genomics-bcftbx Documentation

Release 1.5.2

Peter Briggs

Sep 28, 2018

Contents

A set of utility programs and scripts plus a Python library developed to support NGS and genomics-related bioinformatics within the Bioinformatics Core Facility (BCF) of the Faculty of Biology, Medicine and Health (FBMH) at the University of Manchester (UoM).

CHAPTER 1

Installation and set up

1.1 Installing the genomics/bcftbx package

It is recommended to install directly from github using `pip`:

```
pip install git+https://github.com/fls-bioinformatics-core/genomics.git
```

from within the top-level source directory to install the package.

To use the package without installing it first you will need to add the directory to your `PYTHONPATH` environment, and reference the scripts and programs using their full paths.

1.2 Dependencies

The package consists predominantly of code written in Python, which has been used extensively with Python 2.6 and 2.7.

In addition there are scripts requiring:

- `bash`
- `Perl`
- `R`

and the following packages are required for subsets of the code:

- Perl: `Statistics::Descriptive` and `BioPerl`
- python: `xlwt`, `xlrd` and `xlutils`

Finally, some of the utilities also use 3rd-party software packages, including:

Core software

- `bowtie` http://bowtie-bio.sourceforge.net/index.shtml

- `fastq_screen` http://www.bioinformatics.bbsrc.ac.uk/projects/fastq_screen/
- `convert` (part of `ImageMagick` http://www.imagemagick.org/)

Illumina-specific

- `BCL2FASTQ` http://support.illumina.com/downloads/bcl2fastq_conversion_software_184.html
- `FastQC` http://www.bioinformatics.babraham.ac.uk/projects/fastqc/

SOLiD-specific

- `solid2fastq` (part of `bfast` http://sourceforge.net/projects/bfast/) - there are alternatives, see for example http://kevin-gattaca.blogspot.co.uk/2010/05/plethora-of-solid2fastq-or-csfasta.html
- `SOLiD_preprocess_filter_v2.pl` See https://www.biostars.org/p/71142/

1.3 Set up reference data

1.3.1 bowtie indexes

`fastq_screen` needs bowtie indexes for each of the reference genomes that you want to screen against.

The `fetch_fasta.sh` script can be used to acquire FASTA files for genome builds of common reference organisms, for example:

```
mkdir -p data/genomes/PhiX
cd data/genomes/PhiX/
fetch_fastas.sh PhiX
```

To generate bowtie indexes, use the `bowtie_build_indexes.sh` script, for example:

```
mkdir -p data/genomes/PhiX/bowtie
cd data/genomes/PhiX/bowtie/
bowtie_build_indexes.sh ../fasta/PhiX.fa
```

(This will create both colorspace and nucleotide space indexes by default.)

(Use `bowtie2_build_indexes.sh` to build indexes for bowtie2. Note that bowtie2 does not support colorspace.)

(Alternatively use `build_indexes.sh` to make all the indexes: bfast, bowtie and bowtie2, and SRMA.)

For rRNAs, get the `rRNAs.tar.gz` file and run the `build_rRNA_bowtie_indexes.sh` script, for example:

```
cd data/genomes/
wget .../rRNAs.tar.gz
build_rRNA_bowtie_indexes.sh rRNAs.tar.gz
```

which will extract the FASTA sequences to a subdirectory `rRNAs/fasta/` and create nucleotide- and colorspace bowtie indexes in `rRNAs/bowtie`.

1.3.2 fastq_screen configuration files

The QC scripts currently that there will be the following three `fastq_screen` configuration files:

- `fastq_screen_model_organisms.conf`
- `fastq_screen_other_organisms.conf`

- `fastq_screen_rRNA.conf`

(The actual form of the names are:

```
fastq_screen_<NAME><EXT>.conf
```

where `<NAME>` is one of `model_organisms`, `other_organisms` or `rRNA`, and `<EXT>` is an extension which is used to distinguish between nucleotide- and colorspace indexes.)

Each configuration file defines "databases" with lines of the form:

```
DATABASE     Fly (dm3)       /home/data/genomes/dm3/bowtie/dm3_het_chrM_chrU
```

for nucleotide space indexes, and

```
DATABASE     Fly (dm3)       /home/data/genomes/dm3/bowtie/dm3_het_chrM_chrU_c
```

for colorspace. (In each case the path is the base name for the index files.)

1.4 Create *qc.setup*

When the package is installed a template `qc.setup.sample` file is created in the `config` subdirectory - it needs to be copied to `qc.setup` and edited to set the locations for external software and data.

CHAPTER 2

Usage

The package provides utilities to support specific bioinformatics tasks

2.1 Reference data preparation

2.1.1 Genome sequence data and indexes

Suggested directory structure

For a given genome build, the recommended basic structure is (e.g. for *Rattus norvegicus* `rn4`):

```
rn4/
 |
 +- rn4.info  (metadata file)
 |
 +- fasta/    (sequences)
 |
 +- bowtie/   (indexes for bowtie - both color- and letterspace)
 |
 +- bfast/    (indexes for bfast)
 |
 +- liftOver/ (chain files for liftOver from this assembly to others)
 |
 +- seq/      (per-chromosome nib files for sequence alignments)
 |
...
```

i.e. a top-level directory containing a `.info` file, plus directories for FASTA sequences and derived or additional genome indexes for various aligners and other programs.

Note that the indexes for SRMA are placed in the "fasta" directory, as SRMA needs .fa, .fai and .dict files all to be placed in the same directory.

Creating a directory for a new genome

To add a new genome index:

- Create a new top-level directory for the organism and genome build
- Create a `fasta` subdirectory to hold the sequence data, and download and prepare the FASTA file(s) within this directory (see below for hints)
- Create a `.info` file and record the details of the genome for future reference (see below for more detail on .info files)
- Create and populate `bowtie`, `bfast` etc subdirectories with the appropriate indexes (see below for advice on generating indexes)

Download and prepare FASTA genome files

Note: The *fetch_fasta.sh* script is intended to reproducibly create FASTA files for a set of genomes.

To see which genomes are available run the program without any arguments; to obtain the FASTA file do e.g.:

```
fetch_fasta.sh mm9
```

Where the reference genome is a collection of fasta files for each chromosome, it's necessary to prepare a single file for the bfast and bowtie index generation by concatenating them together, e.g.:

```
cat chr* > hg18_random_chrM.fa
```

The individual chromosome fasta files can then be removed or archived, e.g.:

```
tar -cvf hg18_random_chrM.tar chr*
gzip hg18_random_chrM.tar
```

2.1.2 `.info` metadata files

Standard practice when add a new genome index is to also create a `.info` file (for example `hg18_random_chrM.info`).

These are hand-generated text files consisting of header fields followed by free text.

A typical header looks like (e.g. from `mm9_random_chrM.info` for *Mus musculus* `mm9`):

```
# Organism: Mus musculus
# Genome Build: MM9/NCBI37 July 2007
# Manipulations: Base chr. (1 to 19, X, Y), chrN_random, chrM and chrUn_random -␣
↪unmasked
# Source: wget http://hgdownload.cse.ucsc.edu/goldenPath/mm9/bigZips/chromFa.tar.gz
```

The free text area can contain any additional information that the person preparing the indexes thinks is important (for example, scripts or commands used to generate the indexes for individual programs).

2.1.3 Generate indexes for mapping software

This package includes a number of scripts for fetching and generating genome indexes for `Bfast`, `Bowtie` and `SRMA`.

- *bowtie_build_indexes.sh* can be used to generate color- and nuleotide-space indexes from a FASTA file.

  To use, go to the bowtie subdirectory for the genome and do e.g.:

  ```
  qsub -b y -V -cwd bowtie_build_indexes.sh ../fasta/genome.fa
  ```

  This will create both color and nucleotide space indexes; to only generate colorspace use the `--cs` option of the script, to only get nucleotide space use `--nt`.

- *bowtie2_build_indexes.sh* generates indexes for `bowtie2` (letter space only).
- *bfast_build_indexes.sh* prepares indexes for `bfast`.
- *srma_build_indexes.sh* prepare indexes for `SRMA`.

2.2 Preparing Illumina Data for analysis

2.2.1 Background

This section outlines the general structure of the data from Illumina based sequencers (GA2x, HiSEQ and MiSEQ) and the procedures for converting these data into FASTQ format.

Primary sequencing data

The software on the various sequencers performs image analysis and base calling, producing primary data files in either `.bcl` (binary base call) format, or (for newer instruments), a compressed version `.bcl.gz`.

Additional software is required to convert these data files to FASTQ format, and in the case of multiplexed runs also perform demultiplexing of the data.

The directories produced by the runs have the format:

```
<date_stamp>_<instrument_name>_<run_id>_FC
```

(For example `120518_ILLUMINA-13AD3FA_00002_FC`)

The components are:

- `<date-stamp>`: a 6-digit date stamp in year-month-day format e.g. `120518` is 18th May 2012
- `<instrument_name>`: name of the Illumina instrument e.g. `ILLUMINA-13AD3FA`
- `<run_id>`: id number corresponding to the run e.g. `00002`

A partial directory structure is shown below:

```
<YYMMDD>_<machinename>_<XXXXX>_FC/
         |
         +-- Data/
         |     |
         |     +------ Intensities/
         |                    |
         +                    +-- .pos files
```

(continues on next page)

(continued from previous page)

```
        |                   |
        |                   +-- config.xml
        +-- RunInfo.xml     |
                            +-- L001(2,3...)/  (lanes)
                            |
                            +-- BaseCalls/
                                   |
                                   +-- config.xml
                                   |
                                   +-- SampleSheet.csv
                                   |
                                   +--L001(2,3...)/  (lanes)
                                         |
                                         +-- C1.1/   (lane and cycle)
                                               |
                                               +-- .bcl(.gz) files
                                               |
                                               +-- .stats files
```

Key points:

- The `.bcl` or `.bcl.gz` files are located under the `Data/Intensities/BaseCalls/` directory
- The `config.xml` file under the `BaseCalls` directory is implicitly needed for demultiplexing and fastq conversion
- The `SampleSheet` file is only needed if the demultiplexing needs to be performed.

Fastq generation and demultiplexing

Multiplexed sequencing allows multiple samples to be run per lane. The samples are identified by index sequences (barcodes) that are attached to the template during sample preparation.

Originally bcl-to-fastq programs in Illumina's `CASAVA` software package could be used to perform both these steps, but were unable to handle the compressed bcf files produced by newer instruments. Illumina now provide a `bclToFastq` software package which only includes the components of `CASAVA` required for FASTQ conversion and which can also deal with compressed bcl files.

Note: Both `CASAVA` and `bclToFastq` provide the same programs for the conversion, and use the same protocol and input files. Within this documentation *bcl-to-fastq* is therefor used interchangably to refer to these programs.

The `configureBclToFastq.pl` script from bcl-to-fastq can be used to set up the bcl to fastq conversion, e.g.:

```
configureBclToFastq.pl \
        --input-dir <path_to_BaseCalls_dir> \
        --output-dir <path_to_output_dir> \
        [ --sample-sheet <path_to>/SampleSheet.csv ]
```

This will create the named output directory containing a Makefile which performs the actual conversion; to run, 'cd' to the output directory and then run `make`.

If the `--output-dir` option is omitted then it defaults to `<run_dir>/Unaligned/`. The sample sheet is only required for demultiplexing.

Other useful options:

- `--fastq-cluster-count <n>`: sets the maximum "cluster size" for the output fastq; this can result in multiple fastq output files. Use -1 to force all reads to be put into a single fastq.
- `--mismatches <n>`: number of mismatches allowed for each read; the default is zero (recommended for samples without multiplexing), 1 mismatch is recommended for multiplexed samples with tags of length 6 bases.

According to the CASAVA 1.8.2 documentation: *"FASTQ files contain only reads that passed filtering. If you want all reads in a FASTQ file, use the –with-failed-reads option."*

Note: Comprehensive notes on CASAVA options to use for bcl-to-fastq conversion for different demultiplexing scenarios can be found via https://gist.github.com/3125885

Sample sheets

Warning: Sample sheet files are generated by the software on the instrument. For older instruments these could be fed directly into the bcl-to-fastq conversion software; for newer instruments they are in "experimental manager" format, which needs to be converted to the older format - use the *prep_sample_sheet.py* utility to do this.

The sample sheets accepted by the bcl-to-fastq software are comma-separated files with the following fields on each line:

Field	Description
FCID	Flow cell ID
Lane	Positive integer, indicating the lane number (1-8)
SampleID	ID of the sample
SampleRef	The reference used for alignment for the sample
Index	Index sequences. Multiple index reads are separated by a hyphen (for example, ACCAGTAA-GGACATGA).
Description	Description of the sample
Control	Y indicates this lane is a control lane, N means sample
Recipe	Recipe used during sequencing
Operator	Name or ID of the operator
SampleProject	The project the sample belongs to

The `SampleID` field forms the base of the output fastq name (see below); the `SampleProject` field indicates which project directory the fastq file will be placed into.

It is advised to set both these fields to something descriptive e.g. SampleProject = "Control" and SampleName = "PhiX".

To remove a lane from the analysis remove references to it from the sample sheet file.

The bcl-to-fastq software will automatically use the samplesheet files in the instrument output directories unless overriden by a user-supplied samplesheet file.

The samplesheet can be edited using Excel or similar spreadsheet program, and manipulated using the *prep_sample_sheet.py* utility. The modified samplesheet file name can be supplied as an addition argument to the `bclToFastq.sh` script.

Output directory structure

Example output directory structure is:

```
Unaligned/
   |
   +-- Project_A/
   |          |
   |          +- Sample_A/
   |          |      |
   |          |   fastq.gz file(s)
   |          |
   |          +- Sample_B/
   |                 |
   |              fastq.gz file(s)
   |
   +-- Project_B/
              |
              +- Sample_C/
                     |
                  fastq.gz file(s)
```

In the absence of a sample sheet, one sample is assumed per lane and all samples belong to he same project.

Output fastq files

The general naming scheme for fastq output files is:

```
<sample_name>_<barcode_sequence>_L<lane>_R<read_number>_<set_number>.fastq.gz
```

e.g. `NA10931_ATCACG_L002_R1_001.fastq.gz`

For non-multiplexed runs, the sample name is the lane (e.g. `lane1` etc) and the barcode sequence is `NoIndex`

e.g. `lane1_NoIndex_L001_R1_001.fastq.gz`

The read number is either 1 or 2 (2's only appear for paired-end sequencing).

The quality scores in the output fastq files are Phred+33 (see http://en.wikipedia.org/wiki/FASTQ_format#Quality under the "Encoding" section).

Undetermined reads

When demultiplexing it is likely that the software will be unable to assign some of the reads to a specific sample. In this case the read is assigned to "undetermined" instead, and there will be an additional `Undetermined_indexes` "project" produced under the `Unaligned` directory.

2.2.2 FASTQ generation and analysis directory setup

Overview

This section outlines the protocol for generating FASTQ files from the raw bcl data and setting up per-project analysis directories using the scripts and utilities included in this package.

The basic procedure is:

1. Create top-level analysis directory
2. Generate FASTQ files
3. Populate analysis subdirectories for each project

Subsequently the QC pipeline should be run for each project.

Create top-level analysis directory

Create a top-level analysis directory where the FASTQs and per-project analysis directories will be created, for example:

```
mkdir /scratch/120919_SN7001250_0035_BC133VACXX_analysis
```

Note: Conventionally we name analysis directories by appending `_analysis` to the primary data directory name.

FASTQ generation

Within the top-level directory create a customised copy of the original `SampleSheet.csv` from the primary data directory. This is best done using the *prep_sample_sheet.py* utility, as it will automatically convert the original file to the correct format.

`prep_sample_sheet.py` can automatically address specific issues, for example:

`--fix-spaces`
: replaces spaces in sampleId and sampleProject fields with underscore characters

`--fix-duplicates`
: appends indices to sampleIds to make sampleId/sampleProject combinations unique

These two options together should automatically fix most problems with sample sheets, e.g.:

```
prep_sample_sheet.py \
    --fix-spaces --fix-duplicates \
    -o custom_samplesheet.csv \
    /mnt/data/120919_SN7001250_0035_BC133VACXX/SampleSheet.csv
```

It also has options to edit the sample sheet file fields: for example the `--set-id=...` and `--set-project=` options allow resetting of sampleId and sampleProject fields.

Note: `prep_sample_sheet.py` will only write a new sample sheet file if it thinks that the problems have been addressed; to override this use the `--ignore-warnings` option.

To generate FASTQS, run the *bclToFastq.sh* script in the top-level analysis directory, e.g.:

```
qsub -b y -cwd -V bclToFastq.sh \
    /mnt/data/120919_SN7001250_0035_BC133VACXX \
    Unaligned custom_samplesheet.csv
```

This automatically runs the `configureBlcToFastq.ps` and `make` steps (above) together and creates a new subdirectory called `Unaligned` with the FASTQS.

The general syntax for this step is:

```
bclToFastq.sh /path/to/ILLUMINA_RUN_DIR output_dir [ samplesheet.csv ]
```

Note: If bcl-to-fastq fails to generate the FASTQ files due to some problem with the input data then the *Troubleshooting bcl to FASTQ conversion* section below may help.

Populate analysis subdirectories

Use the *build_illumina_analysis_dirs.py* utility to create subdirectories for each project named in the input sample sheet file, and populate these with links to the FASTQ files generated in the previous step.

Use the `--list` option to see what projects and samples the program will use, e.g.:

```
build_illumina_analysis_dir.py --list \
   /scratch/120919_SN7001250_0035_BC133VACXX_analysis
```

which produces output of the form:

```
Project: AB (4 samples)
        AB1
                AB1_NoIndex_L002_R1_001.fastq.gz
        AB2
                AB2_NoIndex_L003_R1_001.fastq.gz
        AB3
                AB3_NoIndex_L004_R1_001.fastq.gz
        AB4
                AB4_NoIndex_L005_R1_001.fastq.gz
Project: Control (4 samples)
        PhiX1
                PhiX1_NoIndex_L001_R1_001.fastq.gz
        PhiX2
                PhiX2_NoIndex_L006_R1_001.fastq.gz
        PhiX3
                PhiX3_NoIndex_L007_R1_001.fastq.gz
        PhiX4
                PhiX4_NoIndex_L008_R1_001.fastq.gz
```

Use the `--expt=EXPT_TYPE` option to specify a library type for one or more projects, e.g.:

```
build_illumina_analysis_dir.py \
   --expt=AB:ChIP-seq \
   /mnt/analyses/120919_ILLUMINA-73D9FA_00008_FC_analysis
```

This creates new subdirectories for each project which contain symbolic links to the FASTQ files:

```
<YYMMDD>_<machinename>_<XXXXX>_FC_analysis/
        |
        +-- Unaligned/
        |     |
        |    ...
        |
        +-- <PI>_<library>/
        |     |
        |     +-- *.fastq.gz -> ../Unaligned/.../*.fastq.gz
        |
```

(continues on next page)

(continued from previous page)

```
        |
        +-- <PI>_<library>/
        |      |
        |      +-- *.fastq.gz -> ../Unaligned/.../*.fastq.gz
        |
       ...
```

`Unaligned` is the output from the `bclToFastq.sh` run (see the previous section), and will contain the fastq files. The fastq.gz files in these directories are symbolic links to the files in the `Unaligned` directory.

By default the FASTQ names are simplified versions of the original FASTQs; use the `--keep-names` to preserve the full names of the FASTQ files.

Merging replicates

Multiplexed runs can produce large numbers of replicates of each sample, with each replicate producing a single FASTQ file - so if there are 20 samples each with 8 replicates then this will produce 160 FASTQ files.

In this situation it can be more helpful to concatenate the replicates into single FASTQ files, and can be done automatically when creating the analysis subdirectories using the `--merge-replicates` option.

`--merge-replicates` doesn't require any additional input; it produces concatenated FASTQ files (rather than symbolic links) when creating the analysis subdirectory for each project, e.g.:

```
build_illumina_analysis_dir.py \
    --expt=AB:RNA-seq \
    --merge-replicates \
    /mnt/analyses/120919_SN7001250_0035_BC133VACXX_analysis
```

Note: Use the *verify_paired.py* utility to check that the order of reads in the merged files are correct.

2.2.3 Troubleshooting bcl to FASTQ conversion

Failure with error "sample-dir not valid: number of directories must match the number of barcodes"

This might be due to the presence of spaces in the `sampleID` and `sampleProjects` fields in the `sampleSheet.csv` file, which seems to confuse CASAVA.

The solution is to edit the sample sheet file to remove the spaces; this can be done automatically using the `--fix-spaces` option of the *prep_sample_sheet.py* program e.g.:

```
prep_sample_sheet.py --fix-spaces -o custom_SampleSheet.csv sampleSheet.csv
```

will create a copy of the original sample sheet file with any spaces replaced by underscores.

Failure with error "barcode XXXXXX for lane 1 has length Y: expected barcode lenth (including delimiters) is Z"

This can happen when attempting to demultiplex paired barcoded samples. The information that CASAVA needs should be read automatically from the `RunInfo.xml` file, but it appears that this doesn't always happen (or perhaps the information is not consistent with the `bcl` files e.g. because the sequencing run didn't complete properly).

To fix this use the `--use-bases-mask` option of `configureBclToFastq.pl` (or `bclToFastq.sh`) to tell CASAVA how to deal with each base. For example:

```
--use-bases-mask y101,I8,I8,y85
```

instructs the software to treat the first 101 bases as the first sequence, the next 8 as the first index (i.e. barcoded tag attached to the first sequence), the next 8 as the second index, and then the next 85 bases as the second sequence.

Note: See also this BioStars question about dealing with the CASAVA error: *"barcode CTTGTA for lane 1 has length X: expected barcode lenth is Y"* http://www.biostars.org/post/show/49599/casava-error-barcode-cttgta-for-lane-1-has-length-6-expected-barcode-lenth-is-7/#55718

2.3 Preparing SOLiD Data for analysis

2.3.1 Background

Structure of SOLiD run names

For multiplex fragment sequencing the run names will have the form:

```
<instrument-name>_<date-stamp>_FRAG_BC[_2]
```

(For example: `solid0123_20110315_FRAG_BC`).

The components are:

- `<instrument_name>`: name of the SOLiD instrument e.g. `solid0123`
- `<date-stamp>`: a date stamp in year-month-day format e.g. `20110315` is 15th March 2011
- `FRAG`: indicates a fragment library was used
- `BC`: indicates bar-coding was used (note that not all samples in the run might be bar-coded, even if this appears in the name)
- `2`: if this is present then it indicates the data came from flow cell 2; otherwise it's from flow cell 1.

For multiplex paired-end sequencing the run names have the form:

```
<instrument-name>_<date-stamp>_PE_BC
```

Here the `PE` part of the name indicates a paired-end run.

Note: If the run name contains `WFA` then it's a work-flow analysis and not final sequence data.

See also SOLiD 4 System Instrument Operation Quick Reference (PDF) for more information.

Navigating the SOLiD run data directories

Run definition file

Typically the top-level of a SOLiD run data directory should contain the run definition file which has information about the samples and libraries used in the run, including the names that were assigned when the run was set up. For example for a bar-coded sample this might look like:

```
version        userId  runType isMultiplexing  runName runDesc mask    protocol
v1.3   lab_user        FRAGMENT        TRUE    solid0127_20111013_FRAG_BC
↪ 1_spot_mask_sf  SOLiD4 Multiplex
primerSet      baseLength
BC     10
F3     50
sampleName     sampleDesc      spotAssignments primarySetting  library application
↪ secondaryAnalysis       multiplexingSeries      barcodes
DB_SB_JL_pool          1       default primary DB01    SingleTag       sacCer2 BC Kit
↪Module 1-96      "1"
DB_SB_JL_pool          1       default primary DB02    SingleTag       sacCer2 BC Kit
↪Module 1-96      "2"
...
DB_SB_JL_pool          1       default primary SB_DIMB_2       SingleTag       none
↪ BC Kit Module 1-96      "14"
DB_SB_JL_pool          1       default primary SB_DMTA SingleTag       none    BC Kit
↪Module 1-96      "15"
```

Essentially the run definition file consists of a three sections, each delimited by a header line. The last section (with the header line beginning `sampleName...`) has the information on each of the libraries, and can be used to locate the primary data files.

Primary data files (csfasta/qual) for multiplex fragment sequencing

Locating the primary data files within the SOLiD data directories can be quite tedious and confusing. For bar-coded samples the following heuristic can be used:

1. From the top-level of the SOLiD run directory (e.g. `solid0123_20111013_FRAG_BC`) move into the subdirectory with the sample name of interest (e.g. `DB_SB_JL_pool`, from the run definition file in the previous section).
2. Within the sample subdirectory, look for a directory called `results` (which will be a link to one of the other `results...` directories here). Move into the `results` directory.
3. Within the `results` subdirectory, look for a directory called `libraries` and move into this.
4. Within `libraries` you should see subdirectories named for each of the libraries associated with this sample, as they appear in the run definition file (e.g. `DB01`, `DB02`, ..., `SB_DIMB_2`, `SB_DMTA`). Move into the subdirectory for the library of interest.
5. Within the directory for a specific library, there should be one or more subdirectories with names of the form `primary.20111015000420127` (and possibly also `secondary.20111015000420127`). Check each of these subdirectories looking for the one which itself contains three subdirectories `reads`, `rejects` and `reports` (the others will only contain `reads` and `reports`). Move into this directory, and then into the `reads` subdirectory. This is the location of the primary data files (csfasta and qual files).

Typically this results in a path of the form:

```
solid0123_20111013_FRAG_BC/SAMPLE_NAME/results/libraries/LIBRARY_NAME/primary.
↪TIMESTAMP/reads/
```

As a further check, the primary data file names should include `F3` in the name.

Primary data files (csfasta/qual) for multiplex paired-end sequencing

In the case of paired-end sequencing the final data consists of primary data file pairs for both the `F3` and `F5` reads for each library.

Locating the `F3` and `F5` reads uses a similar heuristic to that described above for multiplex fragment sequencing:

1. From the top-level of the SOLiD run directory, move into the subdirectory for the sample name of interest (e.g. `DB_SB_JL_pool`).
2. Look for the `results` directory and move into it.
3. Look for the `libraries` directory and move into it.
4. Within `libraries` there are subdirectories for each of the libraries associated with this sample (e.g. `DB01`, `DB02`, ..., `SB_DIMB_2`, `SB_DMTA`) - move into the one for the library of interest.
5. Here there are one or more subdirectories with names of the form `primary.20111015000420127` etc. Check each of these subdirectories looking for those which contain three subdirectories `reads`, `rejects` and `reports` (not just `reads` and `reports`). There should be two `primary...` directories which match this criterion: in the `reads` directory of one there will be primary data files with `F5-BC` in the name, and in the other files with `F3`.

Automatic location of primary data using analyse_solid_run.py

The heuristics described above are also encoded in the *analyse_solid_run.py* program, which will identify and report the location of the primary data files when without any other arguments i.e.:

```
analyse_solid_run.py solid0123_20111101_FRAG_BC
```

This works for both multiplex fragment and multiplex paired-end sequencing.

2.3.2 Handling the SOLiD data

Copying SOLiD data from the sequencer

The script *rsync_solid_to_cluster.sh* can be used to copy data from the sequencing instrument in a semi-automatic fashion, by prompting the user at each point to ask if they wish to proceed with the next step.

Note: The script needs to be run on the sequencer.

It is recommended to run the script from within a `screen` session; it is started using the command:

```
rsync_solid_to_cluster.sh <solid_run> <user>@<host>:<datadir> [<email_address>]
```

This creates a copy of `<solid_run>` in `<data_dir>` on the remote system, for example:

```
rsync_solid_to_cluster.sh solid0123_20110827_FRAG_BC me@dataserver.foo.ac.uk:/mnt/
↪data me@foo.ac.uk
```

If there are multiple runs (i.e. flowcells) with the same base name then the script will detect the second run and also offer to transfer that as part of the procedure. The output of the actual `rsync` command is written to a time-stamped log file, and if an email address is given then the log will be mailed to that address.

The script performs the following actions, prompting for user confirmation at each stage:

1. Checks that the information provided by the user is correct
2. Does `rsync --dry-run` and presents the output for inspection by the user
3. Performs the rsync operation to copy the data (including removal of group write permissions on the remote copy) and emails a copy of the log file to the user
4. Checks that the local and remote file sizes match

See *rsync_solid_to_cluster.sh* for more information on the script.

Verifying the transferred data using MD5 checksums

Once the data has been transferred use the `--md5sum` option of *analyse_solid_run.py* to generate MD5 checksums for each of the primary data files, for example:

```
analyse_solid_run.py --md5sum solid 0123_20110827_FRAG_BC > chksums
```

Note: This step should be run on the remote system.

The `chksums` file generated above will consist of lines of the form:

```
229e9a651451c9e47f35e45792273185  solid0123_20111014_FRAG_BC/AB_CD_EF_pool/results.
↪F1B1/libraries/AB_A1M1/primary.201312345678901/reads/solid0123_20111014_FRAG_BC_AB_
↪CD_EF_pool_F3_AB_A1M1.csfasta
```

and can be fed into the Linux `md5sum` program on the SOLiD instrument to verify that the original files are the same, e.g.:

```
md5sum -c chksums
```

Note: This should be performed from the parent directory holding the runs on the SOLiD instrument.

Copying sequencing data to another location

Once the data has been transferred from the sequencer to the data store, it maybe be necessary to copy a subset of the data to another location.

In these cases the *analyse_solid_run.py* script can be used generate a template `rsync` script to perform the transfer, for example:

```
analyse_solid_run.py --rsync solid 0127_20110914_FRAG_BC > rsync.sh
```

The template `rsync.sh` script will contain something like:

```
#!/bin/sh
#
# Script command to rsync a subset of data to another location
# Edit the script to remove the exclusions on the data sets to be copied
rsync --dry-run -av -e ssh \
--exclude=AB_SEQ1 \
--exclude=AB_SEQ2 \
--exclude=AB_SEQ3 \
--exclude=AB_SEQ4 \
--exclude=AB_SEQ5 \
--exclude=AB_SEQ6 \
--exclude=AB_SEQ7 \
--exclude=AB_SEQ8 \
/mnt/data/solid0127_20120227_FRAG_BC user@remote.system:/destination/parent/dir
```

You must then edit the script:

- Remove the `--exclude` lines for each of the data sets you wish to transfer (yes, this is counter-intuative!);

- Edit `user@remote.system:/destination/parent/dir` and set to the user, system and directory you want to copy the data to.

To execute do:

```
./rsync.sh
```

which will perform a "dry run" - remove the `--dry-run` argument at the start of the generated script to perform the copy itself.

2.3.3 Preparing analysis directories

Overview

Once the SOLiD data has been transferred to the data store, the steps for creating the analysis directories:

0. Set up the environment to use the scripts
1. Check that the primary data
2. Create and populate the analysis directories
3. Run the automated QC pipeline
4. Generate XLS spreadsheet entry
5. Add the data and analysis directories to the logging file

Check the primary data

The *analyse_solid_run.py* script can be used to check and report on the SOLiD data. Running with the `--verify` option checks that the primary data is available for each sample and library:

```
analyse_solid_run.py --verify <solid_run_dir>
```

Use the `--report` option for a summary of the run:

```
analyse_solid_run.py --report <solid_run_dir>
```

to analyse the run data and get a report of the samples and libraries, e.g.:

```
$ analyse_solid_run.py solid0127_20110725_FRAG_BC
Flow Cell 1 (Quads)
===================
I.D.   : solid0127_20110725_FRAG_BC
Date   : 25/07/11
Samples: 4

Sample AB_E
-----------

Project E: E01-16 (16 libraries)
--------------------------------
Pattern: AB_E/E*
/mnt/data/solid0127_20110725_FRAG_BC/AB_E/.../solid0127_20110725_FRAG_BC_AB_E_F3_E01.
↪csfasta
/mnt/data/solid0127_20110725_FRAG_BC/AB_E/.../solid0127_20110725_FRAG_BC_AB_E_F3_QV_
↪E01.qual
```

(continues on next page)

(continued from previous page)

```
<...15 more file pairs snipped...>

Sample AB_F
-----------

Project F: F01-16 (16 libraries)
--------------------------------
Pattern: AB_F/F*
/mnt/data/solid0127_20110725_FRAG_BC/AB_F/.../solid0127_20110725_FRAG_BC_AB_F_F3_F01.
↪csfasta
/mnt/data/solid0127_20110725_FRAG_BC/AB_F/.../solid0127_20110725_FRAG_BC_AB_F_F3_QV_
↪F01.qual
<...15 more file pairs snipped...>

...
```

This reports details of the location of the primary data for each library (e.g. `E01`) within each sample (e.g. `AB_E`).

Create and populate analysis directories

To get a suggested layout command, run *analyse_solid_run.py* with the `--layout` option, e.g.:

```
analyse_solid_run.py --layout <solid_run_dir>
```

which produces output of the form e.g.:

```
#!/bin/sh
#
# Script commands to build analysis directory structure
#
./build_analysis_dir.py \
--link=relative \
--top-dir=/mnt/analyses/solid0127_20111013_FRAG_BC_analysis \
--name=AB --type=expt --source=AB_CD_EF_pool/AB0* \
--name=CD --type=expt --source=AB_CD_EF_pool/CD_* \
--name=EF --type=expt --source=AB_CD_EF_pool/EF_* \
/mnt/data/solid0127_20111013_FRAG_BC
#
./build_analysis_dir.py \
--link=relative \
--top-dir=/mnt/analyses/solid0127_20111013_FRAG_BC_2_analysis \
--name=UV --type=expt --source=UV_XY_pool/UV_* \
--name=XY --type=expt --source=UV_XY_pool/XY* \
/mnt/data/solid0127_20111013_FRAG_BC_2
```

This output can be redirected to a file e.g.:

```
analyse_solid_dir.py --layout /mnt/data/solid0127_20111013_FRAG_BC > layout.sh
```

and edited as appropriate (specifically: the `--type` arguments should be updated to the appropriate experimental method e.g. `--type=ChIP-seq`, `--type=RNA-seq` etc), before being executed from the command line i.e.:

```
sh layout.sh
```

The *build_analysis_dir.py* program creates the top level analysis directories, with subdirectories for each of the experiments (using a combination of the name and experiment type e.g. `AB_ChIP-seq`). Each subdirectory will contain

symbolic links to the primary data files.

Experiment types

The suggested experiment types are:

- *ChIP-seq*
- *RNA-seq*
- *RIP-seq*
- *reseq*
- *miRNA*

Naming schemes

By default the symbolic link names are "partial" versions of the full primary data file names. Add the `--naming-scheme=SCHEME` option to the layout script to explicitly choose a naming scheme:

Scheme	Template	Example
`partial`	`INSTRUMENT_TIMESTAMP_LIBRARY_QV.ext`	`solid0127_20110725_F01.csfasta`
`minimal`	`LIBRARY.ext`	`F01.csfasta`
`full`	Same as primary data file	`solid0127_20110725_FRAG_BC_AB_F_F3_F01.csfasta`

For the partial scheme, the qual file names always end with `_QV` (regardless of where the `QV` part appears in the original name).

For paired-end data, both the partial and minimal schemes append either `_F3` or `_F5` to the names as appropriate.

Specifying symbolic link types

The `--link` option allows you to specify whether links to primary data should be `relative` (recommended) or `absolute`. If it's not possible to create relative links then absolute links are created even if `relative` links were requested.

Use the `symlinks` utility on Linux to update absolute links to relative links if required.

2.3.4 Generate XLS spreadsheet entry

Running:

```
analyse_solid_run.py --spreadsheet=<output_spreadsheet> <solid_run_dir>
```

writes the data for the last run to a new spreadsheet, or appends it if the named spreadsheet already exists.

Note that if there are two directories for the SOLiD run then the script automatically detects the second one and writes the data for both.

2.4 QC Protocols

2.4.1 Overview

There are two over-arching QC scripts:

- **illumina_qc.sh**: runs the QC pipeline for fastq (or fastq.gz) file generated by an Illumina instrument (fastq_screen and FASTQC). This is described in more detail in *QC for Illumina sequencing data*.
- **solid_qc.sh**: runs the QC pipeline for csfasta and qual file pair (fragment mode) or pair of pairs (paired-end mode) generated by a SOLiD instrument (solid2fastq, fastq_screen, solid_preprocess_filter and qc_boxplotter). This is described in more detail in *QC for SOLiD sequencing data*.

Both scripts automatically run a series of checks and data preparation steps on the data prior to any real analysis taking place. Some of the pipeline components can also be run independently (e.g. *qc_boxplotter*, *fastq_screen.sh* etc) - see the information in the *QC Pipeline* command reference.

Many of the QC scripts read their settings from the `qc.setup` file, which tells them where to find some of the underlying software and data files. See *Create qc.setup* for how to set up this file.

Note: Generally the QC scripts won't overwrite outputs from a previous run; if you want to regenerate the outputs then you'll need to remove the previous outputs first.

Each script only runs on the data for a single sample, so there are two additional helper scripts that are used to process multiple samples and check the results:

- *run_qc_pipeline.py*: used to run a specific script on multiple sets of files, also performing various job management operations (such as submitting jobs to Grid Engine).
- *qcreporter.py*: check the outputs from the top-level SOLiD or Illumina QC scripts, and generate an HTML report.

A typical example of running `illumina_qc.sh` on all FASTQ files in a directory `DIR` might look like:

```
run_qc_pipeline.py --input=FORMAT illumina_qc.sh DIR
```

To verify that the QC has worked:

```
qcreporter.py --platform=illumina --format=FORMAT --verify DIR
```

and to generate the HTML QC report:

```
qcreporter.py --platform=illumina --format=FORMAT DIR
```

More specific examples are given in the following sections, and in the command reference for the utilities.

2.4.2 QC for Illumina sequencing data

The full QC pipeline for Illumina data (e.g. GA2x, MiSEQ, HiSEQ etc sequencer platforms) is encoded in the *illumina_qc.sh* script.

This can be run for a set of Illumina `fastq` or `fastq.gz` format files in a specific directory using the *run_qc_pipeline.py* utility. In its simplest form:

```
run_qc_pipeline.py --input=FORMAT illumina_qc.sh DIR
```

`FORMAT` can be either `fastq` or `fastqgz`; this will detect all matching files in the directory `DIR` and then use `qsub` to submit Grid Engine jobs to run the QC script on each file.

For each sample the `illumina_qc.sh` generates fastq_screen plots for model organisms, other organisms and rRNAs plus the report files from FASTQC.

If the input files are `fastq.gz` then it can also produce uncompressed versions of the files (specify the `--gunzip` option to turn on this behaviour).

2.4.3 QC for SOLiD sequencing data

The full QC pipeline for SOLiD data is encoded in the `solid_qc.sh` script. It runs in either "fragment mode", which takes a CSFASTA/QUAL file pair as input, or in "paired-end mode", which takes two CSFASTA/QUAL file pairs as input (the first should be the F3 pair and the second the corresponding F5 pair).

This can be run in either mode for a set of SOLiD data files in a specific directory using the `run_qc_pipeline.py` command. In its simplest form, for "fragment" (F3) data:

```
run_qc_pipeline.py --input=solid solid_qc.sh DIR
```

or for paired-end (F3/F5) data:

```
run_qc_pipeline.py --input=solid_paired_end solid_qc.sh DIR
```

In each case this will detect all matching file groups in the directory `DIR` and then use `qsub` to submit Grid Engine jobs to run the QC script on each group.

The pipeline consists of:

- `solid2fastq`: creates a FASTQ file from the input CSFASTA/QUAL file pair
- `fastq_screen`: checks the reads against 3 different screens (model organisms, "other" organisms and rRNA) to look for contaminants
- `solid_preprocess_filter`: runs the `SOLiD_prepreprocess_filter_v2.pl` program on the input CSFASTA/QUAL file pair to filter out "bad" reads, and reports the percentage filtered out (also produces a FASTQ and boxplot for the filtered data)
- `qc_boxplotter`: generates quality-score boxplots from the input QUAL file

The main outputs are the FASTQ file and a subdirectory `qc` which holds the screen and boxplot files.

(See the section above on "Illumina QC" for additional options available for `run_qc_pipeline.py`.)

To verify that the QC has worked, run the `qcreporter.py` command:

```
qcreporter.py --platform=solid --format=FORMAT --verify DIR
```

(where `FORMAT` is either `solid` or `solid_paired_end`), and to generate the HTML QC report:

```
qcreporter.py --platform=solid --format=FORMAT DIR
```

Outputs

SOLiD paired-end data

Say that the input files are `PB_F3.csfasta`, `PB_F3.qual` and `PB_F5.csfasta`, `PB_F5.qual`.

Stage	Files	Description	Comments
Quality filtering	`PB_F3_T_F3.csfasta`, `PB_F3_T_F3_QV.qual`	F3 data after quality filter	
	`PB_F5_T_F3.csfasta`, `PB_F5_T_F3_QV.qual`	F5 data after quality filter	Only has F5 reads: ignore the F3 part of "T_F3"
Merge unfiltered	`PB_paired.fastq`	All unfiltered F3 and F5 data in one fastq file	Used for fastq_screen
Merge F3 filtered	`PB_paired_F3_filt.fastq`	Filtered F3 reads with the matching F5 partner	"Lenient" filtering: only the quality of the F3 reads is considered
Merge all filtered	`PB_paired_F3_and_F5_filt.fastq`	Filtered F3 reads and filtered F5 reads	"Strict" filtering: pairs of reads are rejected on the quality of either of the F3 or F5 components
Split FASTQs	`PB_paired_F3_filt.F3.fastq`	F3 reads only from `PB_paired_F3_filt.fastq`	Data to use for mapping
	`PB_paired_F3_filt.F5.fastq`	F5 reads only from `PB_paired_F3_filt.fastq`	
	`PB_paired_F3_and_F5_filt.F3.fastq`	F3 reads only from `PB_paired_F3_and_F5_filt.fastq`	Data to use for mapping
	`PB_paired_F3_and_F5_filt.F5.fastq`	F5 reads only from `PB_paired_F3_and_F5_filt.fastq`	

For each sample the following output files will be produced by `solid_qc.sh`.

"Fragment" mode (default)

Say that the input SOLiD data file pair is *PB.csfasta* and *PB.qual*, then the following FASTQ files are produced:

- *PB.fastq*: all reads
- *PB_T_F3.csfasta* and *PB_T_F3_QV.qual*: primary data after quality filtering
- *PB_T_F3.fastq*: reads after quality filtering

Paired-end mode

Say that the input SOLiD data file pairs are *PB_F3.csfasta*, *PB_F3.qual* and *PB_F5.csfasta*, *PB_F5.qual*, then the following FASTQ files are produced:

Unfiltered data

Merging all the original unfiltered data into a single fastq gives:

- *PB_paired.fastq*: all unfiltered F3 and F5 data merged into a single fastq

- *PB_paired.F3.fastq*: unfiltered F3 data
- *PB_paired.F5.fastq*: unfiltered F5 data

Quality filtered data

Quality filtering on the primary data gives:

- *PB_F3_T_F3.csfasta* and *PB_F3_T_F3_QV.qual*: F3 data after quality filter
- *PB_F5_T_F3.csfasta* and *PB_F5_T_F3_QV.qual*: F5 data after quality filter

(Note that the files with *F5* in the name only have F5 reads - ignore the *F3* part of *T_F3*.)

"Lenient" filtering and merging the F3 filtered data with all F5 gives:

- *PB_paired_F3_filt.fastq*: filtered F3 reads with the matching F5 partner
- *PB_paired_F3_filt.F3.fastq*: just the F3 reads after filtering
- *PB_paired_F3_filt.F5.fastq*: just the matching F5 partners

(This is called "lenient" as only the quality of the F3 reads is considered.)

"Strict" filtering and merging gives:

- *PB_paired_F3_and_F5_filt.fastq*: filtered F3 reads and filtered F5 reads, with "unpartnered" reads removed
- *PB_paired_F3_and_F5_filt.F3.fastq*: just the F3 reads
- *PB_paired_F3_and_F5_filt.F5.fastq*: just the F5 reads

(This is called "strict" filtering as a pair of reads will be rejected on the quality of either of the F3 or F5 components.)

Filtering statistics

The filtering statistics output file name depends on the mode that the pipeline was run using:

- *SOLiD_preprocess_filter.stats*: for fragment mode
- *SOLiD_preprocess_filter_paired.stats*: for paired end mode

In each case the file summarises the number of reads before and after filtering and merging, and indicates the percentage that have been filtered out (with typical values being between 20-30%).

Contamination screens (*fastq_screen.sh*)

Contamination screen outputs are written to the *qc* directory:

- *PB_model_organisms_screen.**: screen against a selection of commonly used genomes
- *PB_other_organisms_screen.**: screen against a selection of less common genomes
- *PB_rRNA_screen.**: screen against a selection of rRNAs

For each there are *.txt* and *.png* files.

Boxplots (*qc_boxplotter.sh*)

Boxplots are written to the *qc* subdirectory:

- *PB.qual_seq-order_boxplot.**: plot using all reads (PDF, PNG and PS formats)
- *PB_T_F3_QV.qual_seq-order_boxplot.**: plot using just the quality filtered reads

2.5 General Utilities

2.5.1 Checking files and directories using MD5 sums

The `md5checker.py` utility provides a way of checking files and directories using MD5 sums; it can generate a set of MD5 sums for a file or the contents of a directory, and then use these to verify the contents of another file, directory or set of files.

Its basic functionality is very much like the standard `md5sum` Linux program (however note that `md5checker.py` should also work on Windows), but it can also compare two directories directly with MD5 sums, without the need for an intermediate checksum file. This function is intended to provide a straightforward way of running MD5 checks for example when copying analysis of data generated in a cluster scratch area to the archive area.

For example: say you have a directory in `$SCRATCH` called `my_work`, which holds the results of various analysis jobs that you've run on the cluster. At some point you decide to copy these results to the data area:

```
cp -a $SCRATCH/my_work /mnt/data/copy_of_my_work
```

Then you run an MD5 sum check on the copy by doing:

```
md5checker.py --diff $SCRATCH/my_work /mnt/data/copy_of_my_work
```

which by default will generate output of the form:

```
Recursively checking files in /scratch/my_work against copies in /mnt/data/copy_of_my_
↪work
important_data.sam: OK
important_data.bam: OK
...
Summary: 147 files checked, 147 okay 0 failed
```

(Note that this differencing mode only considers files that are in `my_work`, so if `copy_of_my_work` contains additional files then these won't be checked or reported.)

Run `md5checker.py -h` to see the other available options.

2.5.2 Checking symbolic links

Use the `symlink_checker.py` utility.

CHAPTER 3

Command Reference

3.1 Genome indexes and reference data utilities

Scripts for setting up genome indexes for various programs:

- *fetch_fasta.sh*: download and build FASTA file for pre-defined organisms
- *build_indexes.sh*: build all indexes from a FASTA file
- *bfast_build_indexes.sh*: build bfast color-space indexes
- *bowtie_build_indexes.sh*: build color- and base-space bowtie indexes
- *bowtie2_build_indexes.sh*: build indexes for bowtie2
- *srma_build_indexes.sh*: build indexes for srma
- *setup_genome_indexes.sh*: automatically and reproducibly set up genome indexes
- *build_rRNA_bowtie_indexes.sh*: create indexes and fastq_screen.conf for rRNA
- *make_seq_alignments.sh*: build sequence alignment (.nib) files from FASTA

3.1.1 fetch_fasta.sh

Reproducibly downloads and builds FASTA files for pre-defined organisms.

Usage:

```
fetch_fasta.sh <name>
```

`<name>` identifies a specific organism and build, for example 'hg18' or `mm9` (run without specifying a name to see a list of all the available organisms).

Outputs

Downloads and creates a FASTA file for the specified organism and puts this into a'fasta' subdirectory in the current working directory. When possible the script verifies the FASTA file by running an MD5 checksum it.

An `.info` file is also written which contains details about the FASTA file, such as source location and additional operations that were performed to unpack and construct the file, and the date and user who ran the script.

Adding new organisms

New organisms can be added to the script by creating additional `setup_<name>` functions for each, and defining the source and operations required to build it. For example:

```
function setup_hg18() {
    set_name    "Homo sapiens"
    set_species "Human"
    set_build   "HG18/NCBI36.1 March 2006"
    set_info    "Base chr. (1 to 22, X, Y), 'random' and chrM - unmasked"
    set_mirror  http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips
    set_archive chromFa.zip
    set_ext     fa
    set_md5sum  8cdfcaee2db09f2437e73d1db22fe681
    # Delete haplotypes
    add_processing_step "Delete haplotypes" "rm -f *hap*"
}
```

See the comments in the head of the script along with the existing `setup_...` functions for more specifics.

3.1.2 build_indexes.sh

Builds all indexes (bowtie, bowtie2, SRMA) within a standard directory structure from a FASTA file, by running the scripts for building the individual indexes.

Usage:

```
build_indexes.sh <fasta_file>
```

Outputs

Typically you would create a new directory for each organism, and then place the FASTA file in a `fasta` subdirectory e.g.:

```
hg18/
    fasta/
         hg18.fasta
```

Then invoke this script from within the top-level `hg18` directory e.g.:

```
build_indexes.sh fasta/hg18.fasta
```

resulting in:

```
hg18/
    fasta/
    bfast/
    bowtie/
```

with the indexes placed in the appropriate directories (see the individual scripts for more details).

3.1.3 bfast_build_indexes.sh

Builds the bfast color-space indexes from a reference FASTA file.

Usage:

```
bfast_build_indexes.sh [OPTIONS] <genome_fasta_file>
```

Run with `-h` option to print full usage information.

Options:

-d <depth>
: Specify depth-of-splitting used by Bfast (default 1)

-w <hash_width>
: Specify hash width used by Bfast (default 14)

--dry-run
: Print commands without executing them

-h
: Print usage information and defaults

Outputs

Index files are created in the directory the script was run in.

- `.bif` index files
- `.brg` index files for base- and color-space
- Symbolic link to the reference (input) FASTA file.

Warning: If `.brg` and/or `.bif` files already exist then bfast index may not run correctly. It's recommended to remove any old files before rerunning the build script.

3.1.4 bowtie_build_indexes.sh

Builds the bowtie color and/or nucleotide space indexes from the reference FASTA file.

Usage:

```
bowtie_build_indexes.sh OPTIONS <genome_fasta_file>
```

Options:

By default both color- and nucleotide space indexes are built; to only build one or the other use one of:

--nt
build nucleotide-space indexes

--cs
build colorspace indexes

Outputs

Index files are created in the directory the script was run in.

- Nucleotide indexes as `<genome_name>.*.ebwt`
- Color space indexes as `<genome_name>_c.*.ebwt`

3.1.5 bowtie2_build_indexes.sh

Builds the indexes for `bowtie2` (letter space only; `bowtie2` doesn't support colorspace) from the reference FASTA file.

Usage:

```
bowtie2_build_indexes.sh <genome_fasta_file>
```

Outputs

Index files are created in the directory the script was run in, with the names `<genome_name>.*.bt2`.

3.1.6 srma_build_indexes.sh

Creates the index files required by SRMA.

Note: By default the script expects the `CreateSequenceDictionary.jar` file to be in the `/usr/share/java/picard-tools` directory; if this is not the case then set the variable `PICARD_TOOLS_DIR` variable in your environment to point to the actual location.

For example for `bash`:

```
export PICARD_TOOLS_DIR=/path/to/my/picard-tools
```

Usage:

```
srma_build_indexes.sh <genome_fasta_file>
```

Outputs

Index files are created in the same directory as the reference FASTA file (which is where SRMA requires them to be); the script itself can be run from anywhere.

- `.fai` and `.dict` files required by SRMA.

3.1.7 index_indexes.sh

Utility for exploring/reporting on existing genome indexes within a directory hierarchy.

Usage:

```
index_indexes.sh <dir>
```

Outputs

Searches `<dir>` and its subdirectories recursively and prints a report of the genome index-specific files (fasta, info etc) it finds.

3.1.8 setup_genome_indexes.sh

Automatically and reproducibly set up genome indexes.

Usage:

```
setup_genome_indexes.sh
```

The `setup_genome_indexes.sh` script doesn't take any options, it runs through hard-coded lists of organisms for obtaining the sequence and creating bowtie, bfast and Picard/SRMA indexes, Galaxy `.loc files` and `fastq_screen .conf` files.

Outputs

The script outputs genome indexes based on the following directory structure for each organism:

```
pwd/
    organism/
            organism.info
            organism.chr.list
            bowtie/
                  ...bowtie indexes...
            bfast/
                  ...bfast indexes...
            fasta/
                  organism.fasta
                  ...picard/srma indexes...
```

It also creates:

- `fastq_screen` directory: containing specified `fastq_screen .conf` files
- Galaxy `.loc` files: for bowtie, bfast, picard, all_fasta and fastq_screen
- `genome_indexes.html` file: HTML file listing the available genome indexes

3.1.9 build_rRNA_bowtie_indexes.sh

Create bowtie indexes and `fastq_screen.conf` file for rRNA sequences.

Usage:

```
build_rRNA_bowtie_indexes.sh <rRNAs>.tar.gz
```

The `build_rRNA_bowtie_indexes.sh` script unpacks the supplied archive file `<rRNAs>.tar.gz` and copies the FASTA-formatted sequence files it contains, then generates bowtie indexes from these and produces a `fastq_screen.conf` file for them.

Inputs

The script expects the input `<rRNAs>.tar.gz` file to unpack into the following directory structure:

```
rRNAs/
     fasta/
          ... fasta files ...
```

Outputs

The script creates the following directory structure in the current directory:

```
pwd/
    rRNAs/
         bowtie/
               ...bowtie indexes...
         fasta/
               ...rRNA fasta files...
```

It also creates `fastq_screen_rRNAs.conf` in the `fastq_screen` subdirectory of the current directory.

3.1.10 make_seq_alignments.sh

Build sequence alignment (`.nib`) files from a FASTA file.

Warning: `faToNib` is no longer distributed with the UCSC tools and `.nib` format is now deprecated in favour of `.2bit`.

The procedure is:

- Split FASTA file into individual chromosomes (uses the *split_fasta.py* utility)
- For each resulting chromosome run the UCSC tool `faToNib` to generate a sequence alignment file
- Copy these to a specified destination directory

Usage:

```
make_seq_alignments.sh [--qsub=...] FASTA SEQ_DIR
```

Generates sequence alignment (`.nib`) files for each chromosome in `FASTA`, and copies them into the (pre-existing) directory `SEQ_DIR`.

Options:

`--qsub[=...]`
: Run operations via Grid Engine (otherwise run directly). Optionally also supply extra arguments using `--qsub="..."` e.g. name of a specific queue.

Inputs

FASTA file with all chromosome sequences.

Outputs

A set of sequence alignment (`.nib`) files in the specified output directory.

3.2 SOLiD data handling utilities

Utilities for transferring data from the SOLiD instrument to the cluster:

- *rsync_solid_to_cluster.sh*: perform transfer of primary data
- *log_seq_data.sh*: maintain logging file of transferred runs and analysis data
- *analyse_solid_run.py*: report on the primary data directories from SOLiD runs
- *build_analysis_dir.py*: construct analysis directories for experiments

3.2.1 rsync_solid_to_cluster.sh

Interactive script to semi-automate the transfer of data from the SOLiD instrument to a destination machine.

Usage:

```
rsync_solid_to_cluster.sh <local_dir> <user>@<remote_host>:<remote_dir> [<email_
↪address>]
```

`<local_dir>` is the directory to be copied; `<user>` is an account on the destination machine `<remote_host>` and `<remote_dir>` is the parent directory where a copy of `<local_dir>` will be made.

The script operates interactively and prompts the user at each step to confirm each action:

1. Check that the information is correct
2. Do `rsync --dry-run` and inspect the output
3. Perform the actual rsync operation (including removing group write permission from the remote copy)
4. Check that the local and remote file sizes match

If there is more than one run with the same base name then the script will detect the second run and offer to copy both in a single script run.

The output from each `rsync` is also captured in a timestamped log file. If an email address is supplied on the command line then copies of the logs will be mailed to this address.

3.2.2 log_seq_data.sh

Script to add entries for transferred SOLiD run or analysis directories to a logging file.

Usage:

```
log_seq_data.sh [-u|-d] <logging_file> <solid_run_dir> [<description>]
```

A new entry for the directory <solid_run_dir> will be added to <logging_file>, consisting of the path to the directory, a UNIX timestamp, and the optional description.

The path can be relative or absolute; relative paths are automatically converted to full paths.

If the logging file doesn't exist then it will be created. A new entry won't be created for any SOLiD run directory that is already in the logging file.

Options:

-u
: Updates an existing entry

-d
: Deletes an existing entry

Examples:

Log a primary data directory:

```
log_seq_data.sh /mnt/data/SEQ_DATA.log /mnt/data/solid0127_20110914_FRAG_BC "Primary␣
↪data"
```

Log an analysis directory (no description):

```
log_seq_data.sh /mnt/data/SEQ_DATA.log /mnt/data/solid0127_20110914_FRAG_BC_analysis
```

Update an entry to add a description:

```
log_seq_data.sh /mnt/data/SEQ_DATA.log -u /mnt/data/solid0127_20110914_FRAG_BC_
↪analysis \
    "Analysis directory"
```

Delete an entry:

```
log_seq_data.sh /mnt/data/SEQ_DATA.log -d /mnt/data/solid0127_20110914_FRAG_BC_
↪analysis
```

3.2.3 analyse_solid_run.py

Utility for performing various checks and operations on SOLiD run directories. If a single solid_run_dir is specified then analyse_solid_run.py automatically finds and operates on all associated directories from the same instrument and with the same timestamp.

Usage:

```
analyse_solid_run.py OPTIONS solid_run_dir [ solid_run_dir ... ]
```

Options:

--only
: only operate on the specified solid_run_dir, don't locate associated run directories

--report
: print a report of the SOLiD run

--report-paths
: in report mode, also print full paths to primary data files

--xls
write report to Excel spreadsheet

--verify
do verification checks on SOLiD run directories

--layout
generate script for laying out analysis directories

--rsync
generate script for rsyncing data

--copy=COPY_PATTERN
copy primary data files to pwd from specific library where names match COPY_PATTERN, which should be of the form '<sample>/<library>'

--gzip=GZIP_PATTERN
make gzipped copies of primary data files in pwd from specific libraries where names match GZIP_PATTERN, which should be of the form '<sample>/<library>'

--md5=MD5_PATTERN
calculate md5sums for primary data files from specific libraries where names match MD5_PATTERN, which should be of the form '<sample>/<library>'

--md5sum
calculate md5sums for all primary data files (equivalent to --md5=*/*)

--no-warnings
suppress warning messages

3.2.4 build_analysis_dir.py

Automatically construct analysis directories for experiments which contain links to the primary data files.

Usage:

```
build_analysis_dir.py [OPTIONS] EXPERIMENT [EXPERIMENT ...] <solid_run_dir>
```

General Options:

--dry-run
report the operations that would be performed

--debug
turn on debugging output

--top-dir=<dir>
create analysis directories as subdirs of <dir>; otherwise create them in cwd.

--naming-scheme=<scheme>
specify naming scheme for links to primary data (one of minimal - library names only, partial - includes instrument name, datestamp and library name (default) or full - same as source data file

--link=<type>
type of links to create to primary data files, either absolute (default) or relative

--run-pipeline=<script>
after creating analysis directories, run the specified <script> on SOLiD data file pairs in each

Options For Defining Experiments:

An "experiment" is defined by a group of options, which must be supplied in this order for each experiment specified on the command line:

```
--name=<name> [--type=<expt_type>] --source=<sample>/<library>
                                   [--source=... ]
```

<name> is an identifier (typically the user's initials) used for the analysis directory e.g. PB

<expt_type> is e.g. reseq, ChIP-seq, RNAseq, miRNA...

<sample>/<library> specify the names for primary data files e.g. PB_JB_pool/PB*

Example:

```
--name=PB --type=ChIP-seq --source=PB_JB_pool/PB*
```

Both <sample> and <library> can include a trailing wildcard character (i.e. *) to match multiple names. */* will match all primary data files. Multiple --sources can be declared for each experiment.

For each experiment defined on the command line, a subdirectory called <name>_<expt_type> (e.g. PB_ChIP-seq - if no <expt_type> was supplied then just the name is used) will be made containing links to each of the primary data files.

3.3 Illumina data handling utilities

Utilities for preparing data on the cluster from the Illumina instrument:

- *analyse_illumina_run.py*: reporting and manipulations of Illumina run data
- *auto_process_illumina.sh*: automatically process Illumina-based sequencing run
- *bclToFastq.sh*: generate FASTQ from BCL files
- *build_illumina_analysis_dirs.py*: create and populate per-project analysis dirs
- *demultiplex_undetermined_fastq.py*: demultiplex undetermined Illumina reads
- *prep_sample_sheet.py*: edit SampleSheet.csv before generating FASTQ
- *report_barcodes.py*: analyse barcode sequences from FASTQ files
- *rsync_seq_data.py*: copy sequencing data using rsync
- *verify_paired.py*: utility to check FASTQs form R1/R2 pair

3.3.1 analyse_illumina_run.py

Utility for performing various checks and operations on Illumina data.

Usage:

```
analyse_illumina_run.py OPTIONS illumina_data_dir
```

illumina_data_dir is the top-level directory containing the Unaligned directory with the fastq.gz files produced by the BCL-to-FASTQ conversion step.

Options:

--report
: report sample names and number of samples for each project

--summary
short report of samples (suitable for logging file)

-l, --list
list projects, samples and fastq files directories

--unaligned=UNALIGNED_DIR
specify an alternative name for the 'Unaligned' directory conatining the fastq.gz files

--copy=COPY_PATTERN
copy fastq.gz files matching COPY_PATTERN to current directory

--verify=SAMPLE_SHEET
check CASAVA outputs against those expected for SAMPLE_SHEET

--stats
Report statistics (read counts etc) for fastq files

3.3.2 auto_process_illumina.sh

Automatically process data from an Illumina-based sequencing platform

Usage:

```
auto_process_illumina.sh COMMAND [ PLATFORM DATA_DIR ]
```

COMMAND can be one of:

```
setup: prepares a new analysis directory. This step must be
       done first and requires that PLATFORM and DATA_DIR
       arguments are also supplied (these do not have to be
       specified for other commands).
       This creates an analysis directory in the current dir
       with a custom_SampleSheet.csv file; this should be
       examined and edited before running the subsequent
       steps.

make_fastqs: runs CASAVA to generate Fastq files from the
       raw bcls.

run_qc: runs the QC pipeline and generates reports.
```

The make_fastqs and run_qc commands must be executed from the analysis directory created by the setup command.

Standard protocol

The `auto_process_illumina.sh` script is intended to automate the major steps in generating FASTQ files from raw Illumina BCL data.

The standard protocol for using the automated script is:

1. Run the `setup` step to create a new analysis directory
2. Move into the analysis directory
3. **Check and if necessary edit the generated sample sheet, based on the predicted output projects and samples**
4. **Check and if necessary edit the bases mask setting in the ''DEFINE_RUN'' line in the ''processing.info'' file**

5. Run the `make_fastqs` step
6. Inspect the summary file which lists the generated FASTQ files along with their sizes and number of reads (and number of undetermined reads)
7. Run the `run_qc` step

The critical step is to check and edit the sample sheet, as this is used to determine which samples are assigned to which project. After editing the sample sheet it is a good idea to check the predicted outputs by running:

```
prep_sample_sheet.py SAMPLE_SHEET
```

and ensure that this is what was actually intended, before running the next steps.

To change the settings used by CASAVA's BCL to FASTQ conversion, it is also necessary to edit the `DEFINE_RUN` line in the `processing.info` file. This line typically looks like:

```
DEFINE_RUN  custom_SampleSheet.csv:Unaligned:y68,I7
```

The colon-delimited values are:

- Sample sheet name in the analysis directory (default: `custom_SampleSheet.csv`)
- The output directory where CASAVA will write the output data file (default: `Unaligned`)
- The bases mask that will be used by CASAVA (default will be determined automatically from the `RunInfo.xml` file in the source data directory)

Optionally a fourth colon-delimited value can be supplied:

- The number of allowed mismatches when demultiplexing (default will be determined from the bases mask value)

Multiple samplesheets

In some cases it might be necessary to split the BCL to FASTQ processing across multiple sample sheets.

In this case the protocol would be:

1. Run the `setup` step
2. Move into the analysis directory
3. **Create multiple sample sheets as required**
4. **Edit the 'processing.info' file to add 'DEFINE_RUN' for each sample sheet**
5. Run the `make_fastqs` step, which will automatically run a separate BCL to FASTQ conversion for each `DEFINE_RUN` line
6. For each BCL to FASTQ conversion, inspect the summary file which lists the generated FASTQ files along with their sizes and number of reads (and number of undetermined reads)
7. Run the `run_qc` step, which will automatically run a separate QC on the outputs of each BCL to FASTQ conversion

The previous section has more detail on the format and content of the `DEFINE_RUN` line. In the case of multiple `DEFINE_RUN` lines, it is advised to specify distinct output directories, e.g.:

```
DEFINE_RUN  pjbriggs_SampleSheet.csv:Unaligned_pjbriggs:y68,I7
```

3.3.3 bclToFastq.sh

Bcl to Fastq conversion wrapper script

Usage:

```
bclToFastq.sh <illumina_run_dir> <output_dir>
```

<illumina_run_dir> is the top-level Illumina data directory; Bcl files are expected to be in the Data/Intensities/BaseCalls subdirectory. <output_dir> is the top-level target directory for the output from the conversion process (including the generated fastq files).

The script runs configureBclToFastq.pl from CASAVA to set up conversion scripts, then runs make to perform the actual conversion. It requires that CASAVA is available on the system.

Options:

--nmismatches N
: set number of mismatches to allow; recommended values are 0 for samples without multiplexing, 1 for multiplexed samples with tags of length 6 or longer (see the CASAVA user guide for details of the --nmismatches option)

--use-bases-mask BASES_MASK
: specify a bases-mask string tell CASAVA how to use each cycle. The supplied value is passed directly to configureBcltoFastq.pl (see the CASAVA user guide for details of how –use-bases-mask works)

--nprocessors N
: set the number of processors to use (defaults to 1). This is passed to the -j option of the 'make' step after running configureBcltoFastq.pl (see the CASAVA user guide for details of how -j works)

3.3.4 build_illumina_analysis_dirs.py

Query/build per-project analysis directories for post-bcl-to-fastq data from Illumina GA2 sequencer.

Usage:

```
build_illumina_analysis_dir.py OPTIONS illumina_data_dir
```

Create per-project analysis directories for Illumina run. illumina_data_dir is the top-level directory containing the Unaligned directory with the fastq.gz files generated from the bcl files. For each Project_... directory build_illumina_analysis_dir.py makes a new subdirectory and populates with links to the fastq.gz files for each sample under that project.

Options:

--dry-run
: report operations that would be performed if creating the analysis directories but don't actually do them

--unaligned=UNALIGNED_DIR
: specify an alternative name for the Unaligned directory conatining the fastq.gz files

--expt=EXPT_TYPE
: specify experiment type (e.g. ChIP-seq) to append to the project name when creating analysis directories. The syntax for EXPT_TYPE is <project>:<type> e.g. --expt=NY:ChIP-seq will create directory NY_ChIP-seq. Use multiple --expt=... to set the types for different projects

--keep-names
: preserve the full names of the source fastq files when creating links

--merge-replicates
create merged fastq files for each set of replicates detected

3.3.5 demultiplex_undetermined_fastq.py

Demultiplex undetermined Illumina reads output from CASAVA.

Usage:

```
demultiplex_undetermined_fastq.py OPTIONS DIR
```

Reassign reads with undetermined index sequences. (i.e. barcodes). DIR is the name (including any leading path) of the 'Undetermined_indices' directory produced by CASAVA, which contains the FASTQ files with the undetermined reads from each lane.

Options:

--barcode=BARCODE_INFO
specify barcode sequence and corresponding sample name as BARCODE_INFO. The syntax is <name>:<barcode>:<lane> e.g. --barcode=PB1:ATTAGA:3

--samplesheet=SAMPLE_SHEET
specify SampleSheet.csv file to read barcodes, sample names and lane assignments from (as an alternative to --barcode).

3.3.6 prep_sample_sheet.py

Prepare sample sheet files for Illumina sequencers for input into CASAVA.

Usage:

```
prep_sample_sheet.py [OPTIONS] SampleSheet.csv
```

Utility to prepare SampleSheet files from Illumina sequencers. Can be used to view, validate and update or fix information such as sample IDs and project names before running BCL to FASTQ conversion.

Options:

-o SAMPLESHEET_OUT
output new sample sheet to SAMPLESHEET_OUT

-f FMT, **--format**=FMT
specify the format of the output sample sheet written by the -o option; can be either CASAVA or IEM (defaults to the format of the original file)

-v, --view
view contents of sample sheet

--fix-spaces
replace spaces in sample ID and project fields with underscores

--fix-duplicates
append unique indices to Sample IDs where original ID and project name combination are duplicated

--fix-empty-projects
create sample project names where these are blank in the original sample sheet

--set-id=SAMPLE_ID
update/set the values in the Sample ID field; SAMPLE_ID should be of the form <lanes>:<name>, where <lanes> is a single integer (e.g. 1), a set of integers (e.g. 1,3,...), a range (e.g. 1-3), or a combination (e.g. 1,3-5,7)

--set-project=SAMPLE_PROJECT
update/set values in the sample project field; SAMPLE_PROJECT should be of the form [<lanes>:]<name>, where the optional <lanes> part can be a single integer (e.g. 1), a set of integers (e.g. 1,3,...), a range (e.g. 1-3), or a combination (e.g. 1,3-5,7). If no lanes are specified then all samples will have their project set to <name>

--ignore-warnings
ignore warnings about spaces and duplicated sampleID/sampleProject combinations when writing new samplesheet.csv file

--include-lanes=LANES
specify a subset of lanes to include in the output sample sheet; LANES should be single integer (e.g. 1), a list of integers (e.g. 1,3,...), a range (e.g. 1-3) or a combination (e.g. 1,3-5,7). Default is to include all lanes

Deprecated options:

--truncate-barcodes=BARCODE_LEN
trim barcode sequences in sample sheet to number of bases specified by BARCODE_LEN. Default is to leave barcode sequences unmodified (deprecated; only works for CASAVA-style sample sheets)

--miseq
convert MiSEQ input sample sheet to CASAVA-compatible format (deprecated; conversion is performed specify -f/–format CASAVA to convert IEM sample sheet to older format)

Examples:

1. Read in the sample sheet file SampleSheet.csv, update the SampleProject and SampleID for lanes 1 and 8, and write the updated sample sheet to the file SampleSheet2.csv:

```
prep_sample_sheet.py -o SampleSheet2.csv --set-project=1,8:Control \
     --set-id=1:PhiX_10pM --set-id=8:PhiX_12pM SampleSheet.csv
```

2. Automatically fix spaces and duplicated sampleID/sampleProject combinations and write out to SampleSheet3.csv:

```
prep_sample_sheet.py --fix-spaces --fix-duplicates \
     -o SampleSheet3.csv SampleSheet.csv
```

3.3.7 report_barcodes.py

Examine barcode sequences from one or more Fastq files and report the most prevalent. Sequences will be pooled from all specified Fastqs before being analysed.

Usage:

```
report_barcodes.py FASTQ [FASTQ...]
```

Options:

--cutoff=CUTOFF
Minimum number of times a barcode sequence must appear to be reported (default is 1000000)

3.3.8 rsync_seq_data.py

Rsync sequencing data to archive location, inserting the correct 'year' and 'platform' subdirectories.

Usage:

```
rsync_seq_data.py [OPTIONS] DIR BASE_DIR
```

Wrapper to rsync sequencing data: DIR will be rsync'ed to a subdirectory of BASE_DIR constructed from the year and platform i.e. BASE_DIR/YEAR/PLATFORM/. YEAR will be the current year (over-ride using the –year option), PLATFORM will be inferred from the DIR name (over-ride using the –platform option). The output from rsync is written to a file rsync.DIR.log.

Options:

--platform=PLATFORM
: explicitly specify the sequencer type

--year=YEAR
: explicitly specify the year (otherwise current year is assumed)

--dry-run
: run rsync with `--dry-run` option

--chmod=CHMOD
: change file permissions using `--chmod` option of rsync (e.g 'u-w,g-w,o-w')

--exclude=EXCLUDE_PATTERN
: specify a pattern which will exclude any matching files or directories from the rsync

--mirror
: mirror the source directory at the destination (update files that have changed and remove any that have been deleted i.e. rsync –delete-after)

--no-log
: write rsync output directly stdout, don't create a log file

3.3.9 verify_paired.py

Utility to verify that two fastq files form an R1/R2 pair.

Usage:

```
verify_paired.py OPTIONS R1.fastq R2.fastq
```

Check that read headers for R1 and R2 fastq files are in agreement, and that the files form an R1/2 pair.

3.4 General NGS utilities

General NGS scripts that are used for both ChIP-seq and RNA-seq.

- *explain_sam_flag.sh*: decodes bit-wise flag from SAM file
- *extract_reads.py*: write out subsets of reads from input data files
- *fastq_edit.py*: edit FASTQ files and data
- *fastq_sniffer.py*: "sniff" FASTQ file to determine quality encoding

- *SamStats*: counts uniquely map reads per chromosome/contig
- *splitBarcodes.pl*: separate multiple barcodes in SOLiD data
- *remove_mispairs.pl*: remove "singleton" reads from paired end fastq
- *remove_mispairs.pl*: remove "singleton" reads from paired end fastq
- *reorder_fasta.py*: reorder chromosomes in FASTA file in karyotypic order
- *sam2soap.py*: convert from SAM file to SOAP format
- *separate_paired_fastq.pl*: separate F3 and F5 reads from fastq
- *split_fasta.py*: extract individual chromosome sequences from fasta file
- *split_fastq* : split fastq file by lane
- *trim_fastq.pl*: trim down sequences in fastq file from 5' end
- *uncompress_fastqgz.sh*: create ungzipped version of a compressed FASTQ file

3.4.1 explain_sam_flag.sh

Convert a decimal bitwise SAM flag value to binary representation and interpret each bit.

3.4.2 extract_reads.py

Usage:

```
extract_reads.py OPTIONS infile [infile ...]
```

Extract subsets of reads from each of the supplied files according to specified criteria (e.g. random, matching a pattern etc). Input files can be any mixture of FASTQ (`.fastq`, `.fq`), CSFASTA (`.csfasta`) and QUAL (`.qual`).

Output file names will be the input file names with `.subset` appended.

Options:

-m PATTERN, **--match**=PATTERN
: Extract records that match Python regular expression `PATTERN`

..cmdoption:: -n N

: Extract `N` random records from the input file(s) (default 500). If multiple input files are specified, the same subsets will be extracted for each.

3.4.3 fastq_edit.py

Usage:

```
fastq_edit.py [options] <fastq_file>
```

Perform various operations on FASTQ file.

Options:

--stats
: Generate basic stats for input FASTQ

--instrument-name=INSTRUMENT_NAME
Update the `instrument name` in the sequence identifier part of each read record and write updated FASTQ file to stdout

3.4.4 fastq_sniffer.py

Usage:

```
fastq_sniffer.py <fastq_file>
```

"Sniff" FASTQ file to try and determine likely format and quality encoding.

Attempts to identify FASTQ format and quality encoding, and suggests likely datatype for import into Galaxy.

Use the `--subset` option to only use a subset of reads from the file for the type determination (using a smaller set speeds up the process at the risk of not being able to accuracy determine the encoding convention).

See http://en.wikipedia.org/wiki/FASTQ_format for information on the different quality encoding standards used in different FASTQ formats.

Options:

--subset=N_SUBSET
try to determine encoding from a subset of consisting of the first `N_SUBSET` reads. (Quicker than using all reads but may not be accurate if subset is not representative of the file as a whole.)

3.4.5 SamStats

Counts how many reads are uniquely mapped onto each chromosome or contig in a SAM file. To run:

```
java -classpath <dir_with_SamStats.class> SamStats <sam_file>
```

or (if using a Jar file):

```
java -cp /path/to/SamStats.jar SamStats <sam_file>
```

(To compile into a jar, do `jar cf SamStats.jar SamStats.class`)

Output is a text file `SamStats_maponly_<sam_file>.stats`

3.4.6 splitBarcodes.pl

Split csfasta and qual files containing multiple barcodes into separate sets.

Usage:

```
./splitBarcodes.pl <barcode.csfasta> <read.csfasta> <read.qual>
```

Expects BC.csfasta, F3.csfasta and F3.qual files containing multiple barcodes. Currently set up for 'BC Kit Module 1-16'.

Note that this utility also requires *BioPerl*.

3.4.7 remove_mispairs.pl

Look through fastq file from solid2fastq that has interleaved paired end reads and remove singletons (missing partner)

Usage:

```
remove_mispairs.pl <interleaved FASTQ>
```

Outputs:

- <FASTQ>.paired: copy of input fastq with all singleton reads removed
- <FASTQ>.single.header: list of headers for all reads that were removed as singletons
- <FASTQ>.pair.header: list of headers for all reads there were kept as part of a pair

3.4.8 remove_mispairs.py

Python implementation of remove_mispairs.pl which can also remove singletons for paired end fastq data file where the reads are not interleaved.

3.4.9 reorder_fasta.py

Reorder the chromosome records in a FASTA file into karyotypic order.

Usage:

```
reorder_fasta.py INFILE.fa
```

Reorders the chromosome records from a FASTA file into 'kayrotypic' order, e.g.:

```
chr1
chr2
...
chr10
chr11
```

The output FASTA file will be called INFILE.karyotypic.fa.

3.4.10 sam2soap.py

Convert a SAM file into SOAP format.

Usage:

```
sam2soap.py OPTIONS [ SAMFILE ]
```

Convert SAM file to SOAP format - reads from stdin (or SAMFILE, if specified), and writes output to stdout unless -o option is specified.

Options:

-o SOAPFILE
: Output SOAP file name

3.4.11 separate_paired_fastq.pl

Takes a fastq file with paired F3 and F5 reads and separate into a file for each.

Usage:

```
separate_paired_fastq.pl <interleaved FASTQ>
```

3.4.12 split_fasta.py

Extract individual chromosome sequences from a fasta file.

Usage:

```
split_fasta.py fasta_file
```

Split input FASTA file with multiple sequences into multiple files each containing sequences for a single chromosome.

For each chromosome CHROM found in the input Fasta file (delimited by a line `>CHROM`), outputs a file called `CHROM.fa` in the current directory containing just the sequence for that chromosome.

3.4.13 split_fastq

Splits a Fastq file by lane.

Usage:

```
split_fastq.py [-h] [-l LANES] FASTQ
```

Split input Fastq file into multiple output Fastqs where each output only contains reads from a single lane.

Options:

-l LANES, **--lanes** LANES
: lanes to extract: can be a single integer, a comma- separated list (e.g. 1,3), a range (e.g. 5-7) or a combination (e.g. 1,3,5-7). Default is to extract all lanes in the Fastq

3.4.14 trim_fastq.pl

Takes a fastq file and keeps the first (5') bases of the sequences specified by the user.

Usage:

```
trim_fastq.pl <single end FASTQ> <desired length>
```

3.4.15 uncompress_fastqgz.sh

Create uncompressed copies of fastq.gz file (if input is fastq.gz).

Usage:

```
uncompress_fastqgz.sh <fastq>
```

`<fastq>` can be either fastq or fastq.gz file.

The original file will not be removed or altered.

3.5 ChIP-seq specific utilities

Scripts and tools for ChIP-seq specific tasks.

- *calc_coverage_stats.pl*: stats from a coverage file
- *convertFastq2Fasta.pl*: convert consensus fastq to fasta format
- *CreateChIPalignFileFromBed.pl*: convert csfasta->BED for GLITR
- *getRandomTags_index.pl, getRandomTags_index_fastq.pl*: extract random subsets of reads
- *make_macs2_xls.py*: convert a MACS output file into an Excel spreadsheet
- *mean_coverage.pl*: mean depth of read coverage from BAM file
- *run_DESeq.R*

3.5.1 calc_coverage_stats.pl

Get stats for coverage using a coverage file from `genomeCoverageBed -d`

Format of the input is:

```
chr position count
```

Outputs mean and median for all positions including 0 count positions

NB requires perl `Statistics::Descriptive` module

3.5.2 convertFastq2Fasta.pl

Convert fastq formatted consensus from `samtools pileup` to fasta

Note: Note that this will be redundant as `mpileup` (the successor to `pileup`) has its own way of doing this. However it may be required for legacy projects.

Usage:

```
perl ~/ChIP_seq/convertFastq2Fasta.pl in.pileup.fq > out.fa
```

3.5.3 CreateChIPalignFileFromBed.pl

Convert csfasta->BED format file to ChIPalign format for GLITR peak caller.

Usage:

```
CreateChIPalignFileFromBed.pl in.bed out.align
```

3.5.4 getRandomTags_index.pl, getRandomTags_index_fastq.pl

Extract random subset of records from fasta and fastq sequence files.

getRandomTags_index.pl

Extract `N` random records from ChIP align fasta files (2-line records):

Usage:

```
getRandomTags_index.pl in.fasta N out.fasta
```

getRandomTags_index_fastq.pl

Extract `N` random records from fastq file (4-line records):

Usage:

```
getRandomTags_index_fastq.pl in.fastq N out.fastq
```

3.5.5 make_macs2_xls.py

Convert a MACS2 tab-delimited output file into an Excel (XLSX) spreadsheet.

Usage:

```
make_macs2_xls.py OPTIONS <macs_output_file>.xls [<xlsx_output_file>]
```

Options:

```
-f XLS_FORMAT, --format=XLS_FORMAT
                      specify the output Excel spreadsheet format; must be
                      one of 'xlsx' or 'xls' (default is 'xlsx')
-b, --bed             write an additional TSV file with chrom,
                      abs_summit+100 and abs_summit-100 data as the columns.
                      (NB only possible for MACS2 run without --broad)
```

If the `xlsx_output_file` isn't specified then it defaults to `XLS_<macs_output_file>.xlsx`.

Note: To process output from MACS 1.4.2 and earlier use `make_macs_xls.py`; this version only supports `.xls` output and doesn't provide either of the `-f` or `-b` options.

3.5.6 mean_coverage.pl

Mean depth of read coverage: calculates the average coverage of all the captured bases in a bam file and presents as a single number.

Originally posted by Michael James Clark on Biostar: http://biostar.stackexchange.com/questions/5181/tools-to-calculate-average-coverage-for-a-bam-file

Usage:

```
/path/to/samtools pileup in.bam | awk '{print $4}' | perl mean_coverage.pl
```

It can also be used for genomic regions:

```
/path/to/samtools view -b in.bam <genomic region> | /path/to/samtools pileup - | awk '
↪{print $4}' | perl mean_coverage.pl
```

Note that this assumes every base is covered at least once (because `samtools pileup` doesn't report bases with zero coverage).

3.5.7 run_DESeq.R

Usage:

```
runDESeq.R [input file] [generic figure label] [output file]
```

Run DESeq in R using a tab delimited file [input file] that has a column of `chr_start_end` called 'regions', and four columns of read counts for:

```
timeA_rep1 timeA_rep2 timeB_rep1 timeB_rep2
```

('conds' order hard-wired).

A [generic figure label] adds specificity to the output diagrams (hard-wired). The final [output file] is created.

3.6 RNA-seq specific utilities

Scripts and tools for RNA-seq specific tasks.

- *bowtie_mapping_stats.py*: summarise statistics from bowtie output in spreadsheet
- *GFFedit*: swap gene names in GFF file to gene ID
- *qc_bash_script.sh*: generalised QC pipeline for RNA-seq
- *Split*: filter reads from bowtie mapping against two genomes

3.6.1 bowtie_mapping_stats.py

Extract mapping statistics for each sample referenced in the input bowtie log files and summarise the data in an XLS spreadsheet. Handles output from both Bowtie and Bowtie2.

Usage:

```
bowtie_mapping_stats.py [options] bowtie_log_file [ bowtie_log_file ... ]
```

By default the output file is called `mapping_summary.xls`; use the `-o` option to specify the spreadsheet name explicitly.

Options:

`-o` `xls_file`
: specify name of the output XLS file (otherwise defaults to `mapping_summary.xls`).

`-t`
: write data to tab-delimited file in addition to the XLS file. The tab file will have the same name as the XLS file, with the extension replaced by `.txt`

Input bowtie log file

The program expects the input log file to consist of multiple blocks of text of the form:

```
...
<SAMPLE_NAME>
Time loading reference: 00:00:01
Time loading forward index: 00:00:00
Time loading mirror index: 00:00:02
Seeded quality full-index search: 00:10:20
# reads processed: 39808407
# reads with at least one reported alignment: 2737588 (6.88%)
# reads that failed to align: 33721722 (84.71%)
# reads with alignments suppressed due to -m: 3349097 (8.41%)
Reported 2737588 alignments to 1 output stream(s)
Time searching: 00:10:27
Overall time: 00:10:27
...
```

The sample name will be extracted along with the numbers of reads processed, with at least one reported alignment, that failed to align, and with alignments suppressed and tabulated in the output spreadsheet.

3.6.2 GFFedit

Takes a GFF file and edits it, changing gene names in the input file to the geneID (if they are not of the DDB_G0... format), and outputs the edited GFF file.

Compilation:

```
javac GFFedit.java
jar cf GFFedit.jar GFFedit.class
```

Usage:

```
java -cp /path/to/GFFedit.jar GFFedit <myfile>.gff
```

Arguments:

myfile.gff
: input GFF file

Output:

GFFedit_<myfile>.gff
: edited version of input file

3.6.3 qc_bash_script.sh

Generalised QC pipeline for RNA-seq: runs bowtie, fastq_screen and qc_boxplotter on SOLiD data.

Usage:

```
qc_bash_script.sh <analysis_dir> <sample_name> <csfasta> <qual> <bowtie_genome_index>
```

Arguments:

analysis_dir
: directory to write the outputs to

sample_name
name of the sample

csfasta
input csfasta file

qual
input qual file

bowtie_genome_index
full path to bowtie genome index

Outputs:

Creates a `qc` subdirectory in `analysis_dir` which contains the fastq_screen and boxplotter output files.

3.6.4 Split

Takes in two SAM files from bowtie where the same sample has been mapped to two genomes ("genomeS" and "genomeB"), and filters the reads to isolate those which map only to genomeS, only to genomeB, and to both genomes (see "Output", below).

Compilation:

```
javac Split.java
jar cf Split.jar Split.class
```

Usage:

```
java -cp /path/to/Split.jar Split <map_to_genomeS>.sam <map_to_genomeB>.sam
```

Arguments:

map_to_genomeS.sam
SAM file from Bowtie with reads mapped to genomeS

map_to_genomeB.sam
SAM file from Bowtie with reads mapped to genomeB

Outputs 4 SAM files:

1. Reads that map to genomeS only
2. Reads that map to genomeB only
3. Reads that map to genomeS and genomeB keeping the genomeS genome coordinates
4. Reads that map to genomeS and genomeB keeping the genomeB genome coordinates

3.7 QC utilities

Scripts for running general QC and data preparation on SOLiD and Illumina sequencing data prior to library-specific analyses.

- *run_qc_pipeline.py*
- *illumina_qc.sh*
- *qcreporter.py*
- *fastq_screen.sh*

- *qc_boxplotter*: generate QC boxplot from SOLiD qual file
- *boxplotps2png.sh*: make PNGs of PS plots from *qc_boxplotter*
- *solid_preprocess_filter.sh*
- *solid_preprocess_truncation_filter.sh*
- *fastq_strand.py*

3.7.1 run_qc_pipeline.py

Overview

The pipeline runner program `run_qc_pipeline.py` is designed to automate running a specified script for all available datasets in one or more directories.

Datasets are assembled in an automated fashion by examining and grouping files in a given directory based on their file names and file extensions. The specified script is then run for each of the datasets, using the specified job runner to handle execution and monitoring for each run. The master pipeline runner performs overall monitoring, basic scheduling and reporting of jobs.

- The `--input` and `--regex` options control the assembly of datasets
- The `--runner` option controls which job runner is used
- The `--limit` and `--email` options control scheduling and reporting

See below for more information on these options.

Usage and options

Usage:

```
run_qc_pipeline.py [options] SCRIPT DIR [ DIR ...]
```

Execute SCRIPT on data in each directory DIR. By default the `SCRIPT` is executed on each CSFASTA/QUAL file pair found in `DIR`, as `SCRIPT CSFASTA QUAL`. Use the –input option to run `SCRIPT` on different types of data (e.g. FASTQ files). `SCRIPT` can be a quoted string to include command line options (e.g. `'run_solid2fastq.sh --gzip'`).

Basic Options:

`--limit``=MAX_CONCURRENT_JOBS`
: queue no more than `MAX_CONCURRENT_JOBS` at one time (default 4)

`--queue``=GE_QUEUE`
: explicitly specify Grid Engine queue to use

`--input``=INPUT_TYPE`
: specify type of data to use as input for the script. `INPUT_TYPE` can be one of: `solid` (CSFASTA/QUAL file pair, default), `solid_paired_end` (CSFASTA/QUAL_F3 and CSFASTA/QUAL_F5 quartet), `fastq` (FASTQ file), `fastqgz` (gzipped FASTQ file)

`--email``=EMAIL_ADDR`
: send email to `EMAIL_ADDR` when each stage of the pipeline is complete

Advanced Options:

`--regexp``=PATTERN`
: regular expression to match input files against

--test=MAX_TOTAL_JOBS
submit no more than MAX_TOTAL_JOBS (otherwise submit all jobs)

--runner=RUNNER
specify how jobs are executed: ge = Grid Engine, drmma = Grid Engine via DRMAA interface, simple = use local system. Default is ge

--debug
print debugging output

Recipes and examples

- Run the full SOLiD QC pipeline on a set of directories:

```
run_qc_pipeline.py solid_qc.sh <dir1> <dir2> ...
```

- Run the SOLiD QC pipeline on paired-end data:

```
run_qc_pipeline.py --input=solid_paired_end solid_qc.sh <dir1> <dir2> ...
```

- Run the Illumina QC pipeline on fastq.gz files in a set of directories:

```
run_qc_pipeline.py --input=fastqgz illumina_qc.sh <dir1> <dir2> ...
```

- Generate gzipped fastq files only in a set of directories:

```
run_qc_pipeline.py "run_solid2fastq.sh --gzip" <dir1> <dir2> ...
```

- Run the fastq_screen steps only on a set of directories:

```
run_qc_pipeline.py --input=fastq fastq_screen.sh <dir1> <dir2> ...
```

- Run the SOLiD preprocess filter steps only on a set of directories:

```
run_qc_pipeline.py solid_preprocess_filter.sh <dir1> <dir2> ...
```

- To get an email notification on completion of the pipeline:

```
run_qc_pipeline.py --email=foo@bar.com ...
```

Hints and tips

Note: To run without using Grid Engine submission, specify `--runner=simple`

This creates a `qc` subdirectory in `DIR` which contains the output QC products from `FastQC` and `fastq_screen`.

Useful additional options for `run_qc_pipeline.py` include:

Option	Function
`--limit=N`	Specify maximum number of jobs that will be submitted at one time (default is 4)
`--log-dir=DIR`	Specify a directory to write log files to
`--ge_args=ARGS`	Specify additional arguments to use with `qsub`, for example: `--ge_args="-j y -l short"`
`--regexp=PATTERN`	Specify a regular expression pattern; only filenames that match the pattern will have the QC script run on them

Note: It is recommended to run `run_qc_pipeline.py` within a Linux `screen` session.

3.7.2 illumina_qc.sh

`illumina_qc.sh` implements a basic QC script for a single input Fastq from Illumina sequencer platfroms; specifically:

- Check for contaminations against a panel of genome indexes using `fastq_screen` (via the *fastq_screen.sh* script)
- Generate QC metrics using `fastQC`
- (optionally) Create uncompressed copies of Fastq file

The input can be a compressed (gzipped) or uncompressed Fastq.

Usage:

```
illumina_qc.sh <fastq[.gz]> [options]
```

Options:

```
--ungzip-fastqs
              create uncompressed versions of
              fastq files, if gzipped copies
              exist
--no-ungzip   don't create uncompressed fastqs
              (ignored, this is the default)
--threads N   number of threads (i.e. cores)
              available to the script (default
              is N=1)
--subset N    number of reads to use in
              fastq_screen (default N=100000,
              N=0 to use all reads)
--no-screens  don't run fastq_screen
--qc_dir DIR  output QC to DIR (default 'qc')
```

3.7.3 qcreporter.py

Overview

`qcreporter.py` generates HTML reports for QC. It can be run on the outputs from either `solid_qc.sh` or `illumina_qc.sh` scripts and will try to determine the platform and run type automatically.

In some cases this automatic detection may fail, in which case the `--platform` and `--format` options can be used to explicit speciy the platform type and/or the type of input files that are expected; see the section on "Reporting recipes" below.

Usage and options

Usage:

```
qcreporter.py [options] DIR [ DIR ...]
```

Generate QC report for each directory `DIR` which contains the outputs from a QC script (either SOLiD or Illumina). Creates a `qc_report.<run>.<name>.html` file in `DIR` plus an archive `qc_report.<run>.<name>.zip` which contains the HTML plus all the necessary files for unpacking and viewing elsewhere.

Options:

`--platform`=`PLATFORM`
: explicitly set the type of sequencing platform (`solid`, `illumina`)

`--format`=`DATA_FORMAT`
: explicitly set the format of files (`solid`, `solid_paired_end`, `fastq`, `fastqgz`)

`--qc_dir`=`QC_DIR`
: specify a different name for the QC results subdirectory (default is `qc`)

`--verify`
: don't generate report, just verify the QC outputs

`--regexp`=`PATTERN`
: select subset of files which match regular expression `PATTERN`

Reporting recipes

The table below indicates the situations in which the reporter should work automatically, and which options to use in cases when it doesn't:

Platform	Data type	QC mode	Autodetect?
SOLiD4	Fragment	Fragment	Yes
SOLiD4	Paired-end	Fragment	Yes
SOLiD4	Paired-end	Paired-end	Yes
GA2x	Fastq.gz	n/a	Yes
GA2x	Fastq	n/a	No: use –format=fastq
HiSEQ/MiSEQ	Fastq.gz	n/a	No: use –platform=illumina
HiSEQ/MiSEQ	Fastq	n/a	**No: use –platform=illumina** –format=fastq

3.7.4 fastq_screen.sh

The fastq_screen part of the QC pipeline is implemented as a shell script `fastq_screen.sh` which can be run independently of the main qc.sh script. It takes a single FASTQ file as input, e.g:

```
fastq_screen.sh sample.fastq
```

This runs the `fastq_screen` program using three sets of genome indexes: common "model" organisms (e.g. human, mouse, rat, fly etc), "other" organisms (e.g. dictystelium), and a set of rRNA indexes.

The script gets its configuration from the following environment variables:

Variable	Function
`FASTQ_SCREEN_CONF_DIR`	Location of fastq_screen configuration files
`FASTQ_SCREEN_CONF_EXT`	Base filename extensions for letterspace fastq_screen configuration files (e.g. if conf file is `fastq_screen_model_organisms_nt.conf` for letterspace then the extension is `_nt`
`FASTQC_CONTAMINANTS_FILE`	Custom contaminants file for fastQC

These can be set in the `qc.setup` file, where the script will read the values from unless over-ridden by the environment.

The three sets of genome indexes are represented by three `fastq_screen` configuration files which should be present in the `FASTQ_SCREEN_CONF_DIR` directory, with the following naming conventions:

- Model organisms: `fastq_screen_model_organisms[EXT].conf`
- Other organisms: `fastq_screen_other_organisms[EXT].conf`
- rRNA: `fastq_screen_rRNA[EXT].conf`

The outputs are written to a `qc` subdirectory of the working directory, and consist of a tab-file and a plot (in PNG format) for each screen indicating the percentage of reads in the input which mapped against each genome. This acts as a check on whether your sample contains what you expect, or whether it has contamination from other sources.

Information on the `fastq_screen` program can be found at http://www.bioinformatics.bbsrc.ac.uk/projects/fastq_screen/

3.7.5 qc_boxplotter

Generates a QC boxplot from a SOLiD .qual file.

Usage:

```
qc_boxplotter.sh <solid.qual>
```

Outputs:

Two files (PostScript and PDF format) with the boxplot, called `<solid.qual>_seq-order_boxplot.ps` and `<solid.qual>_seq-order_boxplot.pdf`, which indicate the quality of the reads as a function of position.

Use *boxplotps2png.sh* to convert the PS outputs to PNG.

3.7.6 boxplotps2png.sh

Utility to generate PNGs from PS boxplots produced from *qc_boxplotter*.

Usage:

```
boxplotps2png.sh BOXPLOT1.ps [ BOXPLOT2.ps ... ]
```

Outputs:

PNG versions of the input postscript files as `BOXPLOT1.png`, `BOXPLOT2.png` etc.

Note: This uses the ImageMagick `convert` program to do the image format conversion.

3.7.7 solid_preprocess_filter.sh

The SOLiD_preprocess_filter part of the QC pipeline is implemented as a shell script `solid_preprocess_filter.sh`. which can be run independently of the main `solid_qc.sh` script. It takes a CSFASTA/QUAL file pair as input, e.g.:

```
qsub -V -b Y -N solid_preprocess_filter -wd /path/to/dir/with/data solid_preprocess_
↪filter.sh sample.csfasta sample.qual
```

and runs the `SOLiD_preprocess_filter_v2.pl` program on it.

The outputs are a "filtered" CSFASTA/QUAL file pair with the same name the inputs but with `_T_F3` appended (e.g. for the example above they would be `sample_T_F3.csfasta` and `sample_T_F3.qual`).

The script also runs a basic comparison of the input and output files to determine how many reads were removed by the filtering process. This analysis is written to the log file and also to a file called `SOLiD_preprocess_filter.stats`, for example:

```
#File       Reads   Reads after filter      Difference      % Filtered
sample01.csfasta    82352   28252   54100   65.69
sample02.csfasta    19479505        15510259        3969246 20.37
sample03.csfasta    19816967        15501222        4315745 21.77
sample04.csfasta    19581546        15293103        4288443 21.90
...
```

Typically around 20-30% of reads removed seems to be normal, anything much higher than this suggests something unusual is going on.

By default the script uses a custom set of options. To replace these with your own preferred set of options for `SOLiD_preprocess_filter_v2.pl`, specify them as arguments to the `solid_preprocess_filter.sh` script, e.g.:

```
qsub -V -b Y -N solid_preprocess_filter -wd /path/to/dir/with/data solid_preprocess_
↪filter.sh -q 3 -p 22 sample.csfasta sample.qual
```

Information on the `SOLiD_preprocess_filter_v2.pl` program can be found at http://hts.rutgers.edu/filter/

3.7.8 solid_preprocess_truncation_filter.sh

This is a variation on the `solid_preprocess_filter.sh` script which truncates the reads before applying the quality filter. It is not currently part of the QC pipeline so it must be run independently. It takes a CSFASTA/QUAL file pair as input, e.g.:

```
qsub -V -b Y -N solid_preprocess_filter -wd /path/to/dir/with/data solid_preprocess_
↪truncation_filter.sh sample.csfasta sample.qual
```

By default the truncation length is 30 bp, but this can be changed by specifying the `-u <length>` option e.g. to use 35 bp do:

```
qsub -V -b Y -N solid_preprocess_filter -wd /path/to/dir/with/data solid_preprocess_
↪truncation_filter.sh -u 35 sample.csfasta sample.qual
```

By default the output files use the input CSFASTA file name as a base for the output files, with the truncation length added (e.g. "sample_30bp"); to specify your own, use the `-o <basename>` option e.g.:

```
qsub -V -b Y -N solid_preprocess_filter -wd /path/to/dir/with/data solid_preprocess_
↪truncation_filter.sh -o myoutput sample.csfasta sample.qual
```

The script outputs the following files:

- `<basename>_T_F3.csfasta`
- `<basename>_QV_T_F3.qual`
- `<basename>_T_F3.fastq`

The script also writes statistics on the numbers of input/output reads to the `SOLiD_preprocess_filter.stats` file.

Other options supplied to the script are directly passed to the underlying `SOLiD_preprocess_filter_v2.pl` program

3.7.9 fastq_strand.py

Utility to determine the strandedness (forward, reverse, or both) from an R1/R2 pair of Fastq files.

Requires that the `STAR` mapper is also installed and available on the user's `PATH`.

Usage examples:

The simplest example checks the strandedness for a single genome:

```
fastq_strand.py R1.fastq.gz R2.fastq.gz -g STARindex/mm10
```

In this example, `STARindex/mm10` is a directory which contains the `STAR` indexes for the `mm10` genome build.

The output is a file called `R1_fastq_strand.txt` which summarises the forward and reverse strandedness percentages:

```
#fastq_strand version: 0.0.1        #Aligner: STAR  #Reads in subset: 1000
#Genome     1st forward     2nd reverse
STARindex/mm10      13.13   93.21
```

To include the count sums for unstranded, 1st read strand aligned and 2nd read strand aligned in the output file, specify the `--counts` option:

```
#fastq_strand version: 0.0.1        #Aligner: STAR  #Reads in subset: 1000
#Genome     1st forward     2nd reverse     Unstranded      1st read strand aligned␣
↪2nd read strand aligned
STARindex/mm10      13.13   93.21   391087  51339   364535
```

Strandedness can be checked for multiple genomes by specifying additional `STAR` indexes on the command line with multiple `-g` flags:

```
fastq_strand.py R1.fastq.gz R2.fastq.gz -g STARindex/hg38 -g STARindex/mm10
```

Alternatively a panel of indexes can be supplied via a configuration file of the form:

```
#Name          STAR index
hg38           /mnt/data/STARindex/hg38
mm10           /mnt/data/STARindex/mm10
```

(NB blank lines and lines starting with a # are ignored). Use the `-c`/`--conf` option to get the strandedness percentages using a configuration file, e.g.:

```
fastq_strand.py -c model_organisms.conf R1.fastq.gz R2.fastq.gz
```

By default a random subset of 1000 read pairs is used from the input Fastq pair; this can be changed using the `--subset` option. If the subset is set to zero then all reads are used.

The number of threads used to run `STAR` can be set via the `-n` option; to keep all the outputs from `STAR` specify the `--keep-star-output` option.

The strandedness statistics can also be generated for a single Fastq file, by only specifying one file on the command line. E.g.:

```
fastq_strand.py -c model_organisms.conf R1.fastq.gz
```

3.8 Microarray utilities

Scripts and tools for microarray specific tasks.

- *annotate_probesets.py*: annotate probe set list based on probeset names
- *best_exons.py*: average data for 'best' exons for each gene symbol in a file
- *xrorthologs.py*: cross-reference data for two species using probeset lookup

3.8.1 annotate_probesets.py

Usage:

```
annotate_probesets.py OPTIONS probe_set_file
```

Annotate a probeset list based on probe set names: reads in first column of tab-delimited input file *probe_set_file* as a list of probeset names and outputs these names to another tab-delimited file with a description for each.

Output file name can be specified with the *-o* option, otherwise it will be the input file name with *_annotated* appended.

Options:

-o `OUT_FILE`
: specify output file name

Example input:

```
...
1769726_at
1769727_s_at
...
```

generates output:

```
...
1769726_at  Rank 1: _at : anti-sense target (most probe sets on the array)
1769727_s_at        Warning: _s_at : designates probe sets that share common probes␣
↪among multiple transcripts from different genes
...
```

3.8.2 best_exons.py

Average data for 'best' exons for each gene symbol in a file.

Usage:

```
best_exons.py [OPTIONS] EXONS_IN BEST_EXONS
```

Read exon and gene symbol data from `EXONS_IN` and picks the top three exons for each gene symbol, then outputs averages of the associated values to `BEST_EXONS`.

Options:

--rank-by=CRITERION
: select the criterion for ranking the 'best' exons; possible options are: `log2_fold_change` (default), or `p_value`.

--probeset-col=PROBESET_COL
: specify column with probeset names (default=0, columns start counting from zero)

--gene-symbol-col=GENE_SYMBOL_COL
: specify column with gene symbols (default=1, columns start counting from zero)

--log2-fold-change-col=LOG2_FOLD_CHANGE_COL
: specify column with log2 fold change (default=12, columns start counting from zero)

--p-value-col=P_VALUE_COL
: specify column with p-value (default=13; columns start counting from zero)

--debug
: Turn on debug output

Description

Program to pick 'top' three exons for each gene symbol from a TSV file with the exon data (file has one exon per line) and output a single line for that gene symbol with values averaged over the top three.

'Top' or 'best' exons are determined by ranking on either the `log2FoldChange` (the default) or `pValue` (see the `--rank-by` option):

- For `log2FoldChange`, the 'best' exon is the one with the biggest absolute `log2FoldChange`; if this is positive or zero then takes the top three largest fold change value. Otherwise takes the bottom three.
- For `pValue`, the 'best' exon is the one with the smallest value.

Outputs a TSV file with one line per gene symbol plus the average of each data value for the 3 best exons according to the specified criterion. The averages are just the mean of all the values.

Input file format

Tab separated values (TSV) file, with first line optionally being a header line.

By default the program assumes:

- Column 0: probeset name (change using `--probeset-col`)
- Column 1: gene symbol (change using `--gene-symbol-col`)
- Column 12: log2 fold change (change using `--log2-fold-change-col`)
- Column 13: p-value (change using `--p-value-col`)

Column numbering starts from zero.

Output file format

TSV file with one gene symbol per line plus averaged data for the 'best' exons, and an extra column which has a * to indicate which gene symbols had 4 or fewer exons associated with them in the input file.

3.8.3 xrorthologs.py

Cross-reference data for two species using probe set lookup

Usage:

```
xrorthologs.py [options] LOOKUPFILE SPECIES1 SPECIES2
```

Description

Cross-reference data from two species given a lookup file that maps probe set IDs from one species onto those onto the other.

`LOOKUPFILE` is a tab-delimited file with one probe set for species 1 per line in first column and a comma-separated list of the equivalent probe sets for species 2 in the fourth column, e.g.

```
...
121_at      7849    18510   1418208_at,1446561_at
1255_g_at   2978    14913   1421061_a
1316_at     7067    21833   1426997_at,1443952_at,1454675_at
1320_at     11099   24000   1419054_a_at,1419055_a_at,1453298_at
1405_i_at   6352    20304   1418126_at
...
```

Data for the two species are in tab-delimited files `SPECIES1` and `SPECIES2`, where the first column in each is a probe set ID (this is the only requirement).

The output consists of two files:

- `SPECIES1_appended.txt`: a copy of `SPECIES1` with the cross-referenced data from `SPECIES2` appended to each line, and
- `SPECIES2_appended.txt`: a copy of `SPECIES2` with the `SPECIES1` data appended.

Where there are multiple matching orthologs to a probe set ID, the data for each match is appended onto a single line on the output.

3.9 General non-bioinformatic utilities

General utility scripts/tools.

- *cd_set_umask.sh*: setup script to automagically set umask for specific directory
- *cmpdirs.py*: compare contents of two directories
- *cluster_load.py*: report Grid Engine usage via qstat wrapper
- *do.sh*: execute shell command iteratively with range of integer index values
- *makeBinsFromBed.pl*: create bin files for binning applications
- *makeRegularBinsFromGenomeTable.R*: make bin file from set of chromosomes
- *make_mock_solid_dir.py*: create mock SOLiD directory structure for testing
- *manage_seqs.py*: handling sets of named sequences (e.g. FastQC contaminants file)
- *md5checker.py*: check files and directories using MD5 sums
- *symlink_checker.py*: check and update symbolic links

3.9.1 cd_set_umask.sh

Script to set and revert a user's `umask` appropriately when moving in and out of a particular directory (or one of its subdirectories).

3.9.2 do.sh

Execute a command multiple times, substituting a range of integer index values for each execution. For example:

```
do.sh 1 43 ln -s /blah/blah#/myfile#.ext ./myfile#.ext
```

will execute:

```
ln -s /blah/blah1/myfile1.ext ./myfile1.ext
ln -s /blah/blah2/myfile2.ext ./myfile2.ext
...
ln -s /blah/blah43/myfile43.ext ./myfile43.ext
```

3.9.3 cmpdirs.py

Recursively compare contents of one directory against another.

Usage:

```
cmpdirs.py [OPTIONS] DIR1 DIR2
```

Compare contents of `DIR1` against corresponding files and directories in *DIR2*.

Files are compared using MD5 sums, symlinks using their targets.

Options:

-n `N_PROCESSORS`
: specify number of cores to use

3.9.4 cluster_load.py

Report current Grid Engine utilisation by wrapping the `qstat` utility.

Usage:

```
cluster_load.py
```

Outputs a report of the form:

```
6 jobs running (r)
44 jobs queued (q)
0 jobs suspended (S)
0 jobs pending deletion (d)

Jobs by queue:
    queue1.q    1 (0/0)
    queue2.q    5 (0/0)
    ...

Jobs by user:
                    Total   r       q       S       d
        user1       2       1       1       0       0
        user2       15      1       14      0       0
        user3       32      4       28      0       0
        ...

Jobs by node:
                    Total   queue1.q        queue2.q
                         r (S/d)         r (S/d)
      node01    1        0 (0/0)         1 (0/0)
      node02    1        0 (0/0)         1 (0/0)
      ...
```

3.9.5 makeBinsFromBed.pl

Utility to systematically and easily create feature `bin` files, to be used in binning applications.

Example use cases include defining a region 500bp in front of the TSS, and making a set of equally spaced intervals between the start and end of a gene or feature.

Usage:

```
makeBinsFromBed.pl [options] BED_FILE OUTPUT_FILE
```

Options:

`--marker` `[ midpoint | start | end | tss | tts ]`
: On which component of feature to position the bin(s) (default midpoint).

  tss: transcription start site (using strand)

  tts: transcription termination site (using strand)

`--binType` `[ centred | upstream | downstream ]`
: How to position the bin relative to the feature (default centred).

  If marker is start/end, position is relative to chromosome.

  If marker is tss/tts, position is relative to strand of feature

--offset n
: All bins are shifted by this many bases (default 0).

  If marker is start/end, n is relative to chromosome.

  If marker is tss/tts, n is relative to strand of feature

--binSize n
: The size of the bins (default 200)

--makeIntervalBins n
: n bins are made of equal size within the feature.

  The bins begin, and are numbered from, the marker.

  If > 0, ignores binSize, offset and binType.

  Incompatible with `--marker midpoint`

Tips:

- To create single bp of the tss, use:

```
--marker tss  --binSize 1 --binType downstream
```

- To get a bin of 1000bp ending 500bp upstream of the tss, use:

```
--marker tss  --binSize 1000 --binType upstream --offset -500
```

3.9.6 makeRegularBinsFromGenomeTable.R

Make a bed file with bins of size `[binSize]` filling every chrom specified in `[Genome Table File]`

Usage:

```
makeRegularBinsFromGenomeTable.R [Genome Table File] [binSize]
```

Arguments:

- `Genome Table File`: name of a two-column tab-delimited file with chromosome name-start position information for each chromosome (i.e. the first two columns of the chromInfo table from UCSC).
- `binSize`: integer size of each bin (in bp) in the output file

Outputs:

- `Bed file`: same name as the genome table file with the extension `<binSize>.bp.bin.bed`, with each chromosome divided into bins of the requested size.

3.9.7 make_mock_solid_dir.py

Make a temporary mock SOLiD directory structure that can be used for testing.

Usage:

```
make_mock_solid_dir.py [OPTIONS]
```

Arguments:

--paired-end
: Create directory structure for paired-end run

3.9.8 manage_seqs.py

Read sequences and names from one or more INFILEs (which can be a mixture of FastQC 'contaminants' format and or Fasta format), check for redundancy (i.e. sequences with multiple associated names) and contradictions (i.e. names with multiple associated sequences).

Usage:

```
manage_seqs.py OPTIONS FILE [FILE...]
```

Options:

-o OUT_FILE
: write all sequences to OUT_FILE in FastQC 'contaminants' format

-a APPEND_FILE
: append sequences to existing APPEND_FILE (not compatible with -o)

-d DESCRIPTION
: supply arbitrary text to write to the header of the output file

Intended to help create/update files with lists of "contaminant" sequences to input into the FastQC program (using FastQC's --contaminants option).

To create a contaminants file using sequences from a FASTA file do e.g.:

```
manage_seqs.py -o custom_contaminants.txt sequences.fa
```

To append sequences to an existing contaminants file do e.g.:

```
manage_seqs.py -a my_contaminantes.txt additional_seqs.fa
```

3.9.9 md5checker.py

Utility for checking files and directories using MD5 checksums.

Usage:

To generate MD5 sums for a directory:

```
md5checker.py [ -o CHKSUM_FILE ] DIR
```

To generate the MD5 sum for a file:

```
md5checker.py [ -o CHKSUM_FILE ] FILE
```

To check a set of files against MD5 sums stored in a file:

```
md5checker.py -c CHKSUM_FILE
```

To compare the contents of source directory recursively against the contents of a destination directory, checking that files in the source are present in the target and have the same MD5 sums:

```
md5checker.py --diff SOURCE_DIR DEST_DIR
```

To compare two files by their MD5 sums:

```
md5checker.py --diff FILE1 FILE2
```

3.9.10 symlink_checker.py

Check and update symbolic links.

Usage:

```
symlink_checker.py OPTIONS DIR
```

Recursively check and optionally update symlinks found under directory DIR

Options:

--broken
: report broken symlinks

--find=REGEX_PATTERN
: report links where the destination matches the supplied REGEX_PATTERN

--replace=NEW_STRING
: update links found by --find option, by substituting REGEX_PATTERN with NEW_STRING

CHAPTER 4

bcftbx Reference

4.1 bcftbx.IlluminaData

Provides classes for extracting data about runs of Illumina-based sequencers (e.g. GA2x or HiSeq) from directory structure, data files and naming conventions.

4.1.1 Core data and run handling classes

class bcftbx.IlluminaData.**IlluminaData**(*illumina_analysis_dir*, *unaligned_dir='Unaligned'*)

Class for examining Illumina data post bcl-to-fastq conversion

Provides the following attributes:

analysis_dir: top-level directory holding the 'Unaligned' subdirectory with the primary fastq.gz files

projects: list of IlluminaProject objects (one for each project defined at the fastq creation stage)

undetermined: IlluminaProject object for the undetermined reads unaligned_dir: full path to the 'Unaligned' directory holding the

primary fastq.gz files

paired_end: True if at least one project is paired end, False otherwise format: Format of the directory structure layout (either

'casava' or 'bcl2fastq2', or None if the format cannot be determined)

lanes: List of lane numbers present; if there are no lanes then this will be a list with 'None' as the only value

Provides the following methods:

get_project(): lookup and return an IlluminaProject object corresponding to the supplied project name

class bcftbx.IlluminaData.**IlluminaProject**(*dirn*)
Class for storing information on a 'project' within an Illumina run

A project is a subset of fastq files from a run of an Illumina sequencer; in the first instance projects are defined within the SampleSheet.csv file which is output by the sequencer.

Note that the "undetermined" fastqs (which hold reads for each lane which couldn't be assigned to a barcode during demultiplexing) is also considered as a project, and can be processed using an IlluminaProject object.

Provides the following attributes:

name: name of the project dirn: (full) path of the directory for the project expt_type: the application type for the project e.g. RNA-seq, ChIP-seq

Initially set to None; should be explicitly set by the calling subprogram

samples: list of IlluminaSample objects for each sample within the project

paired_end: True if all samples are paired end, False otherwise undetermined: True if 'samples' are actually undetermined reads

class bcftbx.IlluminaData.**IlluminaRun**(*illumina_run_dir*, *platform=None*)
Class for examining 'raw' Illumina data directory.

Provides the following properties:

run_dir : name and full path to the top-level data directory basecalls_dir : name and full path to the subdirectory holding bcl files sample_sheet_csv : full path of the SampleSheet.csv file runinfo_xml : full path of the RunInfo.xml file platform : platform e.g. 'miseq' bcl_extension : file extension for bcl files (either "bcl" or "bcl.gz") lanes : list of (integer) lane numbers in the run

class bcftbx.IlluminaData.**IlluminaRunInfo**(*runinfo_xml*)
Class for examining Illumina RunInfo.xml file

Extracts basic information from a RunInfo.xml file:

run_id : the run id e.g.'130805_PJ600412T_0012_ABCDEZXDYY' run_number : the run number e.g. '12' bases_mask : bases mask string derived from the read information

e.g. 'y101,I6,y101'

reads : a list of Python dictionaries (one per read)

Each dictionary in the 'reads' list has the following keys:

number : the read number (1,2,3,...) num_cycles : the number of cycles in the read e.g. 101 is_indexed_read : whether the read is an index (i.e. barcode)

Either 'Y' or 'N'

class bcftbx.IlluminaData.**IlluminaSample**(*dirn*, *fastqs=None*, *name=None*, *prefix='Sample_'*)
Class for storing information on a 'sample' within an Illumina project

A sample is a fastq file generated within an Illumina sequencer run.

Provides the following attributes:

name: sample name dirn: (full) path of the directory for the sample fastq: name of the fastq.gz file (without leading directory, join to

'dirn' to get full path)

paired_end: boolean; indicates whether sample is paired end

4.1.2 Samplesheet handling

class bcftbx.IlluminaData.**SampleSheet**(*sample_sheet=None*, *fp=None*)

Class for handling Illumina sample sheets

This is a general class which tries to handle and convert between older (i.e. 'CASAVA'-style) and newer (IEM-style) sample sheet files for Illumina sequencers, in a transparent manner.

The Experimental Manager (IEM) samplel sheets are text files with data delimited by '[...]' lines e.g. '[Header]', '[Reads]' etc.

The 'Header' section consists of comma-separated key-value pairs e.g. 'Application,HiSeq FASTQ Only'.

The 'Reads' section consists of values (one per line) (possibly number of bases per read?) e.g. '101'.

The 'Settings' section consists of comma-separated key-value pairs e.g. 'Adapter,CTGTCTCTTATACACATCT'.

The 'Data' section contains the data about the lanes, samples and barcode indexes. It consists of lines of comma-separated values, with the first line being a 'header', and the remainder being values for each of those fields.

This older style of sample sheet is used by CASAVA and bcl2fastq v1.8.*. It consists of lines of comma-separated values, with the first line being a 'header' and the remainder being values for each of the fields:

FCID: flow cell ID Lane: lane number (integer from 1 to 8) SampleID: ID (name) for the sample SampleRef: reference used for alignment for the sample Index: index sequences (multiple index reads are separated by a

hyphen e.g. ACCAGTAA-GGACATGA

Description: Description of the sample Control: Y indicates this lane is a control lane, N means sample Recipe: Recipe used during sequencing Operator: Name or ID of the operator SampleProject: project the sample belongs to

Although the CASAVA-style sample sheet looks much like the IEM 'Data' section, note that it has different fields and field names.

To load data from an IEM-format file:

```
>>> iem = SampleSheet('SampleSheet.csv')
```

To access 'header' items:

```
>>> iem.header_items
['IEMFileVersion','Date',..]
>>> iem.header['IEMFileVersion']
'4'
```

To access 'reads' data:

```
>>> iem.reads
['101','101']
```

To access 'settings' items:

```
>>> iem.settings_items
['ReverseComplement',...]
>>> iem.settings['ReverseComplement']
'0'
```

To access 'data' (the actual sample sheet information):

```
>>> iem.data.header()
['Lane','Sample_ID',...]
>>> iem.data[0]['Lane']
1
```

etc.

To load data from a CASAVA style sample sheet:

```
>>> casava = SampleSheet('SampleSheet.csv')
```

To access the data use the 'data' property:

```
>>> casava.data.header()
['Lane','SampleID',...]
>>> casava.data[0]['Lane']
1
```

The data in the 'Data' section can be accessed directly from the SampleSheet instance, e.g.

```
>>> iem[0]['Lane']
```

is equivalent to

```
>>> iem.data[0]['Lane']
```

It is also possible to set new values for data items using this notation.

The data lines can be iterated over using:

```
>>> for line in iem:
>>> ...
```

To find the number of lines that are stored:

```
>>> len(iem)
```

To append a new line:

```
>>> new_line = iem.append(...)
```

A number of methods are available to check and fix common problems, specifically:

- detect and replace 'illegal' characters in sample and project names
- detect and fix duplicated sample name, project and lane combinations
- detect blank sample and project names

Data is loaded it is also subjected to some basic cleaning up, including stripping of unnecessary commas and white space. The 'show' method returns a reconstructed version of the original sample sheet after the cleaning operations were performed.

class bcftbx.IlluminaData.**CasavaSampleSheet**(*samplesheet=None*, *fp=None*)

Class for reading and manipulating sample sheet files for CASAVA

This class is a subclass of the SampleSheet class, and provides an additional method ('casava_sample_sheet') to convert to a CASAVA-style sample sheet, suitable for input into bcl2fastq version 1.8.*.

Raises IlluminaDataError exception if the input data doesn't appear to be in the correct format.

class bcftbx.IlluminaData.**IEMSampleSheet**(*sample_sheet=None*, *fp=None*)
Class for handling Experimental Manager format sample sheet

This class is a subclass of the SampleSheet class, and provides an additional method ('casava_sample_sheet') to convert to a CASAVA-style sample sheet, suitable for input into bcl2fastq version 1.8.*.

bcftbx.IlluminaData.**convert_miseq_samplesheet_to_casava**(*samplesheet=None*, *fp=None*)
Convert a Miseq sample sheet file to CASAVA format

Reads the data in a Miseq-format sample sheet file and returns a CasavaSampleSheet object with the equivalent data.

Note: this is now just a wrapper for the more general conversion function 'get_casava_sample_sheet' (which can handle the conversion without knowing a priori what the SampleSheet format is.

Arguments: samplesheet: name of the Miseq sample sheet file

Returns: A populated CasavaSampleSheet object.

bcftbx.IlluminaData.**get_casava_sample_sheet**(*samplesheet=None*, *fp=None*, *FCID_default='FC1'*)
Load data into a 'standard' CASAVA sample sheet CSV file

Reads the data from an Illumina platform sample sheet CSV file and populates and returns a CasavaSampleSheet object which can be used to generate make a SampleSheet suitable for bcl-to-fastq conversion.

The source sample sheet may be in the format output by the Experimental Manager software (needed when running BaseSpace) or may already be in "standard" format for bcl-to-fastq format.

For Experimental Manager format, the sample sheet consists of sections delimited by headers of the form "[Header]", "[Reads]" etc. The information about the sample names and barcodes are in the "[Data]" section, which is essentially a list of CSV format lines with the following fields:

MiSEQ:

Sample_ID,Sample_Name,Sample_Plate,Sample_Well,I7_Index_ID,index, Sample_Project,Description

HiSEQ:

Lane,Sample_ID,Sample_Name,Sample_Plate,Sample_Well,I7_Index_ID, index,Sample_Project,Description

(Note that for dual-indexed runs the fields are e.g.:

Sample_ID,Sample_Name,Sample_Plate,Sample_Well,I7_Index_ID,index, I5_Index_ID,index2,Sample_Project,Description

i.e. there are an additional pair of fields describing the second index)

The conversion maps a subset of these onto fields in the Casava format:

Sample_ID -> SampleID index -> Index Sample_Project -> SampleProject Description -> Description

If no lane information is present in the original file then this is set to 1. The FCID is set to an arbitrary value.

For dual-indexed samples, the Index field is generated by putting together the index and index2 fields.

All other fields are left empty.

Arguments: samplesheet: name of the Miseq sample sheet file FCID_default: name to use for flow cell ID if not present in

the source file (optional)

Returns: A populated CasavaSampleSheet object.

bcftbx.IlluminaData.**verify_run_against_sample_sheet**(*illumina_data*, *sample_sheet*, *include_sample_dir=False*)

Checks existence of predicted outputs from a sample sheet

Arguments: illumina_data: a populated IlluminaData directory sample_sheet : path and name of a CSV sample sheet include_sample_dir: if True then always include a

'sample_name' directory level when checking for bcl2fastq2 outputs

Returns:

True if all the predicted outputs from the sample sheet are found, False otherwise.

bcftbx.IlluminaData.**samplesheet_index_sequence**(*line*)

Return the index sequence for a sample sheet line

Arguments: line (TabDataLine): line from a SampleSheet instance

Returns: String: barcode sequence, or 'None' if not defined.

bcftbx.IlluminaData.**normalise_barcode**(*seq*)

Return normalised version of barcode sequence

This standardises the sequence so that:

- all bases are uppercase
- dual index barcodes have '-' and '+' removed

4.1.3 Utility classes and functions

class bcftbx.IlluminaData.**IlluminaFastq**(*fastq*)

Class for extracting information about Fastq files

Given the name of a Fastq file from CASAVA/Illumina platform, extract data about the sample name, barcode sequence, lane number, read number and set number.

For Fastqs produced by CASAVA and bcl2fastq v1.8, the format of the names follows the general form:

<sample_name>_<barcode_sequence>_L<lane_number>_R<read_number>_<set_number>.fastq.gz

e.g. for

NA10831_ATCACG_L002_R1_001.fastq.gz

sample_name = 'NA10831' barcode_sequence = 'ATCACG' lane_number = 2 read_number = 1 set_number = 1

For Fastqs produced by bcl2fast v2, the format looks like:

<sample_name>_S<sample_number>_L<lane_number>_R<read_number>_<set_number>.fastq.gz

e.g. for

NA10831_S4_L002_R1_001.fastq.gz

sample_name = 'NA10831' sample_number = 4 lane_number = 2 read_number = 1 set_number = 1

Provides the follow attributes:

fastq: the original fastq file name sample_name: name of the sample (leading part of the name) sample_number: number of the same (integer or None, bcl2fastq v2 only) barcode_sequence: barcode sequence (string or None, CASAVA/bcl2fast v1.8 only) lane_number: integer read_number: integer set_number: integer

bcftbx.IlluminaData.**describe_project**(*illumina_project*)
Generate description string for samples in a project

Description string gives the project name and a human-readable summary of the sample names, plus number of samples and whether the data is paired end.

Example output: "Project Control: PhiX_1-2 (2 samples)"

Arguments illumina_project: IlluminaProject instance

Returns Description string.

bcftbx.IlluminaData.**fix_bases_mask**(*bases_mask*, *barcode_sequence*)
Adjust input bases mask to match actual barcode sequence lengths

Updates the bases mask string extracted from RunInfo.xml so that the index read masks correspond to the index barcode sequence lengths given e.g. in the SampleSheet.csv file.

For example: if the bases mask is 'y101,I7,y101' (i.e. assigning 7 cycles to the index read) but the barcode sequence is 'CGATGT' (i.e. only 6 bases) then the adjusted bases mask should be 'y101,I6n,y101'.

Arguments: bases_mask: bases mask string e.g. 'y101,I7,y101','y250,I8,I8,y250' barcode_sequence: index barcode sequence e.g. 'CGATGT' (single index), 'TAAGGCGA-TAGATCGC' (dual index)

Returns: Updated bases mask string.

bcftbx.IlluminaData.**get_unique_fastq_names**(*fastqs*)
Generate mapping of full fastq names to shorter unique names

Given an iterable list of Illumina file fastq names, return a dictionary mapping each name to its shortest unique form within the list.

Arguments: fastqs: an iterable list of fastq names

Returns: Dictionary mapping fastq names to shortest unique versions

bcftbx.IlluminaData.**split_run_name**(*dirname*)
Split an Illumina directory run name into components

Given a directory for an Illumina run, e.g.

140210_M00879_0031_000000000-A69NA

split the name into components and return as a tuple:

(date_stamp,instrument_name,run_number)

e.g.

('140210','M00879','0031')

Note that this function doesn't return the flow cell ID; use the `split_run_name_full` function to also extract the flow cell information.

bcftbx.IlluminaData.**summarise_projects**(*illumina_data*)
Short summary of projects, suitable for logging file

The summary description is a one line summary of the project names along with the number of samples in each, and an indication if the run was paired-ended.

Arguments: illumina_data: a populated IlluminaData directory

Returns: Summary description.

bcftbx.IlluminaData.**get_unique_fastq_names**(*fastqs*)
Generate mapping of full fastq names to shorter unique names

Given an iterable list of Illumina file fastq names, return a dictionary mapping each name to its shortest unique form within the list.

Arguments: fastqs: an iterable list of fastq names

Returns: Dictionary mapping fastq names to shortest unique versions

4.1.4 Exception classes

class bcftbx.IlluminaData.**IlluminaDataError**
Base class for errors with Illumina-related code

4.2 bcftbx.SolidData

Provides classes for extracting data about SOLiD runs from directory structure, data files and naming conventions.

Typical usage is to create a new SolidRun instance by pointing it at the top-level output directory produced by the sequencer:

```
>>> solid_run = SolidRun('/path/to/solid0123_20141225_FRAG_BC')
```

This will automatically attempt to collect the data about the run, which can then be accessed via other objects linked through the SolidRun object's properties.

The most useful are:

SolidRun.run_info: a SolidRunInfo object which holds data extracted from the run name (e.g. instrument, datestamp etc)

SolidRun.samples: a list of SolidSample objects which hold data about each of the samples in the run.

Each sample in turn holds a list of libraries within that sample (SolidLibrary objects in 'SolidSample.libraries') and a list of projects (SolidProject objects in 'SolidSample.projects'). The 'getLibrary' and 'getProject' methods also provide ways to look up specific libraries or projects.

Projects are groupings of libraries (based on library names) which are assumed to form a single experiment. The libraries within a project can be obtained via the SolidLibrary.projects, or using the 'getLibrary' method.

Finally, SolidLibrary objects hold data about the location of the primary data files. The 'SolidLibrary.csfasta' and 'SolidLibrary.qual' properties hold the locations of the data for the F3 reads, while for paired-end runs the 'SolidLibrary.csfasta_f5' and 'SolidLibrary.qual_f5' properties point to the F5 reads.

(The 'is_paired_end' function can be used to test whether a SolidRun object holds data for a paired-end run.)

4.2.1 SolidRun

class bcftbx.SolidData.**SolidRun**(*solid_run_dir*)
Describe a SOLiD run.

The SolidRun class provides an interface to data about a SOLiD run. It analyses the SOLiD data directory to look for run definitions, statistics files and primary data files.

It uses the same terminology as the SETS interface and the data files produced by the SOLiD instrument, so a run contains 'samples' and each sample contains one or more 'libraries'.

One initialised, access the data about the run via the SolidRun object's properties:

run_dir: directory with the run data run_name: name of the run e.g. solid0123_20130426_FRAG_BC run_info: a SolidRunInfo object with data derived from the run name run_definition: a SolidRunDefinition object with data extracted from

the run_definition.txt file

samples: a list of SolidSample objects representing the samples in the run

class bcftbx.SolidData.**SolidRunInfo**(*run_name*)
Extract data about a run from the run name

Run names are of the form 'solid0123_20130426_FRAG_BC_2'

This class analyses the name and breaks it down into components that can be accessed as object properties, specifically:

name: the supplied run name instrument: the instrument name e.g. solid0123 datestamp: e.g. 20130426 is_fragment_library: True or False is_barcoded_sample: True or False flow_cell: 1 or 2 date: datestamp reformatted as DD/MM/YY id: the run name without any flow cell identifier

class bcftbx.SolidData.**SolidRunDefinition**(*run_definition_file*)
Class to store data from a SOLiD run definition file

Once the SolidRunDefinition object is populated from a run definition file, use the 'nSamples' method to find out how many 'samples' (actually sample/library pairs) are defined, and the 'fields' method to get a list of column headings for each.

Data can be extracted for each sample using the 'getDataItem' method to look up the value for a particular field on a particular line, e.g.:

```
>>> library = run_defn.getDataItem('library',0)
```

The SolidRunDefinition object also has a number of attributes populated from the header of the run definition file, specifically:

version, userId, runType, isMultiplexing, runName, runDesc, mask and protocol.

The attributes are strings and can be accessed directly from the object, e.g.:

```
>>> version = run_defn.version
>>> isMultiplexing = run_defn.isMultiplexing
```

class bcftbx.SolidData.**SolidBarcodeStatistics**(*barcode_statistics_file*)
Store data from a SOLiD BarcodeStatistics file

class bcftbx.SolidData.**SolidProject**(*name*, *run=None*, *sample=None*)
Class to hold information about a SOLiD 'project'

A SolidProject object holds a collection of libraries which together constitute a 'project'.

The definition of a 'project' is quite loose in this context: essentially it's a grouping of libraries within a sample. Typically the grouping is by the initial letters of the library name e.g. DR for DR1, EP for EP_NCYC2669 - but this determination is made at the application level.

Libraries are added to the project via the addLibrary method. Data about the project can be accessed via the following properties:

name: the project name (supplied on object creation) libraries: a list of libraries in the project

Also has the following methods:

getSample(): returns the parent SolidSample getRun(): returns the parent SolidRun isBarcoded(): returns boolean indicating whether the libraries

in the sample are barcoded

4.2.2 SolidSample

class bcftbx.SolidData.**SolidSample**(*name*, *parent_run=None*)

Store information about a sample in a SOLiD run.

A sample has a name and contains a set of libraries. The information about the sample can be accessed via the following properties:

name: the sample name libraries: a list of SolidLibrary objects representing the libraries

within the sample

projects: a list of SolidProject objects representing groups of related libraries within the sample

unassigned: SolidProject object representing the 'unassigned' data barcode_stats: a SolidBarcodeStats object with data extracted from

the BarcodeStatistics file (or None, if no file was available)

parent_run: the parent SolidRun object, or None.

The class also provides the following methods:

addLibrary: to create and append a SolidLibrary object getLibrary: fetch an existing SolidLibrary getProject: fetch an existing SolidProject

Typically the calling subprogram calls the 'addLibrary' method to add a SolidLibrary object, which it then populates itself.

The SolidSample class automatically creates SolidProject objects based on the library names to group libraries considered to belong to the same experiments.

4.2.3 SolidLibrary

class bcftbx.SolidData.**SolidLibrary**(*name*, *parent_sample=None*)

Store information about a SOLiD library.

The following properties hold data about the library:

name: the library name initials: the experimenter's initials prefix: the library name prefix (i.e. name without the trailing

numbers)

index_as_string: the trailing numbers from the name, as a string (preserves any leading zeroes)

index: the trailing numbers from the name as an integer csfasta: full path to the csfasta file for the library (F3 reads) qual: full path to qual file for the library (F3 reads) csfasta_f5: full path to the F5 read (paired-end runs, otherwise

will be None)

qual_f5: full path to the F5 read (paired-end runs, otherwise will be None)

primary_data: list of SolidPrimaryData objects for all possible primary data file pairs associated with the library

parent_sample: parent SolidSample object, or None.

The following methods are also available:

addPrimaryData: creates a new SolidPrimaryData object and appends to the list in the primary_data property

4.2.4 SolidPrimaryData

class bcftbx.SolidData.**SolidPrimaryData**
Class to store references to primary data files

This is a convenience class for storing references to csfasta/qual file pairs within a SolidLibrary instance.

The class provides the following attributes:

csfasta: full path to csfasta file qual: full path to qual file timestamp: timestamp associated with the file pair type: string indicating 'F3' or 'F5', or None

The following methods are provided:

is_f3: indicates if data is F3 is_f5: indicates if data is F5

4.2.5 Functions

bcftbx.SolidData.**extract_library_timestamp**(*path*)
Extract the timestamp string from a path

Given a path of the form '/path/to/data/.../primary.1234567/...', return the timestamp string attached to the 'primary.XXXXXXX' component of the name.

Arguments:

path: absolute or relative path to arbitrary directory or file in the SOLiD data structure

Returns: Timestamp string, or None if no timestamp was identified.

bcftbx.SolidData.**get_primary_data_file_pair**(*dirn*)
Return csfasta/qual file pair from specified directory

Arguments: dirn: directory to search for csfasta/qual pair

Returns: Tuple (csfasta,qual) with full path for each file, or (None,None) if a pair wasn't located.

bcftbx.SolidData.**is_paired_end**(*solid_run*)
Determine if a SolidRun instance is a paired-end run

Arguments: solid_run: a populated SolidRun instance

Returns: True if this is a paired-end run, False otherwise.

bcftbx.SolidData.**match**(*pattern*, *word*)
Check if a word matches pattern

Implements a very simple pattern matching algorithm, which allows only exact matches or glob-like strings (i.e. using a trailing '*' to indicate a wildcard).

For example: 'ABC*' matches 'ABC', 'ABC1', 'ABCDEFG' etc, while 'ABC' only matches itself.

Arguments: pattern: simple glob-like pattern word: string to test against 'pattern'

Returns: True if 'word' is a match to 'pattern', False otherwise.

bcftbx.SolidData.**slide_layout**(*nsamples*)
Description of the slide layout based on number of samples

Arguments: nsamples: number of samples in the run

Returns: A string describing the slide layout for the run based on the number of samples in the run, e.g. "Whole slide", "Quads", "Octets" etc. Returns None if the number of samples doesn't map to a recognised layout.

4.3 bcftbx.Experiment

Experiment.py

The Experiment module provides two classes: the Experiment class defines a single experiment (essentially a collection of one or more related primary data sets) from a SOLiD run; the ExperimentList class is a collection of experiments which are typically part of the same SOLiD run.

class bcftbx.Experiment.**Experiment**
Class defining an experiment from a SOLiD run.

An 'experiment' is a collection of related data.

copy()
Return a new Experiment instance which is a copy of this one.

describe()
Describe the experiment as a set of command line options

dirname(*top_dir=None*)
Return directory name for experiment

The directory name is the supplied name plus the experiment type joined by an underscore, unless no type was specified (in which case it is just the experiment name).

If top_dir is also supplied then this will be prepended to the returned directory name.

class bcftbx.Experiment.**ExperimentList**(*solid_run_dir=None*)
Container for a collection of Experiments

Experiments are created and added to the ExperimentList by calling the addExperiment method, which returns a new Experiment object.

The calling subprogram then populates the Experiment properties as appropriate.

Once all Experiments are defined the analysis directory can be constructed by calling the buildAnalysisDirs method, which creates directories and symbolic links to primary data according to the definition of each experiment.

addDuplicateExperiment(*expt*)
Duplicate an existing Experiment and add to the list

Arguments: expt: an existing Experiment object

Returns: New Experiment object with the same data as the input

addExperiment(*name*)
Create a new Experiment and add to the list

Arguments: name: the name of the new experiment

Returns: New Experiment object with name already set

buildAnalysisDirs(*top_dir=None*, *dry_run=False*, *link_type='relative'*, *naming_scheme='partial'*)
Construct and populate analysis directories for the experiments

For each defined experiment, create the required analysis directories and populate with links to the primary data files.

Arguments:

top_dir: if set then create the analysis directories as subdirs of the specified directory; otherwise operate in cwd

dry_run: if True then only report the mkdir, ln etc operations that would be performed. Default is False (do perform the operations).

link_type: type of link to use when linking to primary data, one of 'relative' or 'absolute'.

naming_scheme: naming scheme to use for links to primary data, one of 'full' (same names as primary data files), 'partial' (cut-down version of the full name which excludes sample names - the default), or 'minimal' (just the library name).

getLastExperiment()
Return the last Experiment added to the list

class bcftbx.Experiment.**LinkNames**(*scheme*)
Class to construct names for links to primary data files

The LinkNames class encodes a set of naming schemes that are used to construct names for the links in the analysis directories that point to the primary CFASTA and QUAL data files.

The schemes are:

full: link name is the same as the source file, e.g. solid0123_20111014_FRAG_BC_AB_CD_EF_pool_F3_CD_PQ5.csfa

partial: link name consists of the instrument name, datestamp and library name, e.g. solid0123_20111014_CD_PQ5.csfasta

minimal: link name consists of just the library name, e.g. CD_PQ5.csfasta

For paired-end data, the 'partial' and 'minimal' names have '_F3' and '_F5' appended as appropriate (full names already have this distinction).

Example usage:

To get the link names using the minimal scheme for the F3 reads ('library' is a SolidLibrary object):

```
>>> csfasta_lnk,qual_lnk = LinkNames('minimal').names(library)
```

To get names for the F5 reads using the partial scheme:

```
>>> csfasta_lnk,qual_lnk = LinkNames('partial').names(library,F5=True)
```

names(*library*, *F5=False*)
Get names for links to the primary data in a library

Returns a tuple of link names:

(csfasta_link_name,qual_link_name)

derived from the data in the library plus the naming scheme specified when the LinkNames object was created.

Arguments: library: SolidLibrary object F5: if True then indicates that names should be returned

for linking to the F5 reads (default is F3 reads)

4.4 bcftbx.FASTQFile

A set of classes for reading through FASTQ files and manipulating the data within them:

- FastqIterator: enables looping through all read records in FASTQ file
- FastqRead: provides access to a single FASTQ read record
- SequenceIdentifier: provides access to sequence identifier info in a read
- FastqAttributes: provides access to gross attributes of FASTQ file

Additionally there are a few utility functions:

- get_fastq_file_handle: return a file handled opened for reading a FASTQ file
- nreads: return the number of reads in a FASTQ file
- fastqs_are_pair: check whether two FASTQs form an R1/R2 pair

Information on the FASTQ file format: http://en.wikipedia.org/wiki/FASTQ_format

class bcftbx.FASTQFile.**FastqAttributes**(*fastq_file=None*, *fp=None*)
Class to provide access to gross attributes of a FASTQ file

Given a FASTQ file (can be uncompressed or gzipped), enables various attributes to be queried via the following properties:

nreads: number of reads in the FASTQ file fsize: size of the file (in bytes)

fsize
Return size of the FASTQ file (bytes)

nreads
Return number of reads in the FASTQ file

class bcftbx.FASTQFile.**FastqIterator**(*fastq_file=None*, *fp=None*, *bufsize=102400*)
Class to loop over all records in a FASTQ file, returning a FastqRead object for each record.

Example looping over all reads:

```
>>> for read in FastqIterator(fastq_file):
>>>     print read
```

Input FASTQ can be in gzipped format; FASTQ data can also be supplied as a file-like object opened for reading, for example:

```
>>> fp = open(fastq_file,'rU')
>>> for read in FastqIterator(fp=fp):
>>>     print read
>>> fp.close()
```

next()
Return next record from FASTQ file as a FastqRead object

class bcftbx.FASTQFile.**FastqRead**(*seqid_line=None*, *seq_line=None*, *optid_line=None*, *quality_line=None*)
Class to store a FASTQ record with information about a read

Provides the following properties for accessing the read data:

seqid: the "sequence identifier" information (first line of the read record) as a SequenceIdentifier object

sequence: the raw sequence (second line of the record) optid: the optional sequence identifier line (third line of the record) quality: the quality values (fourth line of the record)

Additional properties:

raw_seqid: the original sequence identifier string supplied when the object was created

seqlen: length of the sequence maxquality: maximum quality value (in character representation) minquality: minimum quality value (in character representation)

(Note that quality scores can only be obtained from character representations once the encoding scheme is known)

is_colorspace: returns True if the read looks like a colorspace read, False otherwise

class bcftbx.FASTQFile.**SequenceIdentifier**(*seqid*)
Class to store/manipulate sequence identifier information from a FASTQ record

Provides access to the data items in the sequence identifier line of a FASTQ record.

format
Identify the format of the sequence identifier

Returns: String: 'illumina18', 'illumina' or None

is_pair_of(*seqid*)
Check if this forms a pair with another SequenceIdentifier

bcftbx.FASTQFile.**fastqs_are_pair**(*fastq1=None*, *fastq2=None*, *verbose=True*, *fp1=None*, *fp2=None*)
Check that two FASTQs form an R1/R2 pair

Arguments: fastq1: first FASTQ fastq2: second FASTQ

Returns: True if each read in fastq1 forms an R1/R2 pair with the equivalent read (i.e. in the same position) in fastq2, otherwise False if any do not form an R1/R2 (or if there are more reads in one than than the other).

bcftbx.FASTQFile.**get_fastq_file_handle**(*fastq*)
Return a file handle opened for reading for a FASTQ file

Deals with both compressed (gzipped) and uncompressed FASTQ files.

Arguments:

fastq: name (including path, if required) of FASTQ file. The file can be gzipped (must have '.gz' extension)

Returns: File handle that can be used for read operations.

bcftbx.FASTQFile.**nreads**(*fastq=None*, *fp=None*)
Return number of reads in a FASTQ file

Performs a simple-minded read count, by counting the number of lines in the file and dividing by 4.

The FASTQ file can be specified either as a file name (using the 'fastq' argument) or as a file-like object opened for line reading (using the 'fp' argument).

This function can handle gzipped FASTQ files supplied via the 'fastq' argument.

Line counting uses a variant of the "buf count" method outlined here: http://stackoverflow.com/a/850962/579925

Arguments: fastq: fastq(.gz) file fp: open file descriptor for fastq file

Returns: Number of reads

4.5 bcftbx.JobRunner

Classes for starting, stopping and managing jobs.

Class BaseJobRunner is a template with methods that need to be implemented by subclasses. The subclasses implemented here are:

- SimpleJobRunner: run jobs (e.g. scripts) on a local file system.
- GEJobRunner : run jobs using Grid Engine (GE) i.e. qsub, qdel etc
- DRMAAJobRunner : run jobs using the DRMAA interface to Grid Engine

A single JobRunner instance can be used to start and manage multiple processes.

Each job is started by invoking the 'run' method of the runner. This returns an id string which is then used in calls to the 'isRunning', 'terminate' etc methods to check on and control the job.

The runner's 'list' method returns a list of running job ids.

Simple usage example:

```
>>> # Create a JobRunner instance
>>> runner = SimpleJobRunner()
>>> # Start a job using the runner and collect its id
>>> job_id = runner.run('Example',None,'myscript.sh')
>>> # Wait for job to complete
>>> import time
>>> while runner.isRunning(job_id):
>>>     time.sleep(10)
>>> # Get the names of the output files
>>> log,err = (runner.logFile(job_id),runner.errFile(job_id))
```

class bcftbx.JobRunner.**BaseJobRunner**

Base class for implementing job runners

This class can be used as a template for implementing custom job runners. The idea is that the runners wrap the specifics of interacting with an underlying job control system and thus provide a generic interface to be used by higher level classes.

A job runner needs to implement the following methods:

run : starts a job running terminate : kills a running job list : lists the running job ids logFile : returns the name of the log file for a job errFile : returns the name of the error file for a job exit_status: returns the exit status for the command (or

None if the job is still running)

Optionally it can also implement the methods:

errorState: indicates if running job is in an "error state" isRunning : checks if a specific job is running

if the default implementations are not sufficient.

errFile(*job_id*)

Return name of error file relative to working directory

errorState(*job_id*)

Check if the job is in an error state

Return True if the job is deemed to be in an 'error state', False otherwise.

exit_status(*job_id*)
Return the exit status code for the command

Return the exit status code from the command that was run by the specified job, or None if the job hasn't exited yet.

isRunning(*job_id*)
Check if a job is running

Returns True if job is still running, False if not

list()
Return a list of running job_ids

logFile(*job_id*)
Return name of log file relative to working directory

log_dir
Return the current log directory setting

run(*name*, *working_dir*, *script*, *args*)
Start a job running

Arguments: name: Name to give the job working_dir: Directory to run the job in script: Script file to run args: List of arguments to supply to the script

Returns: Returns a job id, or None if the job failed to start

set_log_dir(*log_dir*)
(Re)set the directory to write log files to

terminate(*job_id*)
Terminate a job

Returns True if termination was successful, False otherwise

class bcftbx.JobRunner.**DRMAAJobRunner**(*queue=None*)
Class implementing job runner using DRMAA

DRMAAJobRunner submits jobs to a Grid Engine cluster using the Python interface to Distributed Resource Management Application API (DRMAA), as an alternative to the GEJobRunner.

The DRMAAJobRunner requires: - the drmaa libraries (e.g. libdrmaa.so), pointed to by the

environment variable DRMAA_LIBRARY_PATH

- the Python drmma library, see http://code.google.com/p/drmaa-python/

errFile(*job_id*)
Return the error file name for a job

The name should be '<name>.e<job_id>'

errorState(*job_id*)
Check if the job is in an error state

Return True if the job is deemed to be in an 'error state', False otherwise.

list()
Get list of job ids in the queue.

logFile(*job_id*)
Return the log file name for a job

The name should be '<name>.o<job_id>'

queue(*job_id*)
Fetch the job queue name

Returns the queue as reported by qstat, or None if not found.

run(*name*, *working_dir*, *script*, *args*)
Submit a script or command to the cluster via DRMAA

Arguments: name: Name to give the job working_dir: Directory to run the job in script: Script file to run args: List of arguments to supply to the script

Returns: Job id for submitted job, or 'None' if job failed to start.

terminate(*job_id*)
Remove a job from the GE queue

class bcftbx.JobRunner.**GEJobRunner**(*queue=None*, *log_dir=None*, *ge_extra_args=None*, *poll_interval=5.0*, *timeout=30.0*)
Class implementing job runner for Grid Engine

GEJobRunner submits jobs to a Grid Engine cluster using the 'qsub' command, determines the status of jobs using 'qstat' and terminates then using 'qdel'.

Additionally the runner can be configured for a specific GE queue on initialisation.

Each GEJobRunner instance creates a temporary directory which it uses for internal admin; this will be removed at program exit via 'atexit'.

errFile(*job_id*)
Return the error file name for a job

The name should be '<name>.e<job_id>'

errorState(*job_id*)
Check if the job is in an error state

Return True if the job is deemed to be in an 'error state' (i.e. qstat returns the state as 'E..'), False otherwise.

exit_status(*job_id*)
Return exit status from command run by a job

If the job is still running then returns 'None'.

ge_extra_args
Return the extra GE arguments

list()
Get list of job ids which are queued or running

logFile(*job_id*)
Return the log file name for a job

The name should be '<name>.o<job_id>'

name(*job_id*)
Return the name for a job

queue(*job_id*)
Fetch the job queue name

Returns the queue as reported by qstat, or None if not found.

run(*name*, *working_dir*, *script*, *args*)
Submit a script or command to the cluster via 'qsub'

Arguments: name: Name to give the job working_dir: Directory to run the job in script: Script file to run args: List of arguments to supply to the script

Returns: Job id for submitted job, or 'None' if job failed to start.

terminate(*job_id*)
Remove a job from the GE queue using 'qdel'

class bcftbx.JobRunner.**SimpleJobRunner**(*log_dir=None*, *join_logs=False*)
Class implementing job runner for local system

SimpleJobRunner starts jobs as processes on a local system; the status of jobs is determined using the Linux 'ps eu' command, and jobs are terminated using 'kill -9'.

errFile(*job_id*)
Return the error file name for a job

exit_status(*job_id*)
Return exit status from command run by a job

list()
Return a list of running job_ids

logFile(*job_id*)
Return the log file name for a job

name(*job_id*)
Return the name for a job

run(*name*, *working_dir*, *script*, *args*)
Run a command and return the PID (=job id)

Arguments: name: Name to give the job working_dir: Directory to run the job in script: Script file to run args: List of arguments to supply to the script

Returns: Job id for submitted job, or 'None' if job failed to start.

terminate(*job_id*)
Kill a running job using 'kill -9'

bcftbx.JobRunner.**fetch_runner**(*definition*)
Return job runner instance based on a definition string

Given a definition string, returns an appropriate runner instance.

Definitions are of the form:

RunnerName[(args)]

RunnerName can be 'SimpleJobRunner' or 'GEJobRunner'. If '(args)' are also supplied then these are passed to the job runner on instantiation (only works for GE runners).

4.6 bcftbx.Pipeline

Classes for running scripts iteratively over a collection of data files.

The essential classes are:

- Job: wrapper for setting up, submitting and monitoring running scripts
- PipelineRunner: queue and run script multiple times on standard set of inputs

- SolidPipelineRunner: subclass of PipelineRunner specifically for running on SOLiD data (i.e. pairs of csfasta/qual files)

There are also some useful methods:

- GetSolidDataFiles: collect csfasta/qual file pairs from a specific directory
- GetSolidPairedEndFile: collect csfasta/qual file pairs for paired end data
- GetFastqFiles: collect fastq files from a specific directory
- GetFastqGzFiles: collect gzipped fastq files

The PipelineRunners depend on the JobRunner instances (created from classes in the JobRunner module) to interface with the job management system. So typical usage might look like:

```
>>> import JobRunner
>>> import Pipeline
>>> runner = JobRunner.GEJobRunner() # to use Grid Engine
>>> pipeline = Pipeline.PipelineRunner(runner)
>>> pipeline.queueJob(...)
>>> pipeline.run()
```

4.6.1 Classes

class bcftbx.Pipeline.**Job**(*runner*, *name*, *dirn*, *script*, *args*, *label=None*, *group=None*)

Wrapper class for setting up, submitting and monitoring running scripts

Set up a job by creating a Job instance specifying the name, working directory, script file to execute, and arguments to be supplied to the script.

The job is started by invoking the 'start' method; its status can be checked with the 'isRunning' method, and terminated and restarted using the 'terminate' and 'restart' methods respectively.

Information about the job can also be accessed via its properties. The following properties record the original parameters supplied on instantiation:

name working_dir script args label group_label

Additional information is set once the job has started or stopped running:

job_id The id number for the running job returned by the JobRunner log The log file for the job (relative to working_dir) start_time The start time (seconds since the epoch) end_time The end time (seconds since the epoch) exit_status The exit code from the command that was run (integer, or None)

The Job class uses a JobRunner instance (which supplies the necessary methods for starting, stopping and monitoring) for low-level job interactions.

class bcftbx.Pipeline.**PipelineRunner**(*runner*, *max_concurrent_jobs=4*, *poll_interval=30*, *jobCompletionHandler=None*, *groupCompletionHandler=None*)

Class to run and manage multiple concurrent jobs.

PipelineRunner enables multiple jobs to be queued via the 'queueJob' method. The pipeline is then started using the 'run' method - this starts each job up to a a specified maximum of concurrent jobs, and then monitors their progress. As jobs finish, pending jobs are started until all jobs have completed.

Example usage:

```
>>> p = PipelineRunner()
>>> p.queueJob('/home/foo','foo.sh','bar.in')
... Queue more jobs ...
>>> p.run()
```

By default the pipeline runs in 'blocking' mode, i.e. 'run' doesn't return until all jobs have been submitted and have completed; see the 'run' method for details of how to operate the pipeline in non-blocking mode.

The invoking subprogram can also specify functions that will be called when a job completes ('jobCompletionHandler'), and when a group completes ('groupCompletionHandler'). These can perform any specific actions that are required such as sending notification email, setting file ownerships and permissions etc.

class bcftbx.Pipeline.**SolidPipelineRunner**(*runner*, *script*, *max_concurrent_jobs=4*, *poll_interval=30*)

Class to run and manage multiple jobs for Solid data pipelines

Subclass of PipelineRunner specifically for dealing with scripts that take Solid data (i.e. csfasta/qual file pairs).

Defines the addDir method in addition to all methods already defined in the base class; use this method one or more times to specify directories with data to run the script on. The SOLiD data file pairs in each specified directory will be located automatically.

For example:

solid_pipeline = SolidPipelineRunner('qc.sh') solid_pipeline.addDir('/path/to/datadir') solid_pipeline.run()

4.6.2 Functions

bcftbx.Pipeline.**GetSolidDataFiles**(*dirn*, *pattern=None*, *file_list=None*)

Return list of csfasta/qual file pairs in target directory

Note that files with names ending in '_T_F3' will be rejected as these are assumed to come from the preprocess filtering stage.

Optionally also specify a regular expression pattern that file names must also match in order to be included.

Arguments: dirn: name/path of directory to look for files in pattern: optional, regular expression pattern to filter names with file_list: optional, a list of file names to use instead of

fetching a list of files from the specified directory

Returns: List of tuples consisting of two csfasta-qual file pairs (F3 and F5).

bcftbx.Pipeline.**GetFastqFiles**(*dirn*, *pattern=None*, *file_list=None*)

Return list of fastq files in target directory

Optionally also specify a regular expression pattern that file names must also match in order to be included.

Arguments: dirn: name/path of directory to look for files in pattern: optional, regular expression pattern to filter names with file_list: optional, a list of file names to use instead of

fetching a list of files from the specified directory

Returns: List of file-pair tuples.

bcftbx.Pipeline.**GetFastqGzFiles**(*dirn*, *pattern=None*, *file_list=None*)

Return list of fastq.gz files in target directory

Optionally also specify a regular expression pattern that file names must also match in order to be included.

Arguments: dirn: name/path of directory to look for files in pattern: optional, regular expression pattern to filter names with file_list: optional, a list of file names to use instead of

fetching a list of files from the specified directory

Returns: List of file-pair tuples.

4.7 bcftbx.Md5sum

Md5sum

Classes and functions for performing various MD5 checksum operations.

The code function is the 'md5sum' function, which computes the MD5 hash for a file and is based on examples at:

http://www.python.org/getit/releases/2.0.1/md5sum.py

and

http://stackoverflow.com/questions/1131220/get-md5-hash-of-a-files-without-open-it-in-python

Usage:

```
>>> import Md5sum
>>> Md5Sum.md5sum("myfile.txt")
... eacc9c036025f0e64fb724cacaadd8b4
```

This module implements two methods for generating the md5 digest of a file: the first uses a method based on the hashlib module, while the second (used as a fallback for pre-2.5 Python) uses the now deprecated md5 module. Note however that the md5sum function determines itself which method to use.

There is also a high-level class 'Md5Checker' which implements various class methods for running MD5 checks across all files in a directory, and a wrapper class 'Md5Reporter' which

class bcftbx.Md5sum.**Md5CheckReporter**(*results=None*, *verbose=False*, *fp=<open file '<stdout>', mode 'w'>*)

Provides a generic reporting class for Md5Checker methods

Typical usage modes are either:

```
>>> r = Md5CheckReporter()
>>> for f,s in Md5Checker.md5cmp_dirs(d1,d2):
...     r.add_result(f,s)
```

or more concisely:

```
>>> r = Md5CheckReporter(Md5Checker.md5cmp_dirs(d1,d2))
```

Use the 'summary' method to generate a summary of all the checks.

Use the 'status' method to get a single indicator of success or failure which is consistent with UNIX-style return codes.

To find out how many results were processed in total, how many failed etc use the following properties:

n_files : total number of results examined n_ok : number that passed MD5 checks (MD5_OK) n_failed : number that failed due to different MD5 sums (MD5_FAILED) n_missing: number that failed due to a missing target file

(MISSING_TARGET)

n_errors [number that had errors calculating their MD5 sums] (MD5 ERROR)

add_result (*f*, *status*)
Add a result to the reporter

Takes a file and an Md5Checker status code and adds it to the results.

If the status code indicates a failed check then the file name is added to a list corresponding to the nature of the failure (e.g. MD5 sums didn't match, target was missing etc).

n_errors
Number of files with errors checking MD5 sums

n_failed
Number of failed MD5 sum checks

n_files
Total number of files checked

n_missing
Number of missing files

n_ok
Number of passed MD5 sum checks

status
Return status code

Returns 0 if all files that were checked passed the MD5 check, or 1 if at least one file failed the check for whatever reason.

summary()
Write a summary of the results

Writes a summary of the number of files checked, how many passed or failed MD5 checks and so on, to the specified output stream.

class bcftbx.Md5sum.**Md5Checker**
Provides static methods for performing checks using MD5 sums

The Md5Checker class is a collection of static methods that can be used for performing checks using MD5 sums.

It also provides a set of constants to

classmethod compute_md5sums (*d*, *links=0*)
Calculate MD5 sums for all files in directory

Given a directory, traverses the structure underneath (including subdirectories) and yields the path and MD5 sum for each file that is found.

The 'links' option determines how symbolic links are handled, see the 'walk' function for details.

Arguments: dirn: name of the top-level directory links: (optional) specify how symbolic links are handled

Returns: Yields a tuple (f,md5) where f is the path of a file relative to the top-level directory, and md5 is the calculated MD5 sum.

classmethod md5_walk (*dirn*, *links=0*)
Calculate MD5 sums for all files in directory

Given a directory, traverses the structure underneath (including subdirectories) and yields the path and MD5 sum for each file that is found.

The 'links' option determines how symbolic links are handled, see the 'walk' function for details.

Arguments: dirn: name of the top-level directory links: (optional) specify how symbolic links are handled

Returns: Yields a tuple (f,md5) where f is the path of a file relative to the top-level directory, and md5 is the calculated MD5 sum.

classmethod md5cmp_dirs(*d1*, *d2*, *links=0*)

Compares the contents of one directory with another using MD5 sums

Given two directory names 'd1' and 'd2', compares the MD5 sum of each file found in 'd1' against that of the equivalent file in 'd2', and yields the result as an Md5checker constant for each file pair, i.e.:

MD5_OK: if MD5 sums match; MD5_FAILED: if MD5 sums differ.

If the equivalent file doesn't exist then yields MISSING_TARGET.

If one or both MD5 sums cannot be computed then yields MD5_ERROR.

How symbolic links are handled depends on the setting of the 'links' option:

FOLLOW_LINKS: (default) MD5 sums are computed and compared for the targets of symbolic links. Broken links are treated as if the file was missing.

IGNORE_LINKS: MD5 sums are not computed or compared if either file is a symbolic link, and links to directories are not followed.

Arguments: d1: 'reference' directory d2: 'target' directory to be compared with the reference links: (optional) specify how symbolic links are handled.

Returns: Yields a tuple (f,status) where f is the relative path of the file pair being compared, and status is the Md5Checker constant representing the outcome of the comparison.

classmethod md5cmp_files(*f1*, *f2*)

Compares the MD5 sums of two files

Given two file names, attempts to compute and compare their MD5 sums.

If the MD5s match then returns MD5_OK, if they don't match then returns MD5_FAILED.

If one or both MD5 sums cannot be computed then returns MD5_ERROR.

Note that if either file is a link then MD5 sums will be computed for the link target(s), if they exist and can be accessed.

Arguments: f1: name and path for reference file f2: name and path for file to be checked

Returns: Md5Checker constant representing the outcome of the comparison.

classmethod verify_md5sums(*filen=None*, *fp=None*)

Verify md5sums from a file

Given a file (or a file-like object opened for reading), reads each line and attemps to interpret as an md5sum line i.e. of the form

<md5 sum> <path/to/file>

e.g.

66b201ae074c36ae9bffec7fb74ff03a md5checker.py

It then attempts to verify the MD5 sum against the file located on the file system, and yields the result as an Md5checker constant for each file line i.e.:

MD5_OK: if MD5 sums match; MD5_FAILED: if MD5 sums differ.

If the file cannot be found then it yields MISSING_TARGET; if there is a problem computing the MD5 sum then it yields MD5_ERROR.

Arguments: filen: name of the file containing md5sum output fp : file-like object opened for reading, with md5sum output

Returns: Yields a tuple (f,status) where f is the path of the file being verified (as it appears in the file), and status is the Md5Checker constant representing the outcome.

classmethod walk(*dirn*, *links=0*)
Traverse all files found in a directory structure

Given a directory, traverses the structure underneath (including subdirectories) and yields the path for each file that is found.

How symbolic links are handled depends on the setting of the 'links' option:

FOLLOW_LINKS: symbolic links to files are treated as files; links to directories are followed.

IGNORE_LINKS: symbolic links to files are ignored; links to directories are not followed.

Arguments: dirn: name of the top-level directory links: (optional) specify how symbolic links are handled

Returns: Yields the name and full path for each file under 'dirn'.

bcftbx.Md5sum.**hexify**(*s*)
Return the hex representation of a string

bcftbx.Md5sum.**md5sum**(*f*)
Return md5sum digest for a file or stream

This implements the md5sum checksum generation using both the hashlib module (which should be available in Python 2.5) and the deprecated md5 module (which will be used if hashlib is unavailable, as is the case for Python 2.4 and earlier).

The choice of hashlib versus md5 is made automatically and there is no need for the invoking subprogram to decide: the resulting checksums are the same using either library regardless.

Arguments:

f: name of the file to generate the checksum from, or a file-like object opened for reading in binary mode.

Returns: Md5sum digest for the named file.

4.8 bcftbx.platforms

platforms.py

Utilities and data to identify NGS sequencer platforms

bcftbx.platforms.**get_sequencer_platform**(*sequencer_name*)
Attempt to determine platform from sequencer name

Checks the supplied sequencer name against the patterns in PLATFORMS and returns the first match (or None if no match is found).

Arguments:

sequencer_name: sequencer name (can include a leading directory path)

Returns: Matching sequencer platform, or None.

bcftbx.platforms.**list_platforms**()
Return list of known platform names

4.9 bcftbx.TabFile

Classes for working with generic tab-delimited data.

The TabFile module provides a TabFile class, which represents a tab-delimited data file, and a TabDataLine class, which represents a line of data.

4.9.1 Creating a TabFile

TabFile objects can be initialised from existing files:

```
>>> data = TabFile('data.txt')
```

or an 'empty' TabFile can be created if no file name is specified.

Lines starting with '#' are ignored.

4.9.2 Accessing Data within a TabFile

Within a TabFile object each line of data is represented by a TabDataLine object. Lines of data are referenced using index notation, with the first line of data being index zero:

```
>>> line = data[0]
>>> line = data[i]
```

Note that the index is not the same as the line number from the source file, (if one was specified) - this can be obtained from the 'lineno' method of each line:

```
>>> line_number = line.lineno()
```

len() gives the total number of lines of data in the TabFile object:

```
>>> len(data)
```

It is possible to iterate over the data lines in the object:

```
>>> for line in data:
>>>     ... do something with line ...
```

By default columns of data in the file are referenced by index notation, with the first column being index zero:

```
>>> line = data[0]
>>> value = line[0]
```

If column headers are specified then these can also be used to reference columns of data:

```
>>> data = TabFile('data.txt',column_names=['ex','why','zed'])
>>> line = data[0]
>>> ex = line['ex']
>>> line['why'] = 3.454
```

Headers can also be read from the first line of an input file:

```
>>> data = TabFile('data.txt',first_line_is_header=True)
```

A list of the column names can be fetched using the 'header' method:

```
>>> print data.headers()
```

Use the 'str' built-in to get the line as a tab-delimited string:

```
>>> str(line)
```

4.9.3 Adding and Removing Data

New lines can be added to the TabFile object via the 'append' and 'insert' methods:

```
>>> data.append()  # No data i.e. empty line
>>> data.append(data=[1,2,3]) # Provide data values as a list
>>> data.append(tabdata='1	2	3') # Provide values as tab-delimited string
>>> data.insert(1,data=[5,6,7]) # Inserts line of data at index 1
```

Type conversion is automatically performed when data values are assigned:

```
>>> line = data.append(data=['1',2,'3.4','pjb'])
>>> line[0]
1
>>> line[2]
3.4
>>> line[3]
'pjb'
```

Lines can also be removed using the 'del' built-in:

```
>>> del(data[0]) # Deletes first data line
```

New columns can be added using the 'appendColumn' method e.g.:

```
>>> data.appendColumn('new_col') # Creates a new empty column
```

4.9.4 Filtering Data

The 'lookup' method returns a set of data lines where a key matches a specific value:

```
>>> data = TabFile('data.txt',column_names=['chr','start','end'])
>>> chrom = data.lookup('chr','chrX')
```

Within a single data line the 'subset' method returns a list of values for a set of column indices or column names:

```
>>> data = TabFile(column_names=['chr','start','end','strand'])
>>> data.append(data=['chr1',123456,234567,'+'])
>>> data[0].subset('chr1','start')
['chr1',123456]
```

4.9.5 Sorting Data

The 'sort' method offers a simple way of sorting the data lines within a TabFile. The simplest example is sorting on a specific column:

```
>>> data.sort(lambda line: line['start'])
```

See the method documentation for more detail on using the 'sort' method.

4.9.6 Manipulating Data: whole column operations

The 'transformColumn' and 'computeColumn' methods provide a way to update all the values in a column with a single method call. In each case the calling subprogram must supply a function object which is used to update the values in a specific column.

The function supplied to 'transformColumn' must take a single argument which is the current value of the column in that line. For example: define a function to increment a supplied value by 1:

```
>>> def addOne(x):
>>> ...     return x+1
```

Then use this to add one to all values in the column 'start':

```
>>> data.transformColumn('start',addOne)
```

Alternatively a lambda can be used to avoid defining a new function:

```
>>> data.transformColumn('start',lambda x: x+1)
```

The function supplied to 'computeColumn' must take a single argument which is the current line (i.e. a TabDataLine object) and return a new value for the specified column. For example:

```
>>> def calculateMidpoint(line):
>>> ...   return (line['start'] + line['stop'])/2.0
>>> data.computeColumn('midpoint',calculateMidpoint)
```

Again a lambda expression can be used instead:

```
>>> data.computeColumn('midpoint',lambda line: line['stop'] - line['start'])
```

4.9.7 Writing to File

Use the TabFile's 'write' method to output the content to a file:

```
>>> data.write('newfile.txt') # Writes all the data to newfile.txt
```

It's also possible to reorder the columns before writing out using the 'reorderColumns' method.

4.9.8 Specifying Delimiters

It's possible to use a different field delimiter than tabs, by explicitly specifying the value of the 'delimiter' argument when creating a new TabFile object, for example for a comma-delimited file:

```
>>> data = TabFile('data.txt',delimiter=',')
```

class bcftbx.TabFile.**TabDataLine**(*line=None*, *column_names=None*, *delimiter='t'*, *lineno=None*, *convert=True*)

Class to store a line of data from a tab-delimited file

Values can be accessed by integer index or by column names (if set), e.g.

line = TabDataLine("1 2 3",('first','second','third'))

allows the 2nd column of data to accessed either via line[1] or line['second'].

Values can also be changed, e.g.

line['second'] = new_value

Values are automatically converted to integer or float types as appropriate.

Subsets of data can be created using the 'subset' method.

Line numbers can also be set by the creating subprogram, and queried via the 'lineno' method.

It is possible to use a different field delimiter than tabs, by explicitly specifying the value of the 'delimiter' argument, e.g. for a comma-delimited line:

line = TabDataLine("1,2,3",delimiter=',')

Check if a line is empty:

if not line: print "Blank line"

append(**values*)
Append values to the data line

Should only be used when creating new data lines.

appendColumn(*key*, *value*)
Append keyed values to the data line

This adds a new value along with a header name (i.e. key)

convert_to_str(*value*)
Convert value to string

convert_to_type(*value*)
Internal: convert a value to the correct type

Used to coerce input values into integers or floats if appropriate before storage in the TabDataLine object.

delimiter(*new_delimiter=None*)
Set and get the delimiter for the line

If 'new_delimiter' is not None then the field delimiter for the line will be updated to the supplied value. This affects how lines are represented via the __repr__ built-in.

Returns the current value of the delimiter.

lineno()
Return the line number associated with the line

NB The line number is set by the class or function which created the TabDataLine object, it is not guaranteed by the TabDataLine class itself.

subset(**keys*)
Return a subset of data items

This method creates a new TabDataLine instance with a subset of data specified by the 'keys' argument, e.g.

new_line = line.subset(2,1)

returns an instance with only the 2nd and 3rd data values in reverse order.

To access the items in a subset using index notation, use the same keys as those specified when the subset was created. For example, for

s = line.subset("two","nine")

use s["two"] and s["nine"] to access the data; while for

s = line.subset(2,9)

use s[2] and s[9].

Arguments:

keys: one or more keys specifying columns to include in the subset. Keys can be column indices, column names, or a mixture, and the same column can be referenced multiple times.

class bcftbx.TabFile.**TabFile**(*filen=None*, *fp=None*, *column_names=None*, *skip_first_line=False*, *first_line_is_header=False*, *tab_data_line=<class bcftbx.TabFile.TabDataLine>*, *delimiter='t'*, *convert=True*)

Class to get data from a tab-delimited file

Loads data from the specified file into a data structure than can then be queried on a per line and per item basis.

Data lines are represented by data line objects which must be TabDataLine-like.

Example usage:

data = TabFile(myfile) # load initial data

print '%s' % len(data) # report number of lines of data

print '%s' % data.header() # report header (i.e. column names)

for line in data: ... # loop over lines of data

myline = data[0] # fetch first line of data

append(*data=None*, *tabdata=None*, *tabdataline=None*)

Create and append a new data line

Creates a new data line object and appends it to the end of the list of lines.

Optionally the 'data' or 'tabdata' arguments can specify data items which will be used to populate the new line; alternatively 'tabdataline' can provide a TabDataLine-based object to be appended.

If none of these are specified then a default blank TabDataLine-based object is created, appended and returned.

Arguments: data: (optional) a list of data items tabdata: (optional) a string of tab-delimited data items tabdataline: (optional) a TabDataLine-based object

Returns: Appended data line object.

appendColumn(*name*)

Append a new (empty) column

Arguments: name: name for the new column

computeColumn(*column_name*, *compute_func*)

Compute and store values in a new column

For each line of data the computation function will be invoked with the line as the sole argument, and the result will be stored in a new column with the specified name.

Arguments:

column_name: name or index of column to write transformation result to

compute_func: callable object that will be invoked to perform the computation

filename()
Return the file name associated with the TabFile

header()
Return list of column names

If no column names were set then this will be an empty list.

indexByLineNumber(*n*)
Return index of a data line given the file line number

Given the line number n for a line in the original file, returns the index required to access the data for that line in the TabFile object.

If no matching line is found then raises an IndexError.

insert(*i*, *data=None*, *tabdata=None*, *tabdataline=None*)
Create and insert a new data line at a specified index

Creates a new data line object and inserts it into the list of lines at the specified index position 'i' (nb NOT a line number).

Optionally the 'data' or 'tabdata' arguments can specify data items which will be used to populate the new line; alternatively 'tabdataline' can provide a TabDataLine-based object to be inserted.

Arguments: i: index position to insert the line at data: (optional) a list of data items tabdata: (optional) a string of tab-delimited data items tabdataline: (optional) a TabDataLine-based object

Returns: New inserted data line object.

lookup(*key*, *value*)
Return lines where the key matches the specified value

nColumns()
Return the number of columns in the file

If the file had a header then this will be the number of header columns; otherwise it will be the number of columns found in the first line of data

reorderColumns(*new_columns*)
Rearrange the columns in the file

Arguments:

new_columns: list of column names or indices in the new order

Returns: New TabFile object

sort(*sort_func*, *reverse=False*)
Sort data using arbitrary function

Performs an in-place sort based on the suppled sort_func.

sort_func should be a function object which takes a data line object as input and returns a single numerical value; the data lines will be sorted in ascending order of these values (or descending order if reverse is set to True).

To sort on the value of a specific column use e.g.

```
>>> tabfile.sort(lambda line: line['col'])
```

Arguments:

sort_func: function object taking a data line object as input and returning a single numerical value

reverse: (optional) Boolean, either False (default) to sort in ascending order, or True to sort in descending order

transformColumn(*column_name*, *transform_func*)
Apply arbitrary function to a column

For each line of data the transformation function will be invoked with the value of the named column, with the result being written back to that column (overwriting the existing value).

Arguments: column_name: name of column to write transformation result to transform_func: callable object that will be invoked to perform

the transformation

transpose()
Transpose the contents of the file

Returns: New TabFile object

write(*filen=None*, *fp=None*, *include_header=False*, *no_hash=False*, *delimiter=None*)
Write the TabFile data to an output file

One of either the 'filen' or 'fp' arguments must be given, specifying the file name or stream to write the TabFile data to.

Arguments:

filen: (optional) name of file to write to; ignored if fp is also specified

fp: (optional) a file-like object opened for writing; used in

preference to filen if set to a non-null value Note that the calling program must close the stream in these cases.

include_header: (optional) if set to True, the first line will be a 'header' line

no_hash: (optional) if set to True and include_header is also True then don't put a hash character '#' at the start of the header line in the output file.

delimiter: (optional) delimiter to use when writing data values to file (defaults to the delimiter specified on input)

4.10 bcftbx.simple_xls and bcftbx.Spreadsheet

4.10.1 simple_xls

Simple spreadsheet module intended to provide a nicer programmatic interface to Excel spreadsheet generation.

It is currently built on top of SpreadSheet.py, which itself uses the xlwt, xlrd and xlutils modules. In future the relevant parts may be rewritten to remove the dependence on Spreadsheet.py and call the appropriate xl* classes and functions directly.

Example usage

Start by making a workbook, represented by an XLSWorkBook object:

```
>>> wb = XLSWorkBook("Test")
```

Then add worksheets to this:

```
>>> wb.add_work_sheet('test')
>>> wb.add_work_sheet('data',"My Data")
```

Worksheets have an id and an optional title. Ids must be unique and can be used to fetch the XLSWorkSheet object that represent the worksheet:

```
>>> data = wb.worksheet['data']
```

Cells can be addressed directly using various notations:

```
>>> data['A1'] = "Column 1"
>>> data['A']['1'] = "Updated value"
>>> data['AZ']['3'] = "Another value"
```

The extent of the sheet is defined by the outermost populated rows and columns

```
>>> data.last_column # outermost populated column
>>> data.last_row    # outermost populated row
```

There are various other methods for returning the next row or column; see the documentation for the XLSWorkSheet class.

Data can be added cell-wise (i.e. referencing individual cells as above), row-wise, column-wise and block-wise.

Column-wise operations include inserting a column (shifting columns above it along one to make space):

```
>>> data.insert_column('B',data=['hello','goodbye','whatev'])
```

Append a column (writing data to the first empty column at the end of the sheet):

```
>>> data.append_column(data=['hello','goodbye','whatev'])
```

Write data to a column, overwriting any existing values:

```
>>> data.write_column(data=['hello','goodbye','whatev'])
```

Data can be specified as a list, text or as a single value which is repeated for each cell (i.e. a "fill" value).

Similar row-wise operations also exist:

```
>>> data.insert_row(4,data=['Dozy','Beaky','Mick','Titch'])
>>> data.append_row(data=['Dozy','Beaky','Mick','Titch'])
>>> data.write_row(4,data=['Dozy','Beaky','Mick','Titch'])
```

Block-wise data can be added via a tab and newline-delimited string:

```
>>> data.insert_block_data("This        is              some
        random
        data")
>>> data.insert_block_data("This        is              some
```

(continues on next page)

(continued from previous page)

```
        MORE    random
        data",
...                         col='M',row=7)
```

Formulae can be specified by prefixing a '=' symbol to the start of the cell contents, e.g.:

```
>>> data['A3'] = '=A1+A2'
```

'?' and '#' are special characters that can be used to indicate 'current row' and 'current column' respectively, e.g.:

```
>>> data.fill_column('A','=B?+C?') # evaluates to 'B1+C1' (A1), 'B2+C2' (A2) etc
```

Styling and formatting information can be associated with a cell, either when adding column, row or block data or by using the 'set_style' method. In each case the styling information is passed via an XLSStyle object, e.g.

```
>>> data.set_style(XLSStyle(number_format=NumberFormats.PERCENTAGE),'A3')
```

The workbook can be saved to file:

```
>>> wb.save_as_xls('test.xls')
```

Alternatively the contents of a sheet (or a subset) can be rendered as text:

```
>>> data.render_as_text(include_columns_and_rows=True,
...                     eval_formulae=True,
...                     include_styles=True)
>>> data.render_as_text(start='B1',end='C6',include_columns_and_rows=True)
```

class bcftbx.simple_xls.**CellIndex**(*idx*)
Convenience class for handling XLS-style cell indices

The CellIndex class provides a way of handling XLS-style cell indices i.e. 'A1', 'BZ112' etc.

Given a putative cell index it extracts the column and row which can then be accessed via the 'column' and 'row' attributes respectively.

The 'is_full' property reports whether the supplied index is actually a 'full' index with both column and row specifiers. If it is just a column or just a row then only the appropriate 'column' or 'row' attributes will be set.

is_full
Return True if index has both column and row information

class bcftbx.simple_xls.**ColumnRange**(*i*, *j=None*, *include_end=True*, *reverse=False*)
Iterator for a range of column indices

Range-style iterator for iterating over alphabetical column indices, e.g.

```
>>> for c in ColumnRange('A','Z'):
...   print c
```

next()
Implements Iterator subclass 'next' method

class bcftbx.simple_xls.**Limits**
Limits for XLS files (kept for backwards compatibility)

class bcftbx.simple_xls.**XLSColumn**(*column_index*, *parent=None*)
Class representing a column in a XLSWorkSheet

An XLSColumn object provides access to data in a column from a XLSWorkSheet object. Typically one can be returned by doing something like:

```
>>> colA = ws['A']
```

and individual cell values then accessed by row number alone, e.g.:

```
>>> value = colA['1']
>>> colA['2'] = "New value"
```

full_index(*row*)
Return the full index for a cell in the column

Given a row index, returns the index of the cell that this addresses within the column (e.g. if the column is 'A' then row 2 addresses cell 'A2').

class bcftbx.simple_xls.**XLSLimits**
Limits for XLS files

class bcftbx.simple_xls.**XLSStyle**(*bold=False*, *color=None*, *bgcolor=None*, *wrap=False*, *border=None*, *number_format=None*, *font_size=None*, *centre=False*, *shrink_to_fit=False*)
Class representing a set of styling and formatting data

An XLSStyle object represents a collection of data used for styling and formatting cell values on output to an Excel file.

The style attributes can be set on instantiation, or queried and modified afterwards.

The attributes are:

bold: whether text is bold or not (boolean) color: text color (name) bgcolor: background color (name) wrap: whether text in a cell should wrap (boolean) border: style of cell border (thick, medium, thin etc) number_format: a format code from the NumbersFormat class font_size: font size in points (integer) centre: whether text is centred in the cell (boolean) shrink_to_fit: whether to shrink cell to fit the contents.

The 'name' property can be used to generate a name for the style based on the attributes that have been set, for example:

```
>>> XLSStyle(bold=true).name
... '__bold__'
```

excel_number_format
Return an Excel-style equivalent of the stored number format

Returns an Excel-style number format, or None if the format isn't set or is unrecognised.

name
Return a name based on the attributes

style(*item*)
Wrap 'item' with <style...>...</style> tags

Given a string (or object that can be rendered as a string) return the string representation surrounded by <style...> </style> tags, where the tag attributes describe the style information stored in the XLSStyle object:

font=bold color=(color) bgcolor=(color) wrap border=(border) number_format=(format) font_size=(size) centre shrink_to_fit

class bcftbx.simple_xls.**XLSWorkBook**(*title=None*)
Class for creating an Excel (xls) spreadsheet

An XLSWorkBook instance provides an interface to creating an Excel spreadsheet.

It consists of a collection of XLSWorkSheet objects, each of which represents a sheet in the workbook.

Sheets are created and appended using the add_work_sheet method:

```
>>> xls = XLSWorkBook()
>>> sheet = xls('example')
```

Sheets are kept in the 'worksheet' property and can be acquired by name:

```
>>> sheet = xls.worksheet['example']
```

Once the worksheet(s) have been populated an XLS file can be created using the 'save_as_xls' method:

```
>>> xls.save_as_xls('example.xls')
```

add_work_sheet (*name*, *title=None*)
Create and append a new worksheet

Creates a new XLSWorkSheet object and appends it to the workbook.

Arguments: name: unique name for the worksheet title: optional, title for the worksheet - defaults to the name.

Returns: New XLSWorkSheet object.

save_as_xls (*filen*)
Output the workbook contents to an Excel-format file

Arguments: filen: name of the file to write the workbook to.

save_as_xlsx (*filen*)
Output the workbook contents to an XLSX-format file

Arguments: filen: name of the file to write the workbook to.

class bcftbx.simple_xls.**XLSWorkSheet** (*title*)
Class for creating sheets within an XLS workbook.

XLSWorkSheet objects represent a sheet within an Excel workbook.

Cells are addressed within the sheet using Excel notation i.e. <column><row> (columns start at index 'A' and rows at '1', examples are 'A1' or 'D19'):

```
>>> ws = XLSWorkSheet('example')
>>> ws['A1'] = 'some data'
>>> value = ws['A1']
```

If there is no data stored for the cell then 'None' is returned. Any cell can addressed without errors.

Data can also be added column-wise, row-wise or as a "block" of tab- and new-line delimited data:

```
>>> ws.insert_column_data('B',[1,2,3])
>>> ws.insert_row_data(4,['x','y','z'])
>>> ws.insert_block_data("This      is
```

the data")

A column can be "filled" with a single repeating value:

```
>>> ws.fill_column('D','single value')
```

The extent of the sheet can be determined from the 'last_column' and last_row' properties; the 'next_column' and 'next_row' properties report the next empty column and row respectively.

Cells can contain Excel-style formula by adding an equals sign to the start of the value. Typically formulae reference other cells and perform mathematical operations on them, e.g.:

```
>>> ws['E11'] = "=A1+A2"
```

Wildcard characters can be used which will be automatically translated into the cell column ('#') or row ('?'), for example:

```
>>> ws['F46'] = "=#47+#48"
```

will be transformed to "=F47+F48".

Styles can be applied to cells, using either the 'set_style' method or via the 'style' argument of some methods, to associate an XLSStyle object. Associated XLSStyle objects can be retrieved using the 'get_style' method.

The value of an individual cell can be 'rendered' for output using the 'render_cell' method:

```
>>> print ws.render_cell('F46')
```

All or part of the sheet can be rendered as a tab- and newline-delimited string by using the 'render_as_text' method:

```
>>> print ws.render_as_text()
```

append_column(*data=None*, *text=None*, *fill=None*, *from_row=None*, *style=None*)
Create a new column at the end of the sheet

Appends a new column at the end of the worksheet i.e. in the first available empty column.

By default the appended column is empty, however data can be specified to populate the column.

Arguments:

data: optional, list of data items to populate the inserted column

text: optional, tab-delimited string of text to be used to populate the inserted column

fill: optional, single data item to be repeated to fill the inserted column

from_row: optional, if specified then inserted column is populated from that row onwards

style: optional, an XLSStyle object to associate with the data being inserted

Returns: The index of the appended column.

append_row(*data=None*, *text=None*, *fill=None*, *from_column=None*, *style=None*)
Create a new row at the end of the sheet

Appends a new row at the end of the worksheet i.e. in the first available empty row.

By default the appended row is empty, however data can be specified to populate the row.

Arguments:

data: optional, list of data items to populate the inserted row

text: optional, newline-delimited string of text to be used to populate the inserted row

fill: optional, single data item to be repeated to fill the inserted row

from_row: optional, if specified then inserted row is populated from that column onwards

style: optional, an XLSStyle object to associate with the data being inserted

Returns: The index of the inserted row.

column_is_empty(*col*)
Determine whether a column is empty

Returns False if any cells in the column are populated, otherwise returns True.

columnof(*s*, *row=1*)
Return column index for cell which matches string

Return index of first column where the content matches the specified string 's'.

Arguments: s: string to search for row: row to search in (defaults to 1)

Returns: Column index of first matching cell. Raises LookUpError if no match is found.

fill_column(*column*, *item*, *start=None*, *end=None*, *style=None*)
Fill a column with a single repeated data item

A single data item is inserted into all rows in the specified column which have at least one data item already in any column in the worksheet. A different range of rows can be specified via the 'start' and 'end' arguments.

*** THIS METHOD IS DEPRECATED ***

Consider using insert_column, append_column or write_data.

Arguments: column: index of column to insert the item into (e.g. 'A','MZ') item: data item to be repeated start: (optional) first row to insert data into end: (optional) last row to insert data into style: (optional) XLSStyle object to be associated with each

cell that has data inserted into it

get_style(*idx*)
Return the style information associated with a cell

Returns an XLSStyle object associated with the specific cell.

If no style was previously associated then return a new XLSStyle object.

Arguments: idx: cell index e.g 'A1'

Returns: XLSStyle object.

insert_block_data(*data*, *col=None*, *row=None*, *style=None*)
Insert data items from a block of text

Data items are supplied via a block of tab- and newline-delimited text. Each tab-delimited item is inserted into the next column in a row; newlines indicate that subsequent items are inserted into the next row.

By default items are inserted starting from cell 'A1'; a different starting cell can be explicitly specified via the 'col' and 'row' arguments.

Arguments: data: block of tab- and newline-delimited data col: (optional) first column to insert data into row: (optional) first row to insert data into style: (optional) XLSStyle object to be associated with each

cell that has data inserted into it

insert_column(*position*, *data=None*, *text=None*, *fill=None*, *from_row=None*, *style=None*)
Create a new column at the specified column position

Inserts a new column at the specified column position, pushing up the column currently at that position plus all higher positioned columns.

By default the inserted column is empty, however data can be specified to populate the column.

Arguments:

position: column index specifying position to insert the column at

data: optional, list of data items to populate the inserted column

text: optional, tab-delimited string of text to be used to populate the inserted column

fill: optional, single data item to be repeated to fill the inserted column

from_row: optional, if specified then inserted column is populated from that row onwards

style: optional, an XLSStyle object to associate with the data being inserted

Returns: The index of the inserted column.

insert_column_data(*col*, *data*, *start=None*, *style=None*)
Insert list of data into a column

Data items are supplied as a list, with each item in the list being inserted into the next row in the column.

By default items are inserted starting from row 1, unless a starting row is explicitly specified via the 'start' argument.

*** THIS METHOD IS DEPRECATED ***

Consider using insert_column, append_column or write_data.

Arguments: col: index of column to insert the data into (e.g. 'A','MZ') data: list of data items start: (optional) first row to insert data into style: (optional) XLSStyle object to be associated with each

cell that has data inserted into it

insert_row(*position*, *data=None*, *text=None*, *fill=None*, *from_column=None*, *style=None*)
Create a new row at the specified row position

Inserts a new row at the specified row position, pushing up the row currently at that position plus all higher positioned row.

By default the inserted row is empty, however data can be specified to populate the column.

Arguments:

position: row index specifying position to insert the row at

data: optional, list of data items to populate the inserted row

text: optional, newline-delimited string of text to be used to populate the inserted row

fill: optional, single data item to be repeated to fill the inserted row

from_row: optional, if specified then inserted row is populated from that column onwards

style: optional, an XLSStyle object to associate with the data being inserted

Returns: The index of the inserted row.

insert_row_data(*row*, *data*, *start=None*, *style=None*)
Insert list of data into a row

Data items are supplied as a list, with each item in the list being inserted into the next column in the row.

By default items are inserted starting from column 'A', unless a starting column is explicitly specified via the 'start' argument.

*** THIS METHOD IS DEPRECATED ***

Consider using insert_row, append_row or write_row.

Arguments: row: index of row to insert the data into (e.g. 1, 112) data: list of data items start: (optional) first column to insert data into style: (optional) XLSStyle object to be associated with each

cell that has data inserted into it

last_column
Return index of last column with data

last_row
Return index of last row with data

next_column
Index of first empty column after highest index with data

next_row
Index of first empty row after highest index with data

render_as_text(*include_columns_and_rows=False*, *include_styles=False*, *eval_formulae=False*, *apply_format=False*, *start=None*, *end=None*)
Text representation of all or part of the worksheet

All or part of the sheet can be rendered as a tab- and newline-delimited string.

Arguments:

include_columns_and_rows: (optional) if True then also output a header row of column indices, and a column of row indices (default is to not output columns and rows).

include_styles: (optional) if True then also render the styling information associated with the cell (default is not to apply styling).

apply_format: (optional) if True then format numbers according to the formatting information associated with the cell (default is not to apply formatting).

eval_formulae: (optional) if True then if the cell contains a formula, attempt to evaluate it and return the result. Otherwise return the formula itself (this is the default)

start: (optional) specify the top-lefthand most cell index to start rendering from (default is 'A1').

end: (optional) specify the bottom-righthand most cell index to finish rendering at (default is the cell corresponding to the highest column and row indices. Note that this cell may be empty.)

Returns: String containing the rendered sheet or sheet subset, with items within a row separated by tabs, and rows separated by newlines.

render_cell(*idx*, *eval_formulae=False*, *apply_format=False*)
Text representation of value stored in a cell

Create a text representation of a cell's contents. If the cell contains a formula then '?'s will be replaced with the row index and '#'s with the column index. Optionally the formula can also be evaluated, and any style information associated with the cell can also be rendered.

Arguments: idx: cell index e.g. 'A1' eval_formulae: (optional) if True then if the cell contains

a formula, attempt to evaluate it and return the result. Otherwise return the formula itself (this is the default)

apply_format: (optional) if True then format numbers according to the formatting information associated with the cell (default is not to apply formatting).

Returns: String representing the cell contents.

row_is_empty(*row*)
Determine whether a row is empty

Returns False if any cells in the row are populated, otherwise returns True.

rowof(*s*, *column='A'*)
Return row index for cell which matches string

Return index of first row where the content matches the specified string 's'.

Arguments: s: string to search for column: column to search in (defaults to 'A')

Returns: Row index of first matching cell. Raises LookUpError if no match is found.

set_style(*cell_style*, *start*, *end=None*)
Associate style information with one or more cells

Associates a specified XLSStyle object with a single cell, or with a range of cells (if a second cell index is supplied).

The style associated with a cell can be fetched using the 'get_style' method.

Arguments: cell_style: XLSStyle object start: cell index e.g. 'A1' end: (optional) second cell index; together with

'start' this defines a range of cells to associate the style with.

write_column(*col*, *data=None*, *text=None*, *fill=None*, *from_row=None*, *style=None*)
Write data to rows in a column

Data can be specified as a list, a newline-delimited string, or as a single repeated data item.

Arguments:

data: optional, list of data items to populate the inserted column

text: optional, newline-delimited string of text to be used to populate the inserted column

fill: optional, single data item to be repeated to fill the inserted column

from_row: optional, if specified then inserted column is populated from that row onwards

style: optional, an XLSStyle object to associate with the data being inserted

write_row(*row*, *data=None*, *text=None*, *fill=None*, *from_column=None*, *style=None*)
Write data to rows in a column

Data can be specified as a list, a tab-delimited string, or as a single repeated data item.

Arguments: row: row index specifying which row data: optional, list of data items to populate the

inserted row

text: optional, tab-delimited string of text to be used to populate the inserted row

from_column: optional, if specified then inserted row is populated from that column onwards

style: optional, an XLSStyle object to associate with the data being inserted

class bcftbx.simple_xls.**XLSXLimits**
Limits for XLSX files

bcftbx.simple_xls.**cell**(*col*, *row*)
Return XLS cell index for column and row

E.g. cell('A',3) returns 'A3'

bcftbx.simple_xls.**cmp_column_indices**(*x*, *y*)
Comparision function for column indices

x and y are XLS-style column indices e.g. 'A', 'B', 'AA' etc.

Returns -1 if x is a column index less than y, 1 if it is greater than y, and 0 if it's equal.

bcftbx.simple_xls.**column_index_to_integer**(*col*)
Convert XLS-style column index into equivalent integer

Given a column index e.g. 'A', 'BZ' etc, converts it to the integer equivalent using zero-based counting system (so 'A' is equivalent to zero, 'B' to 1 etc).

bcftbx.simple_xls.**column_integer_to_index**(*idx*)
Convert integer column index to XLS-style equivalent

Given an integer index, converts it to the XLS-style equivalent e.g. 'A', 'BZ' etc, using a zero-based counting system (so zero is equivalent to 'A', 1 to 'B' etc).

bcftbx.simple_xls.**convert_to_number**(*s*)
Convert a number to float or int as appropriate

Raises ValueError if neither conversion is possible.

bcftbx.simple_xls.**eval_formula**(*item*, *worksheet*)
Evaluate a formula using the contents of a worksheet

Given an item, attempts an Excel-style evaluation.

If the item doesn't start with '=' then it is returned as-is. Otherwise the function attempts to evaluate the formula, including looking up (and if necessary also evaluating) the contents of any cells that are referenced.

*** Note that the implementation of the evaluation is very** simplistic and cannot handle complex formulae or functions

Currently it can only deal with:

- basic mathematical operations (+-*/)

bcftbx.simple_xls.**format_value**(*value*, *number_format=None*)
Format a cell value based on the specified number format

bcftbx.simple_xls.**incr_col**(*col*, *incr=1*)
Return column index incremented by specific number of positions

Arguments: col: index of column to be incremented incr: optional, number of cells to shift by. Can be negative to go backwards. Defaults to 1 i.e. next column along.

bcftbx.simple_xls.**is_float**(*s*)
Test if a number is a float

bcftbx.simple_xls.**is_int**(*s*)
Test if a number is an integer

4.10.2 Spreadsheet

Provides classes for writing data to an Excel spreadsheet, using the 3rd party modules xlrd, xlwt and xlutils.

The basic classes are 'Workbook' (representing an XLS spreadsheet) and 'Worksheet' (representing a sheet within a workbook). There is also a 'Spreadsheet' class which is built on top of the other two classes and offers a simplified interface to writing line-by-line XLS spreadsheets.

Simple usage examples

1. Writing a new XLS spreadsheet using the Workbook class

```
>>> wb = Workbook()
>>> ws = wb.addSheet('test1')
>>> ws.addText("Hello   Goodbye
Goodbye Hello")
>>> wb.save('test2.xls')
```

2. Appending to an existing XLS spreadsheet using the Workbook class

```
>>> wb = Workbook('test2.xls')
>>> ws = wb.getSheet('test1')
>>> ws.addText("Some more data for you")
>>> ws = wb.addSheet('test2')
>>> ws.addText("<style font=bold bgcolor=gray25>Hahahah</style>")
>>> wb.save('test3.xls')
```

3. Creating or appending to an XLS spreadsheet using the Spreadsheet class

```
>>> wb = Spreadsheet('test.xls','test')
>>> wb.addTitleRow(['File','Total reads','Unmapped reads'])
>>> wb.addEmptyRow()
>>> wb.addRow(['DR_1',875897,713425])
>>> wb.write()
```

Module constants

MAX_LEN_WORKSHEET_TITLE: maximum length allowed by xlwt for worksheet titles
MAX_LEN_WORKSHEET_CELL_VALUE: maximum number of characters allowed for cell value
MAX_NUMBER_ROWS_PER_WORKSHEET: maximum number of rows allowed per worksheet by xlwt

Dependencies

The Spreadsheet module depends on the xlwt, xlrd and xlutils libraries which can be found at:

- http://pypi.python.org/pypi/xlwt/0.7.2
- http://pypi.python.org/pypi/xlrd/0.7.1
- http://pypi.python.org/pypi/xlutils/1.4.1

Note that xlutils also needs functools: http://pypi.python.org/pypi/functools

but if you're using Python<2.5 then you need a backported version of functools, try:

https://github.com/dln/pycassa/blob/90736f8146c1cac8287f66e8c8b64cb80e011513/pycassa/py25_functools.py

class bcftbx.Spreadsheet.**Spreadsheet**(*name*, *title*)

Class for creating and writing a spreadsheet.

This creates a very simple single-sheet workbook.

addEmptyRow(*color=None*)
Add an empty row to the spreadsheet.

Inserts an empty row into the next position in the spreadsheet.

Arguments: color: optional background color for the empty row

Returns: Integer index of (empty) row just written

addRow(*data*, *set_widths=False*, *bold=False*, *wrap=False*, *bg_color=''*)
Add a row of data to the spreadsheet.

Arguments: data: list of data items to be added.

set_widths: (optional) Boolean; if True then set the column width to the length of the cell contents for each cell in the new row

bold: (optional) use bold font for cells

wrap: (optional) wrap the cell content

bg_color: (optional) set the background color for the cell

Returns: Integer index of row just written

addTitleRow(*headers*)
Add a title row to the spreadsheet.

The title row will have the font style set to bold for all cells.

Arguments: headers: list of titles to be added.

Returns: Integer index of row just written

write()
Write the spreadsheet to file.

class bcftbx.Spreadsheet.**Styles**
Class for creating and caching EasyXfStyle objects.

XLS files have a limit of 4,000 styles, so cache and reuse EasyXfStyle objects to avoid exceeding this limit.

getXfStyle(*bold=False*, *wrap=False*, *color=None*, *bg_color=None*, *border_style=None*, *num_format_str=None*, *font_size=None*, *centre=False*, *shrink_to_fit=False*)
Return EasyXf object to apply styles to spreadsheet cells.

Arguments: bold: indicate whether font should be bold face wrap: indicate whether text should wrap in the cell color: set text colo(u)r bg_color: set colo(u)r for cell background. border_style: set line type for cell borders (thin, medium, thick, etc) font_size: font size (in points) centre: centre the cell content horizontally shrink_to_fit: shrink cell to fit contents

Note that colours must be a valid name as recognised by xlwt.

class bcftbx.Spreadsheet.**Workbook**(*xls_name=''*)
Class for writing data to an XLS spreadsheet.

A Workbook represents an XLS spreadsheet, which conists of sheets (represented by Worksheet instances).

addSheet(*title*, *xlrd_sheet=None*, *xlrd_index=None*)
Add a new sheet to the spreadsheet.

Arguments: title: title for the sheet xlrd_sheet: (optional) an xlrd sheet from an existing XLS

workbook.

getSheet(*title*)
Retrieve a sheet from the spreadsheet.

save(*xls_name*)
Finish adding data and write the spreadsheet to disk.

Note that for a spreadsheet based on an existing XLS file, this doesn't have to be the same name.

Arguments:

xls_name: the file name to write the spreadsheet to. Note that if a file already exists with this name then it will be overwritten.

class bcftbx.Spreadsheet.**Worksheet**(*workbook*, *title*, *xlrd_index=None*, *xlrd_sheet=None*)
Class for writing to a sheet in an XLS spreadsheet.

A Worksheet object represents a sheet in an XLS spreadsheet.

Data can be inserted into the worksheet in a variety of ways:

- addTabData: a Python list of tab-delimited lines; each line forms a line in the output XLS, with each field forming a column.
- addText: a string representing arbitrary text, with newlines delimiting lines and tabs (if any) in each line delimiting fields.

Each can be called multiple times in any order on the same spreadsheet before it is saved, and the data will be appended.

For new Worksheet objects (i.e. those which weren't read from a pre-existing XLS file), it is also possible to insert new columns:

- insertColumn: if a single value is specified then all columns are filled with that value; alternatively a list of values can be supplied which are written one-per-row.

Formulae can be specified using a variation on Excel's '=' notation, e.g.

=A1+B2

adds the values from cells A1 and B2 in the final spreadsheet.

Formulae are written directly as supplied unless they contain special characters '?' (indicates the current line number) or '#' (indicates the current column).

Using '?' allows simple row-wise formulae to be added, e.g.

=A?+B?

will be converted to substitute the row index (e.g. '=A1+B1' for row 1, '=A2+B2' for row 2 etc).

Using '#' allows simple column-wise formulae to be added, e.g.

=#1-#2

will be converted to substitue the column id (e.g. '=A1-A2' for column A, '=B1-B2' for column B etc).

Note that the substitution occurs when the spreadsheet is saved.

Individual items can have basic styles applied to them by wrapping them in <style ...>...</style> tags. Within the leading style tag the following attributes can be specified:

font=bold (sets bold face) color=<color> (sets the text colour) bgcolor=<color> (sets the background colour) border=<style> (sets the cell border style to 'thin', 'medium', 'thick' etc) wrap (specifies that text should wrap) number_format=<format_string> (specifies how to display numbers, see below) font_height=<height> (sets font size in points) centre (specifies that text should be centred) shrink_to_fit (specifies that cells should shrink to fit their contents)

For example <style font=bold bgcolor=gray25>...</style>

Note that styles can also be applied to formulae.

The 'number_format' style attribute allows the calling program to specify how numbers should be displayed, for example:

number_format=0.00 (displays values to 2 decimal places) number_format=0.0% (displays values as percentages to 1 decimal place) number_format=#,### (displays values with , as the delimiter for thousands)

The spreadsheet data is held internally as a list of rows, with each row represented by a tab-delimited string.

addTabData(*rows*)
Write a list of tab-delimited data rows to the sheet.

Given a list of rows with tab-separated data items, append the data to the worksheet.

Arguments:

data: Python list representing rows of tab-separated data items

addText(*text*)
Append and populate rows from text.

Given some arbitrary text as a string, the data it contains will be appended to the worksheet using newlines to indicate multiple rows and tabs to delimit data items.

This method is useful for turning tab-delimited data read from a CSV-type file into a spreadsheet.

Arguments:

text: a string representing the data to add: rows are delimited by newlines, and items by tabs

column_id_from_index(*i*)
Get XLS column id from index of column

cindex is the zero-based column index (an integer); this method returns the matching XLS column identifier (i.e. 'A', 'B', 'AA', 'BA' etc).

freezePanes(*row=None*, *column=None*)
Split panes and mark as frozen

'row' and 'column' are integer indices specifying the cell which defines the pane to be marked as frozen

getColumnId(*name*)
Lookup XLS column id from name of column.

If there is no data, or if the name isn't in the header row of the data, then an exception is raised.

Returns the column identifier (i.e. 'A', 'B' etc) for the column with the matching name.

insertColumn(*position*, *insert_items=None*, *title=None*)
Insert a new column into the spreadsheet.

This inserts a new column into each row of data, at the specified positional index (starting from 0).

Note: at present columns can only be inserted into worksheets that have been created from scratch via Worksheet class (i.e. cannot insert into an existing worksheet read in from a file).

Arguments:

position: positional index for the column to be inserted at (0=A, 1=B etc)

title: (optional) value to be written to the first row (i.e. a column title)

insert_items: value(s) to be inserted; either a single item, or a list of items. Each item can be blank, a constant value, or a formula.

save()
Write the new data to the spreadsheet.

setCellValue(*row*, *col*, *value*)
Set the value of a cell

Given row and column coordinates (using integer indices starting from zero for both), replace the existing value with a new one.

The new value can include style information.

Arguments: row: integer row index (starting at zero) col: integer column index (starting at zero, i.e. 0=A, 1=B etc) value: new value to be written into the cell

4.11 bcftbx.cmdparse

Provides a CommandParser class for handling command lines of the form:

```
PROG COMMAND OPTION ARGS
```

where different sets of options can be defined based on the initial command.

The CommandParser can support arbitrary 'subparser backends' which are created to parse the ARGS list for each defined COMMAND. The default subparser is the 'optparse.OptionParser' class, but this can be swapped for arbitrary subparser (for example, the 'argparse.ArgumentParser' class) when the CommandParser is created.

In addition to the core CommandParser class, there are a number of supporting functions that can be used with any optparse- or argparse-based parser instance, to add the following 'standard' options:

- –nprocessors
- –runner
- –no-save
- –dry-run
- –debug

class bcftbx.cmdparse.**CommandParser**(*description=None*, *version=None*, *subparser=None*)
Class defining multiple command line parsers

This parser can process command lines of the form

PROG CMD OPTIONS ARGS

where different sets of options can be defined based on the major command ('CMD') supplied at the start of the line.

Usage:

Create a simple CommandParser which uses optparse.OptionParser as the default subparser backend using:

```
>>> p = CommandParser()
```

Alternatively, specify argparse.ArgumentParser as the subparser using:

```
>>> p = CommandParser(subparser=argparser.ArgumentParser)
```

Add a 'setup' command:

```
>>> p.add_command('setup',usage='%prog setup OPTIONS ARGS')
```

Add options to the 'setup' command using the appropriate methods of the subparser (e.g. 'add_argument' for an ArgumentParser instance).

For example:

```
>>> p.parser_for('info').add_argument('-f',...)
```

To process a command line, use the 'parse_args' method, for example for an OptionParser-based subparser:

```
>>> cmd,options,args = p.parse_args()
```

Note that the exact form of the returned values depends on on the subparser instance; it will be the same as that returned by the 'parse_args' method of the subparser.

add_command(*cmd*, *help=None*, ***args*)
Add a major command to the CommandParser

Adds a command, and creates and returns an initial subparser instance for it.

Arguments: cmd: the command to be added help: (optional) help text for the command

Other arguments are passed to the subparser instance when it is created i.e. 'usage', 'version', 'description'.

If 'version' isn't specified then the version supplied to the CommandParser object will be used.

Returns: Subparser instance object for the command.

error(*message*)
Exit with error message

handle_generic_commands(*cmd*)
Process 'generic' commands e.g. 'help'

list_commands()
Return the list of commands

parse_args(*argv=None*)
Process a command line

Parses a command line (either those supplied to the calling subprogram e.g. via the Python interpreter, or as a list).

Once the command is identified from the first argument, the remainder of the arguments are passed to the 'parse_args' method of the appropriate subparser for that command.

This method returns a tuple, with the first value being the command, and the rest of the values being those returned from the 'parse_args' method of the subparser.

Arguments:

argv: (optional) a list consisting of a command line. If not supplied then defaults to sys.argv[1:].

Returns: A tuple of (cmd,...), where 'cmd' is the command, and '...' represents the values returned from the 'parse_args' method of the subparser. For example, using the default OptionParser backend returns (cmd,options,arguments), where 'options' and 'arguments' are the options and arguments as returned by OptionParser.parse_args; using ArgumentParser as a backend returns (cmd,arguments).

parser_for(*cmd*)
Return OptionParser for specified command

Returns: The OptionParser object for the specified command.

print_available_commands()
Pretty-print available commands

Returns a 'pretty-printed' string for all options and commands, with standard whitespace formatting.

print_command(*cmd*, *message=None*)
Print a line for a single command

Returns a 'pretty-printed' line for the specified command and text, with standard whitespace formatting.

bcftbx.cmdparse.**add_arg**(*p*, **args*, ***kwds*)
Add an argument or option to a parser

Given an arbitrary parser instance, adds a new option or argument using the appropriate method call and passing the supplied arguments and keywords.

For example, if the parser is an instance of argparse.ArgumentParser, then the 'add_argument' method will be invoked to add a new argument to the parser.

Arguments:

p (Object): parser instance; can be an instance of one of: optparse.OptionContainer (i.e. OptionParser or OptionGroup), or argparse.ArgumentParser

args (List): list of argument values to pass directly to the argument-addition method

kwds (mapping): keyword-value mapping to pass directly to the argument-addition method

bcftbx.cmdparse.**add_debug_option**(*parser*)
Add a '–debug' option to a parser

Given a parser instance 'parser' (either OptionParser or ArgumentParser), add a '–debug' option.

The value of this option can be accessed via the 'debug' attribute of the parser options.

Returns the input parser object.

bcftbx.cmdparse.**add_dry_run_option**(*parser*)
Add a '–dry-run' option to a parser

Given a parser instance 'parser' (either OptionParser or ArgumentParser), add a '–dry-run' option.

The value of this option can be accessed via the 'dry_run' attribute of the parser options.

Returns the input parser object.

bcftbx.cmdparse.**add_no_save_option**(*parser*)
Add a '–no-save' option to a parser

Given a parser instance 'parser' (either OptionParser or ArgumentParser), add a '–no-save' option.

The value of this option can be accessed via the 'no_save' attribute of the parser options.

Returns the input parser object.

bcftbx.cmdparse.**add_nprocessors_option**(*parser*, *default_nprocessors*, *default_display=None*)
Add a '–nprocessors' option to a parser

Given a parser instance 'parser' (either OptionParser or ArgumentParser), add a '–nprocessors' option.

The value of this option can be accessed via the 'nprocessors' attribute of the parser options.

If 'default_display' is not None then this value will be shown in the help text, rather than the value supplied for the default.

Returns the input parser object.

bcftbx.cmdparse.**add_runner_option**(*parser*)
Add a '–runner' option to a parser

Given a parser instance 'parser' (either OptionParser or ArgumentParser), add a '–runner' option.

The value of this option can be accessed via the 'runner' attribute of the parser options (use the 'fetch_runner' function to return a JobRunner object from the supplied value).

Returns the input parser object.

4.12 bcftbx.qc

4.12.1 bcftbx.qc.report

Utilities for generating reports for NGS QC pipeline runs.

class bcftbx.qc.report.**IlluminaQCReporter**(*dirn*, *data_format=None*, *qc_dir='qc'*, *regex_pattern=None*, *version=None*)
Class for reporting QC run on Illumina data

IlluminaQCReporter assembles the data associated with a QC run for a set of Illumina data and generates a HTML document which summarises the results for quick review.

report()
Write the HTML report

Writes a HTML document 'qc_report.html' to the top-level analysis directory.

zip()
Make a zip file containing the report and the images

Generate the 'qc_report.html' file and make a zip file 'qc_report.<run>.<name>.zip' which contains the report plus the associated image files, which can be unpacked elsewhere for viewing.

Returns: Name of the zip file with the report.

class bcftbx.qc.report.**IlluminaQCSample**(*name*, *qc_dir*, *fastq=None*)
Class for holding QC data for an Illumina sample

An Illumina QC run typically consists of contamination screens and output from FastQC.

is_empty
Return True if the sample has no reads, False otherwise

report(*html*)
Write HTML report for this sample

verify()
Check QC products for this sample

Checks that fastq_screens and FastQC files were found. Returns True if the QC products are present and False otherwise.

class bcftbx.qc.report.**QCReporter**(*dirn*, *data_format=None*, *qc_dir='qc'*, *regex_pattern=None*, *version=None*)
Base class for reporting QC runs

This is a general class for reporting runs of the FLS NGS QC pipelines. QC reporters specific to particular pipelines should be subclassed from QCReporter and need to implement the 'report' method to generate the HTML output.

addSample(*sample*)
Add a QCSample class or subclass to the sample list

data_format
Return the format for the primary data files

dirn
Return top-level directory containing data

getPrimaryDataFiles()
Return list of primary data file sets

Returns a list of primary data file names; use the 'primary_data_dir' property to get the directory where the files are actually located.

html
Return HTMLPageWriter instance for the report

name
Return name of experiment

primary_data_dir
Return location of primary data files

qc_dir
Return directory holding QC outputs

report()
Generate a HTML report

This method must be implemented by the subclass.

report_base_name
Return the base name for the report

report_name
Return the full name for the report

run
Return name of run

samples
Return list of samples

verify()
Check that the QC outputs are correct

Returns True if the QC appears to have run successfully, False if not.

zip()
Make a zip file containing the report and the images

Generate the 'qc_report.html' file and make a zip file 'qc_report.<run>.<name>.zip' which contains the report plus the associated image files etc. The archive can then be unpacked elsewhere for viewing.

Returns: Name of the zip file with the report.

exception bcftbx.qc.report.**QCReporterError**
Base class for errors with QCReporter-related code

class bcftbx.qc.report.**QCSample**(*name*, *qc_dir*)
Base class for reporting QC for a single sample

This is a general class for reporting the QC outputs associated with a single sample. It attempts to find all possible associated QC products for the given sample name.

Specific pipelines should subclass QCSample and implement the 'report' method, which can call the 'report_*' methods to produce HTML code specific to the pipeline in question.

addBoxplot(*boxplot*)
Associate a boxplot with the sample

Arguments: boxplot: boxplot file name

addFastQC(*fastqc_dir*)
Associate a FastQC output directory with the sample

addProgramInfo(*programs*)
Collect program information from 'programs' file

addScreen(*screen*)
Associate a fastq_screen with the sample

Arguments: screen: fastq_screen file name

boxplots()
Return list of boxplots for a sample

fastqc
Return name of FastQC run dir

programs
Return data on programs

report()
Generate a HTML report

This method must be implemented by the subclass.

report_boxplots(*html*, *paired_end=False*, *inline_pngs=True*)
Write HTML code reporting the boxplots

Arguments: html: HTMLPageWriter instance to add the generated HTML to inline_pngs: if set True then embed the PNG images as base64

encoded data; otherwise link to the original image file

report_fastqc(*html*, *inline_pngs=True*)
Write HTML code reporting the results from FastQC

Arguments: html: HTMLPageWriter instance to add the generated HTML to

report_programs(*html*)
Write HTML code reporting the program information

report_screens(*html*, *inline_pngs=True*)
Write HTML code reporting the fastq screens

Arguments: html: HTMLPageWriter instance to add the generated HTML to inline_pngs: if set True then embed the PNG images as base64

encoded data; otherwise link to the original image file

screens()
Return list of screens for a sample

verify()
Verify expected QC products for the sample

This method must be implemented by the subclass. It should return True if the QC appears to have run successfully for the sample, False if not.

zip_includes()
Return list of files and directories to archive

class bcftbx.qc.report.**SolidQCReporter**(*dirn*, *data_format=None*, *qc_dir='qc'*, *regex_pattern=None*, *version=None*)
Class for reporting QC run on SOLiD data

SolidQCReporter assembles the data associated with a QC run for a set of SOLiD data and generates a HTML document which summarises the results for quick review.

report()
Write the HTML report

Writes a HTML document 'qc_report.html' to the top-level analysis directory.

verify()
Verify that SOLiD QC completed successfully for all samples

Returns True if the QC appears to have run successfully, False if not.

class bcftbx.qc.report.**SolidQCSample**(*name*, *qc_dir*, *paired_end*)
Class for holding QC data for a SOLiD sample

A SOLiD QC run typically consists of filtered and unfiltered boxplots, quality filtering stats, and contamination screens.

report(*html*)
Write HTML report for this sample

verify()
Check QC products for this sample

Checks that fastq_screens and boxplots were found. Returns True if the QC products are present and False otherwise.

bcftbx.qc.report.**add_dir_to_zip**(*z*, *dirn*, *zip_top_dir=None*)
Recursively add a directory and its contents to a zip archive

z is a zipfile.ZipFile object already opened for writing; this function adds all files in directory dirn and its subdirectories to z.

If zip_top_dir is not None then this is prepended to the file name written to the zip archive.

bcftbx.qc.report.**cmp_boxplots**(*b1*, *b2*)
Compare the names of two boxplots for sorting purposes

bcftbx.qc.report.**cmp_samples**(*s1*, *s2*)
Compare the names of two samples for sorting purposes

bcftbx.qc.report.**count_reads**(*csfasta_file*)
Count the number of reads in a CSFASTA file

Returns number of reads, or None

bcftbx.qc.report.**is_boxplot**(*name*, *f*)
Return True if f is a qc_boxplot associated with sample

'name' can be a file name, or a file 'root' i.e. filename with all trailing extensions removed.

bcftbx.qc.report.**is_fastq_screen**(*name*, *f*)
Return True if f is a fastq_screen file associated with name

'name' can be a file name, or a file 'root' i.e. filename with all trailing extensions removed.

bcftbx.qc.report.**is_fastqc**(*name*, *f*)
Return True if f is a FastQC file associated with name

'name' can be a file name, or a file 'root' i.e. filename with all trailing extensions removed.

bcftbx.qc.report.**is_program_info**(*name*, *f*)
Return True if f is a 'program info' file associated with name

'name' can be a file name, or a file 'root' i.e. filename with all trailing extensions removed.

bcftbx.qc.report.**split_sample_name**(*name*)
Split name into leading part plus trailing number

Returns (start,number)

bcftbx.qc.report.**strip_ngs_extensions**(*name*)
Remove fastq, fastq, csfasta or qual extensions from name

4.13 bcftbx.htmlpagewriter

class bcftbx.htmlpagewriter.**HTMLPageWriter**(*title=''*)
Generic HTML generation class

HTMLPageWriter provides basic operations for writing HTML files.

Example usage:

```
>>> p = HTMLPageWriter("Example page")
>>> p.add("This is some text")
>>> p.write("example.html")
```

add(*content*)
Add content to page body

Note that the supplied content is added to the HTML document as-is; no escaping is performed so the content can include arbitrary HTML tags. Note also that no validation is performed.

Arguments: content: text to add to the HTML document body

addCSSRule(*css_rule*)
Add CSS rule

Defines a CSS rule that will be inlined into a "style" tag in the HTML head when the document is written out.

The rule text is added as-is, e.g.:

```
>>> p = HTMLPageWriter("Example page")
>>> p.addCSSRule("body { color: blue; }")
```

No checking or validation is performed.

Arguments: css_rule: text defining CSS rule

addJavaScript(*javascript*)
Add JavaScript

Defines a line of Javascript code that will be inlined into a "script" tag in the HTML head when the document is written out.

The code is added as-is, no checking or validation is performed.

Arguments: javascript: Javascript code

write (*filen=None*, *fp=None*)
Write the HTML document to file

Generates a HTML document based on the content, styles etc that have been defined by calls to the object's methods.

Can supply either a filename or a file-like object opened for writing.

Arguments: filen: name of the file to write the document to. fp : file-like object opened for writing; if this

is supplied then filen argument will be ignored even if it is not None.

class bcftbx.htmlpagewriter.**PNGBase64Encoder**
Utility class to encode PNG file into a base64 string

Base64 encoded PNGs can be embedded in HTML <img> tags.

To use:

```
>>> p = PNGBase64Encoder.encodePNG("image.png")
```

encodePNG (*pngfile*)
Return base64 string encoding a PNG file.

4.14 bcftbx.utils

utils

Utility classes and functions shared between BCF codes.

General utility classes:

AttributeDictionary OrderedDictionary

File reading utilities:

getlines

File system wrappers and utilities:

PathInfo mkdir mkdirs mklink chmod touch format_file_size commonprefix is_gzipped_file rootname find_program get_current_user get_user_from_uid get_uid_from_user get_group_from_gid get_gid_from_group get_hostname walk list_dirs strip_ext

Symbolic link handling:

Symlink links

Sample name utilities:

extract_initials extract_prefix extract_index_as_string extract_index pretty_print_names name_matches

File manipulations:

concatenate_fastq_files

Text manipulations:

split_into_lines

Command line parsing utilities:

parse_named_lanes parse_lanes

4.14.1 General utility classes

class bcftbx.utils.**AttributeDictionary**(***args*)

Dictionary-like object with items accessible as attributes

AttributeDict provides a dictionary-like object where the value of items can also be accessed as attributes of the object.

For example:

```
>>> d = AttributeDict()
>>> d['salutation'] = "hello"
>>> d.salutation
... "hello"
```

Attributes can only be assigned by using dictionary item assignment notation i.e. d['key'] = value. d.key = value doesn't work.

If the attribute doesn't match a stored item then an AttributeError exception is raised.

len(d) returns the number of stored items.

The AttributeDict behaves like a dictionary for iterations, for example:

```
>>> for attr in d:
>>>     print "%s = %s" % (attr,d[attr])
```

class bcftbx.utils.**OrderedDictionary**

Augumented dictionary which keeps keys in order

OrderedDictionary provides an augmented Python dictionary class which keeps the dictionary keys in the order they are added to the object.

Items are added, modified and removed as with a standard dictionary e.g.:

```
>>> d[key] = value
>>> value = d[key]
>>> del(d[key])
```

The 'keys()' method returns the OrderedDictionary's keys in the correct order.

4.14.2 File handling utilities

bcftbx.utils.**getlines**(*filen*)

Fetch lines from a file and return them one by one

This generator function tries to implement an efficient method of reading lines sequentially from a file, by minimising the number of reads from the file and performing the line splitting in memory. It attempts to replicate the idiom:

```
>>> for line in open(filen):
>>> ...
```

using:

```
>>> for line in getlines(filen):
>>> ...
```

The file can be gzipped; this function should handle this invisibly provided that the file extension is '.gz'.

Arguments: filen (str): path of the file to read lines from

Yields:

String: next line of text from the file, with any newline character removed.

4.14.3 File system wrappers and utilities

class bcftbx.utils.**PathInfo**(*path*, *basedir=None*)

Collect and report information on a file

The PathInfo class provides an interface to getting general information on a path, which may point to a file, directory, link or non-existent location.

The properties provide information on whether the path is readable (i.e. accessible) by the current user, whether it is readable by members of the same group, who is the owner and what group does it belong to, when was it last modified etc.

chown(*user=None*, *group=None*)

Change associated owner and group

'user' and 'group' must be supplied as UID/GID numbers (or None to leave the current values unchanged).

*** Note that chown will fail attempting to change the owner if the current process is not owned by root ***

This is actually a wrapper to the os.lchmod function, so it doesn't follow symbolic links.

datetime

Return last modification time as datetime object

deepest_accessible_parent

Return longest accessible directory that leads to path

Tries to find the longest parent directory above path which is accessible by the current user.

If it's not possible to find a parent that is accessible then raise an exception.

exists

Return True if the path refers to an existing location

Note that this is a wrapper to os.path.lexists so it reports the existence of symbolic links rather than their targets.

gid

Return associated GID (group ID)

Attempts to return the GID (group ID) number associated with the path.

If the GID can't be found then returns None.

group

Return associated group name

Attempts to return the group name associated with the path. If the name can't be found then tries to return the GID instead.

If neither pieces of information can be found then returns None.

is_dir
Return True if path refers to a directory

is_executable
Return True if path refers to an executable file

is_file
Return True if path refers to a file

is_group_readable
Return True if the path exists and is group-readable

Paths may be reported as unreadable for various reasons, e.g. the target doesn't exist, or doesn't have permission for this user to read it, or if part of the path doesn't allow the user to read the file.

is_group_writable
Return True if the path exists and is group-writable

Paths may be reported as unwritable for various reasons, e.g. the target doesn't exist, or doesn't have permission for this user to write to it, or if part of the path doesn't allow the user to read the file.

is_link
Return True if path refers to a symbolic link

is_readable
Return True if the path exists and is readable by the owner

Paths may be reported as unreadable for various reasons, e.g. the target doesn't exist, or doesn't have permission for this user to read it, or if part of the path doesn't allow the user to read the file.

mtime
Return last modification timestamp for path

path
Return the filesystem path

relpath(*dirn*)
Return part of path relative to a directory

Wrapper for os.path.relpath(...).

resolve_link_via_parent
If path or parent directory is a link then return actual path

Resolves and returns the 'real' path for a path where either it or one of its parent directories is a symbolic link.

It will resolve multiple levels of symlinks to generate a path that is free of links (nb it is possible that the resolved path will not be an existing file or directory).

If there are no links in the directory tree then returns the full path of the input.

uid
Return associated UID (user ID)

Attempts to return the UID (user ID) number associated with the path.

If the UID can't be found then returns None.

user
Return associated user name

Attempts to return the user name associated with the path. If the name can't be found then tries to return the UID instead.

If neither pieces of information can be found then returns None.

bcftbx.utils.**mkdir**(*dirn*, *mode=None*, *recursive=False*)
Make a directory

Arguments: dirn: the path of the directory to be created mode: (optional) a mode specifier to be applied to the new directory once it has been created e.g. 0775 or 0664

recursive: (optional) if True then also create any intermediate parent directories if they don't already exist

bcftbx.utils.**mklink**(*target*, *link_name*, *relative=False*)
Make a symbolic link

Arguments: target: the file or directory to link to link_name: name of the link relative: if True then make a relative link (if possible);

otherwise link to the target as given (default)

bcftbx.utils.**chmod**(*target*, *mode*)
Change mode of file or directory

This a wrapper for the os.chmod function, with the addition that it doesn't follow symbolic links.

For symbolic links it attempts to use the os.lchmod function instead, as this operates on the link itself and not the link target. If os.lchmod is not available then links are ignored.

Arguments: target: file or directory to apply new mode to mode: a valid mode specifier e.g. 0775 or 0664

bcftbx.utils.**touch**(*filename*)
Create new empty file, or update modification time if already exists

Arguments: filename: name of the file to create (can include leading path)

bcftbx.utils.**format_file_size**(*fsize*, *units=None*)
Format a file size from bytes to human-readable form

Takes a file size in bytes and returns a human-readable string, e.g. 4.0K, 186M, 1.5G.

Alternatively specify the required units via the 'units' arguments.

Arguments: fsize: size in bytes units: (optional) specify output in kb ('K'), Mb ('M'),

Gb ('G') or Tb ('T')

Returns: Human-readable version of file size.

bcftbx.utils.**commonprefix**(*path1*, *path2*)
Determine common prefix path for path1 and path2

Use this in preference to os.path.commonprefix as the version in os.path compares the two paths in a character-wise fashion and so can give counter-intuitive matches; this version compares path components which seems more sensible.

For example: for two paths /mnt/dir1/file and /mnt/dir2/file, os.path.commonprefix will return /mnt/dir, whereas this function will return /mnt.

Arguments: path1: first path in comparison path2: second path in comparison

Returns: Leading part of path which is common to both input paths.

bcftbx.utils.**is_gzipped_file**(*filename*)
Check if a file has a .gz extension

Arguments: filename: name of the file to be tested (can include leading path)

Returns: True if filename has trailing .gz extension, False if not.

bcftbx.utils.**rootname**(*name*)
Remove all extensions from name

Arguments: name: name of a file

Returns: Leading part of name up to first dot, i.e. name without any trailing extensions.

bcftbx.utils.**find_program**(*name*)
Find a program on the PATH

Search the current PATH for the specified program name and return the full path, or None if not found.

bcftbx.utils.**get_current_user**()
Return name of the current user

Looks up user name for the current user; returns None if no matching name can be found.

bcftbx.utils.**get_user_from_uid**(*uid*)
Return user name from UID

Looks up user name matching the supplied UID; returns None if no matching name can be found.

bcftbx.utils.**get_uid_from_user**(*user*)
Return UID from user name

Looks up UID matching the supplied user name; returns None if no matching name can be found.

NB returned UID will be an integer.

bcftbx.utils.**walk**(*dirn*, *include_dirs=True*, *pattern=None*)
Traverse the directory, subdirectories and files

Essentially this 'walk' function is a convenience wrapper for the 'os.walk' function.

Arguments: dirn: top-level directory to start traversal from include_dirs: if True then yield directories as well as files (default)

pattern: if not None then specifies a regular expression pattern which restricts the set of yielded files and directories to a subset of those which match the pattern

bcftbx.utils.**list_dirs**(*parent*, *matches=None*, *startswith=None*)
Return list of subdirectories relative to 'parent'

Arguments: parent: directory to list subdirectories of matches: if not None then only include subdirectories that exactly match the supplied string

startswith: if not None then then return subset of subdirectories that start with the supplied string

Returns: List of subdirectories (relative to the parent dir).

bcftbx.utils.**strip_ext**(*name*, *ext=None*)
Strip extension from file name

Given a file name or path, remove the extension (including the dot) and return just the leading part of the name.

If an extension is explicitly specified then only remove the extension if it matches.

Extension can be multipart e.g. 'fastq.gz' and can include a leading dot e.g. '.gz' or 'gz'.

Arguments: name: name of a file

Returns: Leading part of name excluding specified extension, or first extension i.e. to last dot.

4.14.4 Symbolic link handling

class bcftbx.utils.**Symlink**(*path*)
Class for interrogating and modifying symbolic links

The Symlink class provides an interface for getting information about a symbolic link.

To create a new Symlink instance do e.g.:

```
>>> l = Symlink('my_link.lnk')
```

Information about the link can be obtained via the various properties:

- target = returns the link target
- is_absolute = reports if the target represents an absolute link
- is_broken = reports if the target doesn't exist

There are also methods:

- resolve_target() = returns the normalise absolute path to the target
- update_target() = updates the target to a new location

is_absolute
Return True if the link target is an absolute link

is_broken
Return True if the link target doesn't exist i.e. link is broken

resolve_target()
Return the normalised absolute path to the link target

target
Return the target of the symlink

update_target(*new_target*)
Replace the current link target with new_target

Arguments: new_target: path to replace the existing target with

bcftbx.utils.**links**(*dirn*)
Traverse and return all symbolic links in under a directory

Given a starting directory, traverses the structure underneath and yields the path for each symlink that is found.

Arguments: dirn: name of the top-level directory

Returns: Yields the name and full path for each symbolic link under 'dirn'.

4.14.5 Sample name utilities

bcftbx.utils.**extract_initials**(*name*)
Return leading initials from the library or sample name

Conventionaly the experimenter's initials are the leading characters of the name e.g. 'DR' for 'DR1', 'EP' for 'EP_NCYC2669', 'CW' for 'CW_TI' etc

Arguments: name: the name of a sample or library

Returns: The leading initials from the name.

bcftbx.utils.**extract_prefix**(*name*)
Return the library or sample name prefix

Arguments: name: the name of a sample or library

Returns: The prefix consisting of the name with trailing numbers removed, e.g. 'LD_C' for 'LD_C1'

bcftbx.utils.**extract_index_as_string**(*name*)
Return the library or sample name index as a string

Arguments: name: the name of a sample or library

Returns: The index, consisting of the trailing numbers from the name. It is returned as a string to preserve leading zeroes, e.g. '1' for 'LD_C1', '07' for 'DR07' etc

bcftbx.utils.**extract_index**(*name*)
Return the library or sample name index as an integer

Arguments: name: the name of a sample or library

Returns: The index as an integer, or None if the index cannot be converted to integer format.

bcftbx.utils.**pretty_print_names**(*name_list*)
Given a list of library or sample names, format for pretty printing.

Arguments: name_list: a list or tuple of library or sample names

Returns: String with a condensed description of the library names, for example:

['DR1', 'DR2', 'DR3', DR4'] -> 'DR1-4'

bcftbx.utils.**name_matches**(*name*, *pattern*)
Simple wildcard matching of project and sample names

Matching options are:

- exact match of a single name e.g. pattern 'PJB' matches 'PJB'
- match start of a name using trailing '*' e.g. pattern 'PJ*' matches 'PJB','PJBriggs' etc
- match using multiple patterns by separating with comma e.g. pattern 'PJB,IJD' matches 'PJB' or 'IJD'. Subpatterns can include trailing '*' character to match more names.

Arguments name: text to match against pattern pattern: simple 'glob'-like pattern to match against

Returns True if name matches pattern; False otherwise.

4.14.6 File manipulations

bcftbx.utils.**concatenate_fastq_files**(*merged_fastq*, *fastq_files*, *bufsize=10240*, *overwrite=False*, *verbose=True*)
Create a single FASTQ file by concatenating one or more FASTQs

Given a list or tuple of FASTQ files (which can be compressed or uncompressed or a combination), creates a single output FASTQ by concatenating the contents.

Arguments: merged_fastq: name of output FASTQ file (mustn't exist beforehand) fastq_files: list of FASTQ files to concatenate bufsize: (optional) size of buffer to use for copying data overwrite: (optional) if True then overwrite the output file if it

already exists (otherwise raise OSError); default is False

verbose: (optional) if True then report operations to stdout, otherwise operate quietly

4.14.7 Text manipulations

bcftbx.utils.**split_into_lines**(*text*, *char_limit*, *delimiters='\t\n'*, *sympathetic=False*)

Split a string into multiple lines with maximum length

Splits a string into multiple lines on one or more delimiters (defaults to the whitespace characters i.e. ' ',tab and newline), such that each line is no longer than a specified length.

For example:

```
>>> split_into_lines("This is some text to split",10)
['This is','some text','to split']
```

If it's not possible to split part of the text to a suitable length then the line is split "unsympathetically" at the line length, e.g.

```
>>> split_into_lines("This is supercalifragilicous text",10)
['This is','supercalif','ragilicous','text']
```

Set the 'sympathetic' flag to True to include a hyphen to indicate that a word has been broken, e.g.

```
>>> split_into_lines("This is supercalifragilicous text",10,
...                  sympathetic=True)
['This is','supercali-','fragilico-','us text']
```

To use an alternative set of delimiter characters, set the 'delimiters' argument, e.g.

```
>>> split_into_lines("This: is some text",10,delimiters=':')
['This',' is some t','ext']
```

Arguments: text: string of text to be split into lines char_limit: maximum length for any given line delimiters: optional, specify a set of non-default

delimiter characters (defaults to whitespace)

sympathetic: optional, if True then add hyphen to indicate when a word has been broken

Returns: List of lines (i.e. strings).

4.15 bcftbx.ngsutils

ngsutils

Utility classes and functions specific to NGS applications.

Extracting reads from Fastq, cfasta and qual files:

- getreads: fetch reads one-by-one from Fastq, cfasta or qual file
- getreads_subset: fetch subset of reads specified by index
- getreads_regexp: fetch subset of reads matching regular expression

4.15.1 Extracting reads from Fastq, cfasta and qual files

bcftbx.ngsutils.**getreads**(*filen*)
Return Fastq, csfasta or qual file reads one-by-one

This generator function iterates through a sequence file (Fastq, csfasta or qual), and yields read records one at a time. The read records are returned as lists of lines.

The file can be gzipped; this function should handle this invisibly provided that the file extension is '.gz'.

Lines starting with '#' at the start of the file will be treated as comments and ignored. Lines starting with '#' which occur in the body of the file (i.e. after one or more lines of data) will be treated as data.

Example usage:

```
>>> for r in getreads('illumina_R1.fq'):
>>> ... print r
```

Arguments: filen (str): path of the file to fetch reads from

Yields:

List: next read record from the file, as a list of lines.

bcftbx.ngsutils.**getreads_subset**(*filen*, *indices*)
Fetch subset of reads from Fastq, csfasta or qual file

This generator function iterates through a sequence file (Fastq, csfasta or qual), and yields a subset of the read records which are referenced by the supplied iterable indices.

The subset compromises of reads at the index positions specified by the list of indices, with index 0 being the first read in the file. Each read is returned as a list of lines.

The file can be gzipped; this function should handle this invisibly provided that the file extension is '.gz'.

Example usage (returns 1st, 3rd and 5th reads only):

```
>>> for r in getreads_subset('illumina_R1.fq',(0,2,4)):
>>> ... print r
```

Arguments: filen (str): path of the file to fetch reads from indices (list): list of read indices to return

Yields:

List: next read record from the file, as a list of lines.

bcftbx.ngsutils.**getreads_regex**(*filen*, *pattern*)
Fetch matching reads from Fastq, csfasta or qual file

This generator function iterates through a sequence file (Fastq, csfasta or qual), and yields a subset of read records. Each read is returned as a list of lines.

The subset compromises of reads which match the supplied regular expression.

The file can be gzipped; this function should handle this invisibly provided that the file extension is '.gz'.

Example usage:

```
>>> for r in getreads_regexp('illumina_R1.fq',"2102:3130"):
>>> ... print r
```

Arguments: filen (str): path of the file to fetch reads from pattern (list): Python regular expression pattern

Yields:

List: next read record from the file, as a list of lines.

CHAPTER 5

Related projects

These are other BCF-related utilities external to this package.

5.1 RnaChipIntegrator

Performs various analyses which integrate ChIP and RNA-seq results by looking for nearest gene/transcript TSS positions to ChIP peaks or binding regions.

Download from:

- http://fls-bioinformatics-core.github.com/RnaChipIntegrator

5.2 GFFUtils

Utility programs which can perform various operations on GFF and GTF files (e.g. clean up, extract annotation).

Download from:

- https://github.com/fls-bioinformatics-core/GFFUtils

5.3 bedUtils

Utilities for making BED/bedGraph files:

- *bedGraphSplitter.py*:
- *bedMaker.py*: create a BED file from a tab-delimited input data file
- *bedMaker_unexplained.py*: create a BED file from "unexplained" data

Download from:

- https://github.com/fls-bioinformatics-core/bedUtils

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

Symbols

A

B

C

N

O

P

Q

R

S

T

U

V

W

X

Z