits.bam
_condition_uncompressedtext_product_{}/deletions.bed,deletions.bed
 condition uncompressedtext product {}/insertions.bed,insertions.bed
 condition uncompressedtext product {}/junctions.bed,junctions.bed
_condition_uncompressedtext_product_{}/prep_reads.info,prep_reads.info
_condition_bam_product_{}/unmapped.bam,unmapped.bam
```

If you want to generate SAM output, use the tophat2 `--no-convert-bam` option, in conjunction with the tardis uncompressed SAM conditioning directive :

```
tardis.py -w -c 1000000 -d /data/working/myproject tophat2 --no-convert-bam -o
_condition_output_th2temp --num-threads 4 -r 300 --mate-std-dev=20
/dataset/datacache/oar31/bowtie2_index/oar31 _condition_paired_fastq_input_dataset_9767.fastq
_condition_paired_fastq_input_dataset_9768.fastq
_condition_uncompressedsam_product_{}/accepted_hits.sam,accepted_hits.sam
_condition_uncompressedtext_product_{}/deletions.bed,deletions.bed
 condition uncompressedtext product {}/insertions.bed,insertions.bed
_condition_uncompressedtext_product_{}/junctions.bed,junctions.bed
```

```
 condition uncompressedtext product {}/prep reads.info,prep_reads.info
_condition_bam_product_{}/unmapped.bam,unmapped.bam
```

(Note that tophat2 appears to always generate BAM format for the unmapped reads even when `--no-convert-bam` is specified - hence we have still used the tardis BAM conditioning directive to handle the unmapped reads output)

Product conditioning using a regular expression to specify output

Tophat2 allows you to specify an output folder, into which it writes a number of output files (if you don't specify a folder, it uses a default of "tophat_out"). Example:

```
tophat2 -o th2temp --num-threads 4 -r 300 --mate-std-dev=20
/dataset/datacache/oar31/bowtie2_index/oar31 dataset_9767.fastq  dataset_9768.fastq
```

(where oar31 is the location of a reference genome, and the two fastq arguments are paired-end files)

This will write the following output files to the sub-folder `th2temp`:

```
accepted_hits.sam
align_summary.txt
deletions.bed
insertions.bed
junctions.bed
logs
prep_reads.info
unmapped.bam
```

```
tardis.py -w -c 200000 -d /dataset/AG_1000_sheep/scratch/general_processing_062015/NZCMPF100016647269 -batonfile
/dataset/AG_1000_sheep/scratch/general_processing_062015/NZCMPF100016647269/C6KHFANXX-1418-36-05-
01_ATTCCT_L005_R1_001.baton /dataset/hiseq/active/bin/quadtrim -d sheep
_condition_paired_fastq_input_/dataset/BLGsheep/archive/NZGL01418_2/Raw/C6KHFANXX-1418-36-05-
01_ATTCCT_L005_R1_001.fastq.gz
_condition_paired_fastq_input_/dataset/BLGsheep/archive/NZGL01418_2/Raw/C6KHFANXX-1418-36-05-
01_ATTCCT_L005_R2_001.fastq.gz '_condition_uncompressedfastq_product_\S*?_R1_\S*?\d{5}-
pass.fq,/dataset/AG_1000_sheep/scratch/general_processing_062015/NZCMPF100016647269/C6KHFANXX-1418-36-05-
01_ATTCCT_L005_R1_001.fastq.quadtrim'  '_condition_uncompressedfastq_product_\S*?_R2_\S*?\d{5}-
pass.fq,/dataset/AG_1000_sheep/scratch/general_processing_062015/NZCMPF100016647269/C6KHFANXX-1418-36-05-
01_ATTCCT_L005_R2_001.fastq.quadtrim' '_condition_uncompressedfastq_product_\S*?_RX_\S*?\d{5}-
singleton.fq,/dataset/AG_1000_sheep/scratch/general_processing_062015/NZCMPF100016647269/C6KHFANXX-1418-36-05-
01_ATTCCT_L005_RX_001.singlefastq.quadtrim' \>
_condition_text_output_/dataset/AG_1000_sheep/scratch/general_processing_062015/NZCMPF100016647269/C6KHFANXX-
1418-36-05-01_ATTCCT_L005_001.stdout
```

Random sampling of (paired) fastq files

We want to take a "pair-aware" random sample of paired end sequence data, with both mates of every pair sampled extracted. Use the `-s` option to specify the sample rate (one in ten in this example)

```
tardis.py -s .1 -d /bifo/scratch/sampling cat _condition_paired_fastq_input_C7N0GANXX-1804-04-4-
1 L004 R1.fastq \>  condition fastq output_sample_R1.fastq \; cat
 condition paired fastq input_C7N0GANXX-1804-04-4-1_L004_R2.fastq \>
_condition_fastq_output_sample_R2.fastq
```

Several outputs, including stdout redirect

```
cutadapt -m 40 -f fastq -b AATGATACGGCGACCACCGAGATCTACACTCTTTCCCTACACGACGCTCTTCCGATCT -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACATCACGATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACCGATGTATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACTTAGGCATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACTGACCAATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACACAGTGATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACGCCAATATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACCAGATCATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACACTTGAATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACGATCAGATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACTAGCTTATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACGGCTACATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACCTTGTAATCTCGTATGCCGTCTTCTGCTTG 5kb_example.fastq --too-short-
output=tooshort.fastq --untrimmed-output=untrimmed.fastq > 5kb_example.cutadapt.fastq
```

```
tardis.py -w -c 1000 cutadapt -m 40 -f fastq -b
AATGATACGGCGACCACCGAGATCTACACTCTTTCCCTACACGACGCTCTTCCGATCT -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACATCACGATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACCGATGTATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACTTAGGCATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACTGACCAATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACACAGTGATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACGCCAATATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACCAGATCATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACACTTGAATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACGATCAGATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACTAGCTTATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACGGCTACATCTCGTATGCCGTCTTCTGCTTG -b
GATCGGAAGAGCACACGTCTGAACTCCAGTCACCTTGTAATCTCGTATGCCGTCTTCTGCTTG
_condition_fastq_input_ 5kb_example.fastq.gz --too-short-
output=_condition_uncompressedfastq_output_tooshort.fastq --untrimmed-
output=_condition_fastq_output_untrimmed.fastq \>
_condition_uncompressedfastq_output_5kb_example.cutadapt.fastq
```

Executing a short chain of commands – throughput conditioning

Originally 3 separate commands :

```
bwa aln protein_mRNAs.fa Lane2_1_Trimmed.fastq.gz > Lane2_1_Trimmed_vs_protein_mRNAs.sai

bwa aln protein_mRNAs.fa Lane2_2_Trimmed.fastq.gz > Lane2_2_Trimmed_vs_protein_mRNAs.sai

bwa sampe protein_mRNAs.fa Lane2_1_Trimmed_vs_protein_mRNAs.sai
Lane2_2_Trimmed_vs_protein_mRNAs.sai Lane2_1_Trimmed.fastq.gz Lane2_2_Trimmed.fastq.gz >
Lane2_Trimmed_vs_protein_mRNAs.sam
```

These can be chained together and conditioned as follows, using the _condition_throughput_ directive to condition the intermediate products. (Note that shell special characters ; and > that are to be passed through to the final conditioned command that gets executed on the cluster need to be escaped).

```
tardis.py -w -c 500000 bwa aln protein_mRNAs.fa _condition_paired_fastq_input_
Lane2_1_Trimmed.fastq.gz \> _condition_throughput_Lane2_1_Trimmed_vs_protein_mRNAs.sai \; bwa
aln protein_mRNAs.fa _condition_paired_fastq_input_Lane2_2_Trimmed.fastq.gz \>
_condition_throughput_Lane2_2_Trimmed_vs_protein_mRNAs.sai \; bwa sampe protein_mRNAs.fa
_condition_throughput_Lane2_1_Trimmed_vs_protein_mRNAs.sai
_condition_throughput_Lane2_2_Trimmed_vs_protein_mRNAs.sai
_condition_paired_fastq_input_Lane2_1_Trimmed.fastq.gz
_condition_paired_fastq_input_Lane2_1_Trimmed.fastq.gz \>
_condition_sam_output_Lane2_Trimmed_vs_protein_mRNAs.bam
```

List file processing

A common use case, is the need to process a large number of files through some pipeline.

For example 132 paired end files were received from the sequencing provider :

```
A04_NoIndex_L001_R1_001.fastq.gz  ... A04_NoIndex_L001_R1_066.fastq.gz
A04_NoIndex_L001_R2_001.fastq.gz  ... A04_NoIndex_L001_R2_066.fastq.gz
```

Processing these files in this form can be time-consuming and inefficient, for example

- We can merge these part files to make a single pair of input files. This merge is I/O intensive and slow, and will double the disk footprint of this data at least for a time (and possibly for the duration of the project on the system, since it would not be advisable to delete the originals post-merge in case of error) :

```
./merge_files.sh  # yields *_all.fastq
/usr/bin/DynamicTrim.pl -probcutoff 0.05 A04_NoIndex_L001_R1_all.fastq
/usr/bin/DynamicTrim.pl -probcutoff 0.05 A04_NoIndex_L001_R2_all.fastq
```

- We can process each file individually. This leads to a large number of input and output files to manage, which can be time-consuming and error prone.

```
/usr/bin/DynamicTrim.pl -probcutoff 0.05 A04_NoIndex_L001_R1_001.fastq
/usr/bin/DynamicTrim.pl -probcutoff 0.05 A04_NoIndex_L001_R1_002.fastq
/usr/bin/DynamicTrim.pl -probcutoff 0.05 A04_NoIndex_L001_R1_003.fastq
etc
```

tardis supports the transparent use of list files – i.e. files that contain the names of the actual files to be processed. A list file name can be used anywhere in a marked-up command, where a data file name would be used. Tardis automatically recognises files as being list files using the following rule :

1. The file name must end with `.list` (not case sensitive)
2. All non-trivial records in the file must correspond to either absolute or relative file paths on the system, to files that exist. (Relative file paths are relative to the folder containing the list file)

Using list files, all 132 files can be processed with just two commands, and we avoid the time and cost of having to merge the input files – they remain compressed in place. However the results will be as if the files had been merged – i.e. we will end up with (in this case) two large trimmed files (together with the stats and histograms products).

```
# prepare list file
ls A04_NoIndex_L001_R1_*.fastq.gz > A04_NoIndex_L001_R1.list
# execute the same command as usual - but with the list filename instead of a fastq(.gz) data
file name
tardis.py -w -c 1500000 /usr/bin/DynamicTrim.pl -probcutoff 0.05
_condition_fastq_input_A04_NoIndex_L001_R1.list
_condition_uncompressedfastq_product_.trimmed,A04_NoIndex_L001_R1.trimmed
_condition_uncompressedtext_product_.trimmed_segments,A04_NoIndex_L001_R1.trimmed_segments
_condition_uncompressedpdf_product_.trimmed_segments.hist.pdf,A04_NoIndex_L001_R1.trimmed_segmen
ts.hist.pdf
```

Piping blast results to a protein translation utility (referring to an input file twice, and avoiding intermediate files)

The big_blast_translate.py script uses blastx results to determine the translation frame of input sequences and translates them. It is designed for processing very large files (for example entire lanes of RNASeq data), so avoids loading blast hits into memory. To avoid large intermediate files we can pipe blast output to the utility. When a command refers to the same input file more than once as in this example, tardis only splits the file once, and each reference to that input in the conditioned command is bound with the same chunk name).

In this example, we want to process a 3% random sample of all lanes of an RNAseq experiment through this utility, so that we would need to sample each lane, translate to fasta and then pipe through the utility, i.e. something like…

```
# …sample 3% of each lane…
# …fastq-to-fasta each sample…
blastx -query lane1_sampled_formatted.seq -num_threads 4 -db uniprot_sprot.fa -evalue 1.0e-6 -
outfmt "7 qseqid qstart qend qframe stitle" | big_blast_translate.py -o
lane1_sampled_translated.seq -b - -f "GN\=\S+" lane1_sampled_formatted.seq
blastx -query lane2_sampled_formatted.seq -num_threads 4 -db uniprot_sprot.fa -evalue 1.0e-6 -
outfmt "7 qseqid qstart qend qframe stitle" | big_blast_translate.py -o
lane2_sampled_translated.seq -b - -f "GN\=\S+" lane2_sampled_formatted.seq
blastx -query lane3_sampled_formatted.seq -num_threads 4 -db uniprot_sprot.fa -evalue 1.0e-6 -
outfmt "7 qseqid qstart qend qframe stitle" | big_blast_translate.py -o
lane3_sampled_translated.seq -b - -f "GN\=\S+" lane3_sampled_formatted.seq
.
etc
.
```

(the –f argument to the script passes in a regexp filter used to filter translations – in this case we want to output only those seqs for which we did manage to obtain a consensus frame, via hits to genes)

This can be launched using a single command, using a list-file to process all lanes, fastq2fasta conditioning to do format conversion on the fly. Note how we need to escape the pipe symbol, and also the quoted –f argument. If you are unsure whether the escaping is correct, use the –v option to generate conditioned commands which are not launched, and examine the conditioned commands generated.

```
tardis.py -w -c 500000 -s .03 -d my_scratch_dir blastx -query
_condition_fastq2fasta_input_AllLanes.list -num_threads 4 -db uniprot_sprot.fa -outfmt \"7
qseqid qstart qend qframe sacc stitle\" \| big_blast_translate.py -b -  -f \"GN\=\S+\" -o
_condition_text_output_AllLanesSampled.genes.translated  _condition_fastq2fasta_input_
AllLanes.list
```

Wait conditioning - using tardis to launch a job on a cluster and wait for the output

The original is run as a direct command in a shell script as part of a pipeline

```
bwa mem -t 4 -M BWA_reference mysample.R1.fastq.flex.gz mysample.R2.fastq.flex.gz | samtools
view -bS - > mysample.bam
```

tardis can be used to launch the job on a cluster and wait for the output (i.e. no input splitting or output processing is done – tardis is just handling the admin of launching the job on the cluster, and synchronously waiting for the job to finish).

```
tardis.py -w -d my_work_dir bwa mem -t 4 -M  BWA_reference mysample.R1.fastq.flex.gz
mysample.R2.fastq.flex.gz \| samtools view -bS - \> _condition_wait_output_mysample.bam
```

In this case we want to use tardis to launch a bam indexing job onto the cluster, but wait for it to complete , as this is part of a pipeline.

The bam indexing command does not include the name of an output file, so in this case you wait for the (by)product of the

Using tardis in a make-file to implement a simple sequence processing pipeline

This simple pipeline processes paired fastq files through fastqc, flexbar and bwa. Here the flexbar and bwa steps are parallelised to run on the cluster using tardis, but the fastqc step is not. This make-file is based on a naming convention for the paired end files for the ABCProject, for example "LaneName_1.fastq.gz", "LaneName_2.fastq.gz" - it would need to be tweaked for other patterns (or generalised).

```
#
#ABCProjectworkflow version 0.1
#

# reference http://www.gnu.org/software/make/manual/make.html
# example :
# make -f ABC1.mk -d --no-builtin-rules /dataset/scratch/ABCProject/laneA.bam
#

datadir =  /dataset/scratch/ABCProject
workingdir = /dataset/scratch/ABCProject

#RUN_TARDIS=echo tardis.py # this is for a dry run. Change to next lines for the real run
#RUN_FASTQC=echo fastqc # this is for a dry run. Change to next lines for the real run
#

RUN_TARDIS=tardis.py # this is for a dry run. Change to next line for the real run
RUN_FASTQC=fastqc # this is for a dry run. Change to next line for the real run

# variables for tardis and other apps
TARDIS_chunksize=100000
TARDIS_workdir=/dataset/scratch/tardis
BWA_reference=/dataset/active/reference_genome.fa

##############################################
# how to make an ABC bam file. Made from
# flexbar output from both pairs, and indirectly also
# FastQC output
# Note that we only mention 1 of the pairs in the rule - but process both
##############################################

%.bam: %_1_fastqc.zip %_1.fastq.flex.gz
	$(RUN_TARDIS) -w -c $(TARDIS_chunksize) -d $(TARDIS_workdir) bwa aln -t 8
$(BWA_reference) _condition_paired_fastq_input_$*_1.fastq.flex.gz \>
_condition_throughput_$*_1.sai \; bwa aln -t 8 $(BWA_reference)
_condition_paired_fastq_input_$*_2.fastq.flex.gz \> _condition_throughput_$*_2.sai \; bwa sampe
$(BWA_reference) _condition_throughput_$*_1.sai _condition_throughput_$*_2.sai
_condition_paired_fastq_input_$*_1.fastq.flex.gz
_condition_paired_fastq_input_$*_2.fastq.flex.gz \> _condition_sam_output_$*



##############################################
# how to make flexbar intermediate files
# Note that we only mention 1 of the pairs in the rule - but process both
##############################################
```

```
%_1.fastq.flex.gz:

        $(RUN_TARDIS) -w -c $(TARDIS_chunksize) -d $(TARDIS_workdir) flexbar --threads 4 --reads
_condition_paired_fastq_input_$*_1.fastq.gz --reads2 _condition_paired_fastq_input_$*_2.fastq.gz
--format sanger --max-uncalled 2 --min-read-length 35 --pre-trim-phred 20 --adapters
/home/mccullocha/galaxy/flexbar_test/adapters.txt --adapter-trim-end ANY --adapter-min-overlap 4
--adapter-threshold 3 --single-reads --target _condition_output_flexbartag
_condition_fastq_product_{}_1.fastq,$*_1.fastq.flex
_condition_fastq_product_{}_2.fastq,$*_2.fastq.flex
_condition_fastq_product_{}_1_single.fastq,$*_1_single.fastq.flex
_condition_fastq_product_{}_2_single.fastq,$*_2_single.fastq.flex \>
_condition_text_output_$*.flex.stdout

##############################################
# how to make fastqc files (not parallelised)
##############################################
%_1_fastqc.zip:
        $(RUN_FASTQC) $*_1.fastq.gz -o $(workingdir) ; $(RUN_FASTQC) $*_2.fastq.gz -o
$(workingdir)

#
# keep the flexbar and fastqc output
.PRECIOUS: %_1_fastqc.zip  %_1.fastq.flex.gz %_2.fastq.flex.gz
```

Filtering records on the fly

You write your own filter function and specify it in the .tardishrc configuration file as in the example below. At runtime tardis compiles your function and uses it to filter records before they are written out to chunks. (Your function has to have the name "record_filter_func" or it will not be used)

```
[tardis_engine]

seqlength_min=200

def record_filter_func(seqrecord):
   # need to filter id like @M02810:27:000000000-AAJE2:1:1101:9707:1735/2
   # to
   # id like @M02810:27:000000000-AAJE2:1:1101:9707:1735 2:N:0:21
   # so that these are parsed OK by bwa paired alignment
   # and both names to @M02810:27:000000000-AAJE2:1:1101:9707:1735
   import re
   seqrecord.name = re.sub("\/1$","",seqrecord.name)
   seqrecord.name = re.sub("\/2$","",seqrecord.name)
   seqrecord.id = re.sub("\/2$", " 2:N:0:21",seqrecord.id)
   seqrecord.id = re.sub("\/1$", " 1:N:0:21",seqrecord.id)
   return seqrecord
```

Implementation

Overview

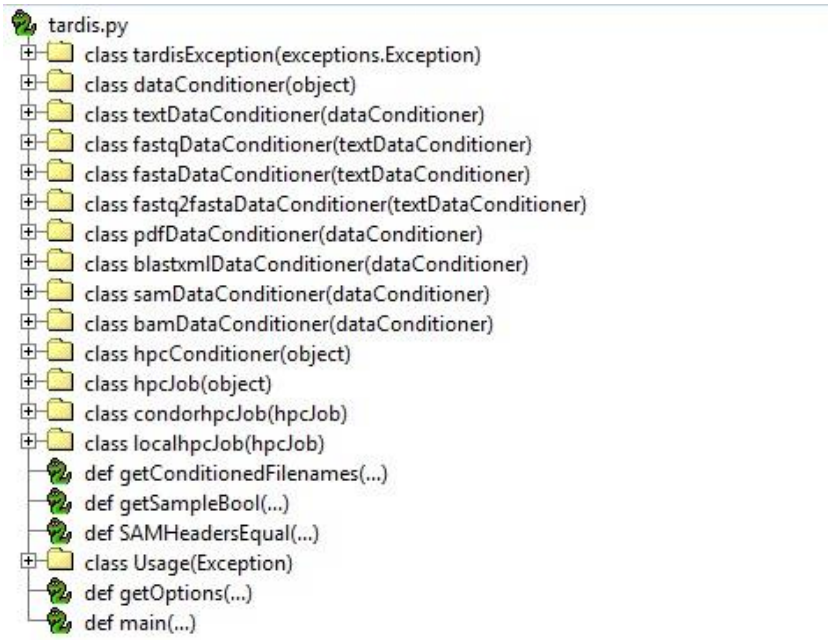

Figure 1 - Class Diagram

Related Utilities

tardis_analyze.py

tardis_analyze analyses a tardis run folder and optionally reports statistics and job repair information. The main use case currently is to look for “awol” chunks (that did not report an exit status, and hence may be incomplete), and also chunks that exited with an error code. tardis_analyze outputs commands that can be used to manually rerun the jobs that appear to be incomplete, or failed.

You only need to consider running tardis_analyze, if either tardis does not finish, or it exits with an error code. (There are normally three possible outcomes for a tardis run:

1. All chunk jobs exit with return status 0. Tardis collates all of the chunk outputs into your final joined-together-file, and itself then exits with code 0.

2. All jobs finish, but some finish with a non-zero exit code. tardis does not collate the outputs, because the result would be incomplete and possibly misleading. tardis itself exits with a non-zero code, so that the caller knows something went wrong.

3. Some jobs either do not finish, or “go awol” – i.e. no exit code is written to the chunk log file. In that case tardis will wait “forever” for the jobs it is expecting.

- only in situations (2) and (3) is there any need to consider running tardis_analyze.py – if tardis finishes and returns exit code 0, then “everything worked”, and there is no need to analyse the tardis working folder.

Note that in situation (3), if you compile a redo script using tardis_analyze and run it, and the original tardis process is still running and waiting, then as noted below, you can signal the waiting tardis master that the redo chunks have completed by uncommenting the lines indicated. Tardis will see all of the status codes written to the log files and will then wrap up the job by collating all of the chunks into your final joined-together-output (assuming all the codes were 0)

Usage:

```
tardis_analyze.py [-l path_to_log_folder ] [-x repair|statistics]
```

Example:

```
tardis_analyze.py -l /dataset/reseq_wf_dev/scratch/temp/inbfop03.agresearch.co.nz/tardis_xtlzI2
-x repair
```

Output:

```
tardis_analyze.py  using {'action': 'repair', 'logfolder':
'/dataset/reseq_wf_dev/scratch/temp/inbfop03.agresearch.co.nz/tardis_xtlzI2'}

        Analyzed 5000 log files.

        349 jobs need repair

        Summary of jobs by status (reported by condor):

        {0: 4651, None: 293, 2: 56}

        Suggested repair code is below, split into (up to ) 10 batches. You can for example
write this to 10 batch-files, then execute all of them concurrently (assuming you have
capacity). (Also included are commented out commands that will append a normal termination code
to each log file - if your tardis master job is still running, this will then pickup the
repaired output(it is polling for the termination line in each log file), and if all outputs are
picked up, will complete the job and merge the output. These commands can be uncommented and
executed after checking that the repairs ran without error, and completed)

                **** Batch 1 ****

            /dataset/reseq_wf_dev/scratch/temp/inbfop03.agresearch.co.nz/tardis_xtlzI2/run97.sh
# (return code = None)
            #echo "JOB REPAIR RESULTS : Normal termination (return value 0)" >>
/dataset/reseq_wf_dev/scratch/temp/inbfop03.agresearch.co.nz/tardis_xtlzI2/run97.log

            /dataset/reseq_wf_dev/scratch/temp/inbfop03.agresearch.co.nz/tardis_xtlzI2/run258.sh
# (return code = None)
            #echo "JOB REPAIR RESULTS : Normal termination (return value 0)" >>
/dataset/reseq_wf_dev/scratch/temp/inbfop03.agresearch.co.nz/tardis_xtlzI2/run258.log

            /dataset/reseq_wf_dev/scratch/temp/inbfop03.agresearch.co.nz/tardis_xtlzI2/run583.sh
# (return code = 2)
            #echo "JOB REPAIR RESULTS : Normal termination (return value 0)" >>
/dataset/reseq_wf_dev/scratch/temp/inbfop03.agresearch.co.nz/tardis_xtlzI2/run583.log

.

.

.

etc
```

Example – analyse all of the tardis working folders that were generated in the last week for a certain project

```
find /dataset/Sheep_GHG/scratch  -name "tardis*" -type d -mtime -7 -exec tardis_analyze.py  -l
{} -x repair \; > Weeks_analysis.log
```

Output:

```
As above, but concatenated for all of the working folders found
```

summarise_text.py

Often the non-sequence concatenated text output of a tardis run such as logging or statistics needs further processing to obtain totals. summarise_text.py is a simple utility that can be used to obtain summary totals.

Usage:

```
summarise_text.py [-h] -p PATTERN [-t {add}] [-l] filename
```

Appendix 1

Condor job file template example

```
Executable   = $script
Universe     = vanilla
error        = $script.err.$(Cluster).$(Process)
output       = $script.out.$(Cluster).$(Process)
log          = $log
#
# additional requirements - e.g. dynamically exclude specific machines
Requirements = (Machine != "fawlty.myorg.org") && (Machine != "basil.myorg.org")
Queue
```

Appendix 2

Galaxy wrapper modification example (fastx_trimmer.xml)

```
Original templating :

        <command>zcat -f '$input' | fastx_trimmer -v -f $first -l $last -o $output
#if $input.ext == "fastqsanger":
-Q 33
#end if
        </command>

        <inputs>
                <param format="fasta,fastqsolexa,fastqsanger" name="input" type="data"
label="Library to clip" />

                <param name="first" size="4" type="integer" value="1">
                        <label>First base to keep</label>
                </param>

                <param name="last" size="4" type="integer" value="21">
                        <label>Last base to keep</label>
                </param>
        </inputs>
```

```
With tardis related mark-up, and including the additional input fields

        <command>
#import string

#if str($use_hpc) == "yes":
  #if str($use_hpc_samplerate) != "":

    ## not strictly necessary to set up the command as an array
```

```
    ## of args and then join them as we do here , but sometimes works
    ## better

    #set $prog = ["tardis.py -w -c ", str($use_hpc_chunksize) , " -s ", str($use_hpc_samplerate)
, " -d /dataset/galaxy_scratch2011/scratch/tardis zcat -f _condition_fastq_input_", str($input),
" \\| fastx_trimmer -v -f ",str($first), " -l ", str($last), " -o
_condition_uncompressedfastq_output_", str($output)]
  #else:
    #set $prog = ["tardis.py -w -c ", str($use_hpc_chunksize) , " -d
/dataset/galaxy_scratch2011/scratch/tardis zcat -f _condition_fastq_input_", str($input), " \\|
fastx_trimmer -v -f ", str($first), " -l ", str($last), " -o
_condition_uncompressedfastq_output_", str($output)]
  #end if
  #set $prog = string.join($prog, "")
#else
  #set $prog = ["zcat -f ", str($input), " | fastx_trimmer -v -f ",str($first), " -l ",
str($last), " -o ", str($output)]
  #set $prog = string.join($prog, "")
#end if

 $prog

#if $input.ext == "fastqsanger":
  -Q 33
#end if
        </command>

        <inputs>
                <param format="fasta,fastqsolexa,fastqsanger" name="input" type="data"
label="Library to clip" />

                <param name="first" size="4" type="integer" value="1">
                        <label>First base to keep</label>
                </param>

                <param name="last" size="4" type="integer" value="21">
                        <label>Last base to keep</label>
                </param>

    <!—add HPC switch -->
    <param name="use_hpc" type="boolean" label="Use HPC cluster" truevalue="yes" falsevalue="no"
checked="False" help="the input file will be split, and chunks will be processed on the compute
farm. The chunk outputs will be pasted together before entry into your history"/>
    <param name="use_hpc_chunksize" type="integer" label="Chunk size" value="1500000"
min="100000" help="the number of sequences each split should contain"/>
    <param name="use_hpc_samplerate" type="float" optional="True" label="sample rate (leave
empty to process the complete file)" value="" min=".000001" max="1.0" help="for example entering
.001 will randomly sample and process roughly 1 in every 1000 records"/>
        </inputs>
```