Gpipe Documentation

Release

Andreas Heger

December 09, 2013

Contents

This document brings together the various pipelines and scripts written before and during CGAT.

Note: The documentation is under construction.

The CGAT Code collection is documented *here*.

CHAPTER 1

Overview

The CGAT code collection has grown out of the work in comparative genomics by the Ponting group in the last decade. Now, CGAT has added functionality to do next-generation sequencing analysis.

The collection has three major components, these are directories in the package.

- *Scripts* A collection of useful scripts for genomics and NGS analysis
- *Modules* A collection of modules with utility functions for genomics and NGS analysis.
- *CGAT Pipelines* A collection of pipelines for common workflows in genomics and NGS analysis.

CHAPTER 2

Scripts and modules

The CGAT code collection is as set of tools and modules for genomics. Most of these scripts are written in python. The tools are grouped by topic:

2.1 Scripts

This document contains all the scripts for/by CGAT. Scripts are written to be called from the command line.

2.1.1 Genomics

2.1.2 Trees

2.1.3 Alignment

2.1.4 Visualization

2.1.5 Graphs

2.1.6 Sequences and rates

2.1.7 Matrices and Tables

2.1.8 Stats

2.1.9 Tools

Databases

Cluster and jobs

Other

cgat_html_add_toc.py - insert table of contents in html document

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Purpose Read an html document on stdin and add a table of contents based on section headings in the document. This document uses the `<h1></h1>`, `<h2></h2>`, ... html tags.

Usage Example:

```
python cgat_html_add_toc.py --help
```

Type:

```
python cgat_html_add_toc.py --help
```

for command line help.

Command line options

2.1.10 Unsorted

mali_phylip2fasta.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Purpose

Todo

describe purpose of the script.

Usage

Example:

```
python mali_phylip2fasta.py --help
```

Type:

```
python mali_phylip2fasta.py --help
```

for command line help.

Command line options

2.2 Modules

Contents:

2.2.1 Genomics

AString.py - a compact string of characters

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

class AString.**AString**(**args*)
an array posing as a sequence.

This class conserves memory as it uses only 1 byte per letter, while python strings use the machine word size for a letter.

It exports a mixture of the methods in the python string and python array classes.

Note: Using this class will incur a heavy penalty compared to using :class:array.array directly. Only use sparingly for heavy computations.

---

upper()
return upper case version.

lower()
return lower case version.

FastaIterator.py - iterate over fasta files

The difference to the biopython iterator is that this one skips over comment lines starting with "#".

Code

class FastaIterator.**FastaIterator**(*f*, **args*, ***kwargs*)
a iterator of *fasta* formatted files.

FastaIterator.**iterate_together**(**args*)
iterate synchronously over one or more fasta files.

The iteration finishes once any of the files is exhausted.

yield output tuples of sequences.

FastaIterator.**count**(*filename*)
count number of sequences in fasta file.

Intervals.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

Intervals.**combine**(*intervals*)
combine intervals.

Overlapping intervals are concatenated into larger intervals.

Intervals.**prune**(*intervals*, *first=None*, *last=None*)
truncates all intervals that are extending beyond first or last.

Empty intervals are deleted.

Intervals.**complement**(*intervals*, *first=None*, *last=None*)
complement a list of intervals with intervals not in list.

Intervals.**addComplementIntervals**(*intervals*, *first=None*, *last=None*)
complement a list of intervals with intervals not in list and return both.

Intervals.**combineAtDistance**(*intervals*, *min_distance*)
combine a list intervals and merge those that are less than a certain distance apart.

Intervals.**DeleteSmallIntervals**(*intervals*, *min_length*)
combine a list of non-overlapping intervals, and delete those that are too small.

Intervals.**getIntersections**(*intervals*)
combine intervals.

Overlapping intervals are reduced to their intersection.

Intervals.**RemoveIntervalsContained**(*intervals*)
remove intervals that are fully contained in another.

[(10, 100), (20, 50), (70, 120), (130, 200), (10, 50), (140, 210), (150, 200)]

results:

[(10, 100), (70, 120), (130, 200), (140, 210)]

Intervals.**RemoveIntervalsSpanning**(*intervals*)
remove intervals that are full covering another, i.e. always keep the smallest.

[(10, 100), (20, 50), (70, 120), (40,80), (130, 200), (10, 50), (140, 210), (150, 200)]

result:

[(20, 50), (40, 80), (70, 120), (150, 200)]

Intervals.**ShortenIntervalsOverlap**(*intervals*, *to_remove*)
shorten intervals, so that there is no overlap with another set of intervals.

assumption: intervals are not overlapping

Intervals.**joined_iterator**(*intervals1*, *intervals2*)
iterate over the combination of two intervals.

returns the truncated intervals delineating the ranges of overlap between intervals1 and intervals2.

Intervals.**intersect**(*intervals1*, *intervals2*)
intersect two interval sets.

Return a set of intervals that is spanned by intervals in both sets. Returns the union of the two intervals.

Intervals.**getLength**(*intervals*)
return sum of intervals.

Intervals.**truncate**(*intervals1*, *intervals2*)
truncate intervals in intervals1 by intervals2

Example: truncate([(0,5)], [(0,3)]) = [(3,5)]

Intervals.**calculateOverlap**(*intervals1*, *intervals2*)
calculate overlap between intervals.

The intervals within each set should not be overlapping.

Intervals.**fromArray**(*a*)
get intervals from a binary array.

Mali.py - Tools for multiple alignments

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

class Mali.**SequenceCollection**
Bases: Mali.Mali

reads in a sequence collection, but permits several entries per id.

Note that this might cause problems with interleaved formats like phylips or clustal.

This mapping is achieved by appending a numeric suffix. The suffix is retained for the life-time of the object, but not output to a file.

addEntry(*s*)
add an aligned string object.

readFromFile(*infile*, *format='fasta'*)
read multiple alignment from file in various format.

addAnnotation(*key*, *annotation*)
add annotation.

apply(*f*)
apply function f to every row in the multiple alignment.

buildColumnMap(*other*, *join_field=None*)
build map of columns in other to this.

checkLength()
check lengths of aligned strings.

Return false if they are inconsistent.

clipByAnnotation(*key*, *chars=''*)
restrict alignment to positions where annotation identified by key in chars.

if chars is empty, nothing is clipped.

copyAnnotations(*other*)
copy annotations from annother mali.

filter(*f*)
filter multiple alignment using function *f*.

The function *f* should return True for entries that should be kept and False for those that should be removed.

getAlphabet()
get alphabet from the multiple alignment.

Alphabet is "na", if more than 90% of characaters are "actgxn", otherwise it is "aa".

getAnnotation(*key*)
return annotation associated with key.

getColumns()
return mali in column orientation.

getConsensus(*mark_with_gaps=False*)
return consensus string.

The consensus string returns the most frequent character per column that is not a gap. If mark_with_gaps is set to True, positions with any gap characater are set to gaps.

getLength()
deprecated.

getResidueNumber(*key*, *position*)
return residue number in sequence key at position position.

getWidth()
deprecated.

insertColumns(*position*, *num_gaps*, *keep_fixed=None*, *char='-'*)
insert gaps at position into multiple alignment.

if keep_constant is a list of identifiers, those are kept constant, instead, gaps are added to the end.

lower()
convert all characters in mali to lowercase.

lowerCase()
set all characters to lower case.

mapColumns(*columns*, *map_function*)
apply map_function to all residues in columns.

mapIdentifiers(*map_old2new=None*, *pattern_identifier='ID%06i'*)
map identifiers in multiple aligment.

if map_old2new is not given, a new map is created (map_new2old)

markCodons(*mode='case'*)
mark codons.

markTransitions(*map_id2transitions*, *mode='case'*)
mark transitions in the multiple alignment.

if mode == case, then upper/lower case is used for the transitions

Otherwise, a character given by mode is inserted.

maskColumn(*column*, *mask_char='x'*)
mask a column.

maskColumns(*columns*, *mask_char='x'*)
mask columns in a multiple alignment.

propagateMasks(*min_chars=1*, *mask_char='x'*)
propagate masked characters to all rows of a multiple alignment within a column.

If there is at least min_chars in a mali column, that are masks, propagate the masks to all other rows.

propagateTransitions(*min_chars=1*)
propagate lower case in a column to all residues.

recount (*reset_first=False*)
recount residue in alignments.

removeEmptySequences ()
remove sequences that are completely empty.

removeEndGaps ()
remove end gaps.

end gaps do not include any characters and thus the alignment coordinates won't change.

removeGaps (*allowed_gaps=0*, *minimum_gaps=1*, *frame=1*)
remove gappy columns.

allowed_gaps: number of gaps allowed for column to be kept minimum_gaps: number of gaps for column to be removed

set minimum_gaps to the number of sequences to remove columns with all gaps.

If frame is > 1 (3 most likely), then a whole codon is removed as soon as there is one column to be removed.

removePattern (*match_function*, *allowed_matches=0*, *minimum_matches=1*, *delete_frame=1*, *search_frame=1*)
remove columns (or group of columns), that match a certain pattern.

allowed_matches: number of matches allowed so that column is still kept minimum_matches: number of matches required for column to be removed

set minimum_matches to the number of sequences to remove columns with all gaps.

Patterns are matches in search_frame. For example, if frame is 3, whole codons are supplied to match_function.

delete_frame specifies the frame for deletion. If it is set to 3, codons are removed if already one column matches.

Example: remove all columns that contain at least one stop-codon:

removePattern(lambda x: x.upper() in ("TAG", "TAA", "TGA"), allowed_matches = 0, minimum_matches = 1, search_frame = 3, delete_frame = 3)

removeUnalignedEnds ()
remove unaligned ends in the multiple alignment.

unaligned ends correspond to lower-case characters.

rename (*old_name*, *new_name*)
rename an entry.

setAnnotation (*key*, *value*)
set annotation associated with key to value.

shiftAlignment (*map_id2offset*)
shift alignment by offset.

shuffle (*frame=1*)
shuffle multiple alignment.

The frame determines the block size for shuffling. Use 3 for codons in a multiple alignment without frame-shifts.

takeColumns (*columns*)
restrict alignments to certain columns.

truncate (*first*, *last*)
truncate alignment within range.

upper()
convert all characters in mali to uppercase.

upperCase()
set all characters to upper case.

writeToFile(*outfile*, *write_ranges=True*, *format='plain'*, *options=None*)
write alignment to file.

If options is given, these lines are output into the multiple alignment.

Mali.**convertMali2Alignlib**(*mali*)
convert a multiple alignment of type Mali into an alignlib_lite.py_multiple alignment object.

Mali.**convertAlignlib2Mali**(*mali*, *identifiers=None*, *seqs=None*)
convert a multiple alignment into an alignlib_lite.py_multiple alignment object.

MaliIO.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

MaliIO.**writeFasta**(*outfile*, *mali*, *line_width=60*, *identifiers=None*, *skip_first=0*, *gap_char=None*)
print multiple alignment in fasta format.

MaliIO.**WriteMODELLER**(*outfile*, *mali*, *line_width=60*, *identifiers=None*, *skip_first=0*, *gap_char=None*)
print multiple alignment in PIR (MODELLER) format. example:

>P1;5fd1 structureX:5fd1:1 : :106 : :ferredoxin:Azotobacter vinelandii: 1.90: 0.19 AFVVTDNCIKCK-YTDCVEVCPVDCFYEGPNFLVIHPDECIDCALCEPECPAQAIFSEDEVPEDMQEFIQLNAELA EVWP-NITEKKDPLPDAEDWDGVKGKLQHLER*

MaliIO.**writeClustalW**(*outfile*, *mali*, *line_width=60*, *identifiers=None*)
print alignment in ClustalW format. (have to still look up the format, dsc can read it.)

MaliIO.**readPicasso**(*infile*)
read alignment in the non-defined picasso format.

MaliIO.**compressAlignment**(*alignment*, *gap_character='-'*, *ignore_beginning=0*)
compress an alignment string. Lower-case characters at the beginning are ignored if so wished. – xxabBCDEfgHI becomes: -6+4

This was necessary for radar output (e.g., 46497)

MaliIO.**readFasta**(*infile*, *pattern_identifier='\\S+'*)
read alignment in fasta format.

MaliIO.**convertGaps**(*mali*, *old_gap='.'*, *new_gap='-'*)
convert gaps characters in mali.

MaliIO.**removeGappedColumns**(*mali*, *gap_char='-'*)
remove all gapped columns in mali.

MaliIO.**getSubset**(*mali*, *identifiers*, *not_in_set=False*)
return subset of mali which only contains identifiers.

MaliIO.**getFrameColumnsForMaster**(*mali*, *master*, *gap_char='-'*)
get columns in frame according to master.

MaliIO.**getFrameColumnsForMasterPattern**(*mali*, *identifiers*, *master_pattern*, *gap_char='-'*)
get columns in frame for all masters matching the pattern.

MaliIO.**getMapFromMali**(*seq1*, *seq2*, *gap_char='-'*)
build map of positions between mali.

MaliIO.**getCodonSequence**(*sequence*, *frame_columns*, *gap_char='-'*, *remove_stops=True*)
return a pruned sequence given frame columns.

everything not in frame is deleted, only complete codons are kept.

MaliIO.**getPercentIdentity**(*seq1*, *seq2*, *gap_char='-'*)
get number of identical residues between seq1 and seq2.

GenomicIO.py - Subroutines for working on I/O of large genomic files

Author

Release Id

Date December 09, 2013

Tags Python

I tried the Biopython parser, but it was too slow for large genomic chunks.

GenomicIO.**index_file**(*filenames*, *db_name*)
index file/files.

Two new files are create - db_name.fasta and db_name.idx

GenomicIO.**index_exists**(*filename*)
check if a certain file has been indexed.

GenomicIO.**getSequence**(*db_name*, *sbjct_token*, *sbjct_strand*, *sbjct_from*, *sbjct_to*, *as_array=False*, *forward_coordinates=False*)
get genomic fragment.

GenomicIO.**splitFasta**(*infile*, *chunk_size*, *dir='/tmp'*, *pattern=None*)
split a fasta file into a subset of files.

If pattern is not given, random file names are chosen.

GenomicIO.**getConverter**(*format*)
return a converter function for converting various coordinate schemes into 0-based, both strand, closed-open ranges.

converter functions have the parameters x, y, s, l: with x and y the coordinates of a sequence fragment, s the strand (True is positive) and l being the length of the contig.

Maq.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

exception `Maq.`**`Error`**
Bases: `exceptions.Exception`

Base class for exceptions in this module.

exception `Maq.`**`ParsingError`**(*message*, *line=None*)
Bases: `Maq.Error`

Exception raised for errors while parsing

Attributes: message – explanation of the error

class `Maq.`**`Match`**
a maq match.

`Maq.`**`iterator`**(*infile*)
iterate over the contents of a maq file.

AGP.py - working with AGP files

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

to assemble contigs to scaffolds.

Code

Regions.py - helper functions for working with genomic segments

Version: $Id: Regions.py 2781 2009-09-10 11:33:14Z andreas $

class `Regions.`**`RegionFilter`**
Filter class based on regions.

`readFromFile`(*infile*, *ignore_strand=False*)
read regions from a file.

`getOverlaps`(*sbjct_token*, *sbjct_strand*, *sbjct_from*, *sbjct_to*)
return overlapping regions with region.

ProfileLibrary.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

ProfileLibraryCompass.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

2.2.2 Phylogeny

PamMatrices.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

2.2.3 Parsers and wrappers

WrapperAdaptiveCAI.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

WrapperBl2Seq.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

WrapperENC.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

WrapperMuscle.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

exception `WrapperMuscle.`**`Error`**
Bases: `exceptions.Exception`

Base class for exceptions in this module.

exception `WrapperMuscle.`**`ParsingError`**(*message*, *line*)
Bases: `WrapperMuscle.Error`

Exception raised for errors while parsing

Attributes: message – explanation of the error

exception `WrapperMuscle.`**`UsageError`**(*message*)
Bases: `WrapperMuscle.Error`

Exception raised for errors while starting

Attributes: message – explanation of the error

2.2.4 Math and Stats

CorrespondenceAnalysis.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

CorrespondenceAnalysis.**GetIndices**(*matrix*)
return order (1st eigenvector) of row and column indicies.

This procedure fails if there are row or columns with a sum of 0.

CorrespondenceAnalysis.**GetPermutatedMatrix**(*matrix*, *map_row_new2old*, *map_col_new2old*, *row_headers=None*, *col_headers=None*)
return a permuted matrix. Note, that currently this is very inefficient, as I do not know how to do this in numpy.

Histogram.py - Various functions to deal with histograms

Author

Release Id

Date December 09, 2013

Tags Python

Histograms can be calculated from a list/tuple/array of values. The histogram returned is then a list of tuples of the format [(bin1,value1), (bin2,value2), ...].

Histogram.**CalculateFromTable**(*dbhandle*, *field_name*, *from_statement*, *num_bins=None*, *min_value=None*, *max_value=None*, *intervals=None*, *increment=None*)
get a histogram using an SQL-statement. Intervals can be either supplied directly or are build from the data by providing the number of bins and optionally a minimum or maximum value.

If no number of bins are provided, the bin-size is 1.

This command uses the INTERVAL command from MYSQL, i.e. a bin value determines the upper boundary of a bin.

Histogram.**CalculateConst**(*values*, *num_bins=None*, *min_value=None*, *max_value=None*, *intervals=None*, *increment=None*, *combine=None*)
calculate a histogram based on a list or tuple of values.

Histogram.**Calculate**(*values*, *num_bins=None*, *min_value=None*, *max_value=None*, *intervals=None*, *increment=None*, *combine=None*, *no_empty_bins=0*, *dynamic_bins=False*, *ignore_out_of_range=True*)
calculate a histogram based on a list or tuple of values.

use scipy for calculation.

Histogram.**Scale**(*h*, *scale=1.0*)
rescale bins in histogram.

Histogram.**Convert**(*h*, *i*, *no_empty_bins=0*)
add bins to histogram.

Histogram.**Combine**(*source_histograms*, *missing_value=0*)
combine a list of histograms Each histogram is a sorted list of bins and counts. The counts can be tuples.

Histogram.**Print**(*h*, *intervalls=None*, *format=0*, *nonull=None*, *format_value=None*, *format_bin=None*)
print a histogram.

A histogram can either be a list/tuple of values or a list/tuple of lists/tuples where the first value contains the bin and second contains the values (which can again be a list/tuple).

format 0 = print histogram in several lines 1 = print histogram on single line

Histogram.**Write**(*outfile*, *h*, *intervalls=None*, *format=0*, *nonull=None*, *format_value=None*, *format_bin=None*)
print a histogram.

A histogram can either be a list/tuple of values or a list/tuple of lists/tuples where the first value contains the bin and second contains the values (which can again be a list/tuple).

Parameters **format** – output format. 0 = print histogram in several lines, 1 = print histogram on single line

Histogram.**Fill**(*h*)
fill every empty value in histogram with previous value.

Histogram.**Add**(*h1*, *h2*)
adds values of histogram h1 and h2 and returns a new histogram

Histogram.**SmoothWrap**(*histogram*, *window_size*)
smooth histogram by sliding window-method, where the window is wrapped around the borders. The sum of all values is entered at center of window.

Histogram.**PrintAscii**(*histogram*, *step_size=1*)
print histogram ascii-style.

Histogram.**Count**(*data*)
count categorized data. Returns a list of tuples with (count, token).

Histogram.**Accumulate**(*h*, *num_bins=2*, *direction=1*)
add successive counts in histogram. Bins are labelled by group average.

Histogram.**Cumulate**(*h*, *direction=1*)
calculate cumulative distribution.

Histogram.**AddRelativeAndCumulativeDistributions**(*h*)
adds relative and cumulative percents to a histogram.

Histogram.**histogram**(*values*, *mode=0*, *bin_function=None*)
Return a list of (value, count) pairs, summarizing the input values. Sorted by increasing value, or if mode=1, by decreasing count. If bin_function is given, map it over values first. Ex: vals = [100, 110, 160, 200, 160, 110, 200, 200, 220] histogram(vals) ==> [(100, 1), (110, 2), (160, 2), (200, 3), (220, 1)] histogram(vals, 1) ==> [(200, 3), (160, 2), (110, 2), (100, 1), (220, 1)] histogram(vals, 1, lambda v: round(v, -2)) ==> [(200.0, 6), (100.0, 3)]

Histogram.**cumulate**(*histogram*)
cumulate histogram in place.

histogram is list of (bin, value) or (bin, (values,))

Histogram.**normalize**(*histogram*)
normalize histogram in place.

histogram is list of (bin, value) or (bin, (values,))

Histogram.**fill**(*iterator*, *bins*)
fill a histogram from bins.

The values are given by an iterator so that the histogram can be built on the fly.

Description:

Count the number of times values from array a fall into numerical ranges defined by bins. Range x is given by bins[x] <= range_x < bins[x+1] where x =0,N and N is the length of the bins array. The last range is given by bins[N] <= range_N < infinity. Values less than bins[0] are not included in the histogram.

Arguments: iterator – The iterator. bins – 1D array. Defines the ranges of values to use during histogramming.

Returns: 1D array. Each value represents the occurences for a given bin (range) of values.

Histogram.**fillHistograms**(*infile*, *columns*, *bins*)
fill several histograms from several columns in a file.

The histograms are built on the fly.

Description:

Count the number of times values from array a fall into numerical ranges defined by bins. Range x is given by bins[x] <= range_x < bins[x+1] where x =0,N and N is the length of the bins array. The last range is given by bins[N] <= range_N < infinity. Values less than bins[0] are not included in the histogram.

Arguments: file – The input file. columns – columns to use bins – a list of 1D arrays. Defines the ranges of values to use during histogramming.

Returns: a list of 1D arrays. Each value represents the occurences for a given bin (range) of values.

WARNING: missing value in columns are ignored

Histogram2D.py - functions for handling two-dimensional histograms.

Author

Release Id

Date December 09, 2013

Tags Python

Histogram2D.**Calculate**(*values*, *mode=0*, *bin_function=None*)
Return a list of (value, count) pairs, summarizing the input values. Sorted by increasing value, or if mode=1, by decreasing count.

If bin_function is given, map it over values first.

Histogram2D.**Print**(*h*, *bin_function=None*)
print a histogram.

A histogram can either be a list/tuple of values or a list/tuple of lists/tuples where the first value contains the bin and second contains the values (which can again be a list/tuple).

Parameters **format** – output format. 0 = print histogram in several lines, 1 = print histogram on single line

MatlabTools.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

MatlabTools.**WriteMatrix**(*matrix*, *outfile=<open file '<stdout>', mode 'w' at 0x7f116e56d150>*, *separator='\t'*, *format='%f'*, *row_headers=None*, *col_headers=None*)
write matrix to stream.

MatlabTools.**ReadMatrix**(*file*, *separator='\t'*, *numeric_type=<type 'float'>*, *take='all'*, *headers=False*)
read a matrix. There probably is a routine for this in Numpy, which I haven't found yet.

MatlabTools.**ReadSparseMatrix**(*filename*, *separator='\t'*, *numeric_type=<type 'float'>*, *is_symmetric=None*)
read sparse matrix.

MatlabTools.**ReadBinarySparseMatrix**(*filename*, *separator='\t'*, *numeric_type=<type 'float'>*, *is_symmetric=None*)
read sparse matrix.

MatlabTools.**readMatrix**(*infile*, *format='full'*, *separator='\t'*, *numeric_type=<type 'float'>*, *take='all'*, *headers=True*, *missing=None*)
read a matrix from file ane return a numpy matrix.

formats accepted are: * full * sparse * phylip

MatlabTools.**writeMatrix**(*outfile*, *matrix*, *format='full'*, *separator='\t'*, *value_format='%f'*, *row_headers=None*, *col_headers=None*)
write matrix to stream.

2.2.5 Gpipe and OPTIC

Exons.py - A library to read/write/manage exons.

Author

Release Id

Date December 09, 2013

Tags Python

class Exons.**Exon**
class for exons.

contains info about the genomic location of an exon and its location within a peptide sequence.

The field mAlignment is set optionally.

Read(*line*, *contig_sizes={}*, *format='exons'*, *extract_id=None*, *converter=None*)
read exon from tab-separated line.

extract_id is a regular expression object to extract the identifier from the identifier column.

if converter is given, it is used to convert to zero-based open-closed both strand coordinates.

Merge(*other*)
Merge this exon with another (adjacent and preceeding) exon.

Do not merge if the distance between exons is not divisible by 3. Merging of two exons invalidated peptide coordinates for all following exons. These need to be updated.

InvertGenomicCoordinates(*lgenome*)
invert genomic alignment on sequence.

Negative strand is calculated from the other end.

Exons.**UpdatePeptideCoordinates**(*exons*)
updates peptides coordinates for a list of exons.

Exons have to be sorted.

Exons.**PostProcessExons**(*all_exons*, *do_invert=None*, *remove_utr=None*, *filter=None*, *reset=False*, *require_increase=False*, *no_invert=False*, *contig_sizes={}*, *from_zero=False*, *delete_missing=False*, *set_peptide_coordinates=False*, *set_rank=False*)

do post-processing of exons

exons is a dictionary of lists of exons.

Exons are sorted by mPeptideFrom.

Operations include:

-invert: sort out forward/reverse strand coordinates

-set_peptide_coordinates: sets the peptide coordinates of exons.

-set-rank: set rank of exons

-remove_utr: remove any utr (needs peptide coordinates)

-delete_missing: if set set true, exons on contigs not in contig_sizes will be deleted.

-from_zero: exon genomic coordinates start at 0

-reset: exon genomic coordinates start 0

Exons.**GetExonBoundariesFromTable**(*dbhandle*, *table_name_predictions='predictions'*, *table_name_exons='exons'*, *only_good=False*, *do_invert=None*, *remove_utr=None*, *filter=None*, *reset=False*, *require_increase=False*, *contig_sizes={}*, *prediction_ids=None*, *table_name_quality='quality'*, *table_name_redundant='redundant'*, *non_redundant_filter=False*, *schema=None*, *quality_filter=None*, *from_zero=False*, *delete_missing=False*)

get exon boundaries from table.

Exons.**CountNumExons**(*exons*)

return hash with number of exons per entry.

Exons.**SetRankToPositionFlag**(*exons*)

set rank for all exons.

Set rank to 1 : if it is first exon, -1: if it is the last exon (single exon genes are -1) 0 : if it is an internal exon.

Exons.**ReadExonBoundaries**(*file*, *do_invert=None*, *remove_utr=None*, *filter=None*, *reset=False*, *require_increase=False*, *no_invert=False*, *contig_sizes={}*, *converter=None*, *from_zero=False*, *delete_missing=False*, *format='exons'*, *gtf_extract_id=None*)

read exons boundaries from tab separated file.

if remove_utr is set, the UTR of the first/last exon is removed.

if reset is set, then the genomic part is moved so that it starts at 1. if require_increase is set, then exons are sorted in increasing order.

If do_invert is set: negative strand coordinates are converted to positive strand coordinates

if no_invert is set: coordinates are kept as they are.

if from_zero is set: coordinates are mapped from 0. Thus reverse strand coordinates will be negative.

if delete_missing is True and sbjct-token is not in contig_sizes but the exon needs to be inverted: delete transcript.

The exon file format is tab-separated and can be of the two formats:

format="exons": id, contig, strand, frame, rank, peptide_from, peptide_to, genome_from, genome_to

format = "gtf": contig, ignored, ignored, genome_from, genome_to, ignored, strand, frame, id

if converter is given, use it to convert to forward/reverse strand coordinates.

gtg_extract_id: regular expression object to extract id from id column.

Exons.**Alignment2Exons**(*alignment*, *query_from=0*, *sbjct_from=0*, *add_stop_codon=1*)
convert a Peptide2DNA alignment to exon boundaries.

Exons.**Exons2Alignment**(*exons*)
build alignment string from a (sorted) list of exons.

Exons.**RemoveRedundantEntries**(*l*)
remove redundant entries (and 0s) from list.

One liner?

Exons.**CompareGeneStructures**(*xcmp_exons*, *ref_exons*, *map_ref2cmp=None*, *cmp_sequence=None*, *ref_sequence=None*, *threshold_min_pide=0*, *threshold_slipping_exon_boundary=9*, *map_cmp2ref=None*, *threshold_terminal_exon=15*)
Compare two gene structures.

This function is useful for comparing the exon boundaries of a predicted peptide with the exon boundaries of the query peptide.

cmp_exons are exons for the gene to test. ref_exons are exons from the reference.

Exon boundaries are already mapped to the peptide for the reference.

map_ref2cmp: Alignment of protein sequences for cmp and ref. map_cmp2ref: Alignment of cmp to ref. If given, mapping is done from cmp to ref. Invalid exon boundaries can be set to -1.

threshold_terminal_exon: Disregard terminal exons for counting missed boundaries, if they are maximum x nucleotides long.

Exons.**MapExons**(*exons*, *map_a2b*)
map peptide coordinates of exons with map.

returns a list of mapped exons.

Exons.**CountMissedBoundaries**(*cmp_boundaries*, *reference_boundaries*, *max_slippage=9*, *min_from=0*, *max_to=0*)
count missed boundaries comparing cmp to ref.

Exons.**GetExonsRange**(*exons*, *first*, *last*, *full=True*, *min_overlap=0*, *min_exon_size=0*)
get exons in range (first:last) (peptide coordinates).

Set full to False, if you don't require full coverage.

Exons.**ClusterByExonIdentity**(*exons*, *max_terminal_num_exons=3*, *min_terminal_exon_coverage=0.0*, *max_slippage=0*, *loglevel=0*)
build clusters of transcripts with identical exons.

The boundaries in the first/last exon can vary.

Returns two maps map_cluster2transcripts and map_transcript2cluster

Exons.**ClusterByExonOverlap**(*exons*, *min_overlap=0*, *min_min_coverage=0*, *min_max_coverage=0*, *loglevel=0*)
build clusters of transcripts with overlapping exons.

Exons need not be identical.

Returns two maps map_cluster2transcripts and map_transcript2cluster

Exons.**CheckOverlap**(*exons1*, *exons2*, *min_overlap=1*)
check if exons overlap.

(does not check chromosome and strand.)

Exons.**CheckCoverage**(*exons1*, *exons2*, *max_terminal_num_exons=3*, *min_terminal_exon_coverage=0.0*, *max_slippage=0*)
check if one set of exons covers the other.

Note: does not check chromosome and strand, just genomic coordinates.

Exons.**CheckContainedAinB**(*exons1*, *exons2*, *min_terminal_exon_coverage=0.0*, *loglevel=0*)
check if all exons in exons1 are contained in exons2.

Note: does not check contig and strand.

Exons.**CheckCoverageAinB**(*exons1*, *exons2*, *min_terminal_num_exons=3*, *min_terminal_exon_coverage=0.0*, *max_slippage=0*, *loglevel=0*)
check if exons1 are all in exons2

Note: does not check contig and strand.

Exons.**GetPeptideLengths**(*exons*)
for all exons get maximum length in coding nucleotides.

Exons.**GetGenomeLengths**(*exons*)
for all exons get maximum nucleotide.

Exons.**CalculateStats**(*exons*)
calculate some statistics for all exons.

minimum/maximum intron/exon length, number of exons gene length

Exons.**MatchExons**(*map_a2b*, *in_exons1*, *in_exons2*, *threshold_slipping_boundary=9*)
returns a list of overlapping exons (mapped via map_a2b).

Orthologs.py - tools to deal with Leo's orthology pipeline.

Author

Release Id

Date December 09, 2013

Tags Python

Orthologs.**FilterBDGP**(*data*)
remove genes that are

transcripts from BDGP and genes from ENSEMBL.

If only BDGP genes are present, keep them.

Orthologs.**GetGenes**(*transcripts*)
from a list of transcripts get genes.

Orthologs.**ReadInterpretation**(*infile*, *separator*, *genome1=None*, *genome2=None*, *filter_restrict_genes1={}*, *filter_restrict_genes2={}*, *filter_remove_transcripts1={}*, *filter_remove_transcripts2={}*, *filter_restrict_transcripts1={}*, *filter_restrict_transcripts2={}*)
read interpretation file.

Orthologs.**ReadOrphans**(*infile*, *separator*, *genome1=None*, *genome2=None*, *filter_restrict_genes1={}*, *filter_restrict_genes2={}*, *filter_remove_transcripts1={}*, *filter_remove_transcripts2={}*, *filter_restrict_transcripts1={}*, *filter_restrict_transcripts2={}*)
read interpretation file.

Orthologs.**ClusterOrthologsByGenes**(*orthologs*)
cluster orthologs by genes.

if an orthologous cluster contains the same genes in either species, they are merged.

Orthologs.**GetDegeneracy**(*t1*, *t2*)
get degeneracy of orthology assignments.

given are two lists of transcripts. returns a tuple with gene and transcript degeneracy code, respectively.

BlastAlignments.py - tools for working with alignments

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

class BlastAlignments.**Map**
a blast alignment.

MapRange(*query_token*, *query_from*, *query_to*)
map something.

GetClone()
get copy of self.

BlastAlignments.**ReadMap**(*file*, *multiple=False*)
read a map from a file.

If multiple is true, return a list of mappings.

BlastAlignments.**iterator_links**(*infile*)
a simple iterator over all entries in a file.

2.2.6 Tools

Experiment.py - Tools for scripts

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

The Experiment modules contains utility functions for logging and record keeping of scripts.

This module is imported by most CGAT scripts. It provides convenient and consistent methods for

- Record keeping
- Benchmarking

See <*no title*> on how to use this module.

The basic usage of this module within a script is:

```
"""script_name.py - my script

Mode Documentation
"""
import sys
import optparse
import CGAT.Experiment as E

def main( argv = None ):
    """script main.

    parses command line options in sys.argv, unless *argv* is given.
    """

    if not argv: argv = sys.argv

    # setup command line parser
    parser = E.OptionParser( version = "%prog version: $Id$",
                                    usage = globals()["__doc__"] )

    parser.add_option("-t", "--test", dest="test", type="string",
                      help="supply help"  )

    ## add common options (-h/--help, ...) and parse command line
    (options, args) = E.Start( parser )

    # do something
    # ...
    E.info( "an information message" )
    E.warn( "a warning message")

    ## write footer and output benchmark information.
    E.Stop()

if __name__ == "__main__":
    sys.exit( main( sys.argv) )
```

Record keeping

The central functions in this module are the `Start()` and `Stop()` methods which are called before or after any work is done within a script.

`Experiment.`**`Start`**(*parser=None, argv=['/home/docs/checkouts/readthedocs.org/user_builds/cgat/envs/latest/bin/sphinx-build', '-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex'], quiet=False, no_parsing=False, add_csv_options=False, add_mysql_options=False, add_psql_options=False, add_pipe_options=True, add_cluster_options=False, add_output_options=False, return_parser=False*)

set up an experiment.

param parser an `E.OptionParser` instance with commandi line options. *param argv* command line options to parse. Defaults to `sys.argv` *quiet* set *loglevel* to 0 - no logging *no_parsing* do not parse com-

mand line options *return_parser* return the parser object, no parsing *add_csv_options* add common options for parsing *tsv* separated files *add_mysql_options* add common options for connecting to mysql databases *add_psql_options* add common options for connecting to postgres databases *add_pipe_options* add common options for redirecting input/output *add_cluster_options* add common options for scripts submitting jobs to the cluster *add_output_options* add commond options for working with multiple output files *returns* a tuple (options,args) with options (a `E.OptionParser` object

and a list of positional arguments.

The `Start()` method will also set up a file logger.

The default options added by this method are:

`-v/--verbose` the *loglevel*

`timeit` turn on benchmarking information and save to file

`timeit-name` name to use for timing information,

`timeit-header` output header for timing information.

Optional options added are:

add_csv_options

`dialect` csv_dialect. the default is `excel-tab`, defaulting to *tsv* formatted files.

add_psql_options

`-C/--connection` psql connection string

`-U/--user` psql user name

add_cluster_options

`--use-cluster` use cluster

`--cluster-priority` cluster priority to request

`--cluster-queue` cluster queue to use

`--cluster-num-jobs` number of jobs to submit to the cluster at the same time

`--cluster-options` additional options to the cluster for each job.

add_output_options

`-P/--output-filename-pattern` Pattern to use for output filenames.

The `Start()` is called with an E.OptionParser object. `Start()` will add additional command line arguments, such as `--help` for command line help or `--verbose` to control the *loglevel*. It can also add optional arguments for scripts needing database access, writing to multiple output files, etc.

`Start()` will write record keeping information to a logfile. Typically, logging information is output on stdout, prefixed by a *#*, but it can be re-directed to a separate file. Below is a typical output:

```
# output generated by /ifs/devel/andreas/cgat/beds2beds.py --force --exclusive --method=unmerged-comb
# job started at Thu Mar 29 13:06:33 2012 on cgat150.anat.ox.ac.uk -- e1c16e80-03a1-4023-9417-f3e44e
# pid: 16649, system: Linux 2.6.32-220.7.1.el6.x86_64 #1 SMP Fri Feb 10 15:22:22 EST 2012 x86_64
# exclusive                             : True
# filename_update                       : None
# ignore_strand                         : False
# loglevel                              : 1
# method                                : unmerged-combinations
# output_filename_pattern               : 030m.intersection.tsv.dir/030m.intersection.tsv-%s.bed.g
```

```
# output_force                          : True
# pattern_id                            : (.*).bed.gz
# stderr                                : <open file '<stderr>', mode 'w' at 0x2ba70e0c2270>
# stdin                                 : <open file '<stdin>', mode 'r' at 0x2ba70e0c2150>
# stdlog                                : <open file '030m.intersection.tsv.log', mode 'a' at 0x1f
# stdout                                : <open file '<stdout>', mode 'w' at 0x2ba70e0c21e0>
# timeit_file                           : None
# timeit_header                         : None
# timeit_name                           : all
# tracks                                : None
```

The header contains information about:

- the script name (`beds2beds.py`)
- the command line options (`--force --exclusive --method=unmerged-combinations --output-filename-pattern=030m.intersection.tsv.dir/030m.intersection.tsv-%s.bed.gz --log=030m.intersection.tsv.log Irf5-030m-R1.bed.gz Rela-030m-R1.bed.gz`)
- the time when the job was started (`Thu Mar 29 13:06:33 2012`)
- the location it was executed (`cgat150.anat.ox.ac.uk`)
- a unique job id (`e1c16e80-03a1-4023-9417-f3e44e33bdcd`)
- the pid of the job (`16649`)
- the system specification (`Linux 2.6.32-220.7.1.el6.x86_64 #1 SMP Fri Feb 10 15:22:22 EST 2012 x86_64`)

It is followed by a list of all options that have been set in the script.

Once completed, a script will call the `Stop()` function to signify the end of the experiment.

`Experiment.`**`Stop`**`()`
stop the experiment.

`Stop()` will output to the log file that the script has concluded successfully. Below is typical output:

```
# job finished in 11 seconds at Thu Mar 29 13:06:44 2012 -- 11.36  0.45  0.00  0.01 -- e1c16e80-03a1
```

The footer contains information about:

- the job has finished (`job finished`)
- the time it took to execute (`11 seconds`)
- when it completed (`Thu Mar 29 13:06:44 2012`)
- **some benchmarking information (`11.36 0.45 0.00 0.01`) which is** `user time`, `system time`, `child user time`, `child system time`.
- the unique job id (`e1c16e80-03a1-4023-9417-f3e44e33bdcd`)

The unique job id can be used to easily retrieve matching information from a concatenation of log files.

Benchmarking

Complete reference

class `Experiment.`**`AppendCommaOption`**(**opts*, ***attrs*)
Bases: `optparse.Option`

Option with additional parsing capabilities.

- "," in arguments to options that have the action 'append' are treated as a list of options. This is what galaxy does, but generally convenient.
- Option values of "None" and "" are treated as default values.

class Experiment.**OptionParser**(**args*, ***kwargs*)
Bases: optparse.OptionParser

CGAT derivative of OptionParser.

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

parse_args(*args=None*, *values=None*)

parse_args(args [[string] = sys.argv[1:],] values : Values = None)

-> (values : Values, args : [string])

Parse the command-line options found in 'args' (default: sys.argv[1:]). Any errors result in a call to 'error()', which by default prints the usage message to stderr and calls sys.exit() with an error message. On success returns a pair (values, args) where 'values' is an Values instance (with all your option values) and 'args' is the list of arguments left over after parsing options.

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(),

any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

Experiment.**openFile**(*filename*, *mode='r'*, *create_dir=False*)
open file in *filename* with mode *mode*.

If *create* is set, the directory containing filename will be created if it does not exist.

gzip - compressed files are recognized by the suffix `.gz` and opened transparently.

Note that there are differences in the file like objects returned, for example in the ability to seek.

returns a file or file-like object.

Experiment.**getHeader**()
return a header string with command line options and timestamp

Experiment.**getParams**(*options=None*)
return a string containing script parameters.

Parameters are all variables that start with `param_`.

Experiment.**getFooter**()
return a header string with command line options and timestamp.

Experiment.**Start**(*parser=None, argv=['/home/docs/checkouts/readthedocs.org/user_builds/cgat/envs/latest/bin/sphinx-build', '-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex'], quiet=False, no_parsing=False, add_csv_options=False, add_mysql_options=False, add_psql_options=False, add_pipe_options=True, add_cluster_options=False, add_output_options=False, return_parser=False*)
set up an experiment.

param parser an `E.OptionParser` instance with commandi line options. *param argv* command line options to parse. Defaults to `sys.argv` *quiet* set *loglevel* to 0 - no logging *no_parsing* do not parse command line options *return_parser* return the parser object, no parsing *add_csv_options* add common options for parsing *tsv* separated files *add_mysql_options* add common options for connecting to mysql databases *add_psql_options* add common options for connecting to postgres databases *add_pipe_options* add common options for redirecting input/output *add_cluster_options* add common options for scripts submitting jobs to the cluster *add_output_options* add commond options for working with multiple output files *returns* a tuple (options,args) with options (a `E.OptionParser` object

and a list of positional arguments.

The `Start()` method will also set up a file logger.

The default options added by this method are:

-v/--verbose the *loglevel*

timeit turn on benchmarking information and save to file

timeit-name name to use for timing information,

timeit-header output header for timing information.

Optional options added are:

add_csv_options

dialect csv_dialect. the default is `excel-tab`, defaulting to *tsv* formatted files.

add_psql_options

-C/--connection psql connection string

-U/--user psql user name

add_cluster_options

--use-cluster use cluster

--cluster-priority cluster priority to request

--cluster-queue cluster queue to use

--cluster-num-jobs number of jobs to submit to the cluster at the same time

--cluster-options additional options to the cluster for each job.

add_output_options

-P/--output-filename-pattern Pattern to use for output filenames.

Experiment.**Stop**()
stop the experiment.

Experiment.**benchmark**(*func*)
decorator collecting wall clock time spent in decorated method.

Experiment.**cachedmethod**(*function*)
decorator for caching a method.

Experiment.**cachedfunction**
Decorator that caches a function's return value each time it is called. If called later with the same arguments, the cached value is returned, and not re-evaluated.

Taken from http://wiki.python.org/moin/PythonDecoratorLibrary#Memoize

Experiment.**log**(*loglevel*, *message*)
log message at loglevel.

Experiment.**info**(*message*)
log information message, see the `logging` module

Experiment.**warning**(*message*)
log warning message, see the `logging` module

Experiment.**warn**(*message*)
log warning message, see the `logging` module

Experiment.**debug**(*message*)
log debugging message, see the `logging` module

Experiment.**error**(*message*)
log error message, see the `logging` module

Experiment.**critical**(*message*)
log critical message, see the `logging` module

Experiment.**getOutputFile**(*section*)
return filename to write to.

Experiment.**openOutputFile**(*section*, *mode='w'*)
open file for writing substituting section in the output_pattern (if defined).

If the filename ends with ".gz", the output is opened as a gzip'ed file.

class Experiment.**Counter**
Bases: `object`

a counter class.

The counter acts both as a dictionary and a object permitting attribute access.

Counts are automatically initialized to 0.

Instantiate and use like this:

```
c = Counter()
c.input += 1
c.output += 2
c["skipped"] += 1

print str(c)
```

Store data returned by function.

asTable()
return values as tab-separated table (without header).

Experiment.**run**(*cmd*, *return_stdout=False*, ***kwargs*)
executed a command line cmd.

returns the return code.

If *return_stdout* is True, the contents of stdout are returned.

`kwargs` are passed on to subprocess.call or subprocess.check_output.

raises OSError if process failed or was terminated.

CSV.py - Tools for parsing CSV files

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

CSV.**ConvertDictionary**(*d*, *map={}*)
tries to convert values in a dictionary.

if map contains 'default', a default conversion is enforced. For example, to force int for every column but column id, supply map = { 'default' : "int", "id" : "str" }

CSV.**GetMapColumn2Type**(*rows*, *ignore_empty=False*, *get_max_values=False*)
map fields to types based on rows.

Preference is Int to Float to String.

If get_max_values is set to true, the maximum values for integer columns are returned in a dictionary.

class CSV.**CommentStripper**(*infile*)
iterator class for stripping comments from file.

class CSV.**DictReader**(*infile*, **args*, ***kwargs*)
Bases: `csv.DictReader`

like csv.DictReader, but skip comments (lines starting with "#").

class CSV.**DictReaderLarge**(*infile*, *fieldnames*, **args*, ***kwargs*)
drop-in for csv.DictReader - handles very large fields

Warning - minimal implementation - does not handle dialects

CSV.**ReadTable**(*lines*, *as_rows=True*, *with_header=True*, *ignore_incomplete=False*, *dialect='excel-tab'*)
read a table from infile

returns table as rows or as columns. If remove_incomplete, incomplete rows are simply ignored.

CSV.**ReadTables**(*infile*, **args*, ***kwargs*)
read a set of csv tables.

Individual tables are separated by // on a single line.

CSV.**GroupTable**(*table*, *group_column=0*, *group_function=<built-in function min>*, *missing_value='na'*)
group table by *group_column*.

The table need not be sorted.

IOTools - tools for I/O operations

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

IOTools.**readMap**(*infile*, *columns=(0, 1)*, *map_functions=(<type 'str'>, <type 'str'>)*, *both_directions=False*, *has_header=False*)
read a map (pairs of values) from infile. returns a hash.

Use map functions to convert elements. If both_directions is set to true, both mapping directions are returned.

IOTools.**readList**(*infile*, *column=0*, *map_function=<type 'str'>*, *map_category={}*, *with_title=False*)
read a list of values from infile.

Use map_function to convert values. Use map_category, to map read values directory If with_title, first line is assumed to be a title

IOTools.**ReadList**(*infile*, *column=0*, *map_function=<type 'str'>*, *map_category={}*)
read a list of values from infile.

Use map_function to convert values. Use map_category, to map read values directory

IOTools.**readMultiMap**(*infile*, *columns=(0, 1)*, *map_functions=(<type 'str'>, <type 'str'>)*, *both_directions=False*, *has_header=False*, *dtype=<type 'dict'>*)
read a map (pairs of values) from infile. returns a hash.

Use map functions to convert elements. If both_directions is set to true, both mapping directions are returned. This function can have n:n matches

IOTools.**readTable**(*file*, *separator='\t'*, *numeric_type=<type 'float'>*, *take='all'*, *headers=True*, *truncate=None*, *cumulate_out_of_range=True*)
read a table of values. There probably is a routine for this in Numpy, which I haven't found yet.

If cumulate_out_of_range is set to true, the terminal bins will contain the cumulative values of bins out of range.

IOTools.**writeTable**(*outfile*, *table*, *columns=None*, *fillvalue=''*)
write a table to outfile.

If table is a dictionary, output columnwise. If *columns* is a list, only output columns in columns in the specified order.

IOTools.**readMatrix**(*infile*, *dtype=<type 'float'>*)
read a numpy matrix from infile.

return tuple of matrix, row_headers, col_headers

IOTools.**writeMatrix**(*outfile*, *matrix*, *row_headers*, *col_headers*, *row_header=''*)
write a numpy matrix to outfile.

row_header gives the title of the rows

IOTools.**getInvertedDictionary**(*dict*, *make_unique=False*)
returns an inverted dictionary with keys and values swapped.

IOTools.**readSequence**(*file*)
read sequence from a fasta file.

returns a tuple with description and sequence

IOTools.**getLastLine**(*filename*, *nlines=1*, *read_size=1024*)
return last line of a file.

IOTools.**getNumLines**(*filename*, *ignore_comments=True*)
get number of lines in filename.

IOTools.**ReadMap**(**args*, ***kwargs*)
compatibility - see readMap.

IOTools.**isEmpty**(*filename*)
return True if file exists and is empty.

raises OSError if file does not exist

class IOTools.**FilePool**(*output_pattern=None*, *header=None*, *force=True*)
manage a pool of output files

This class will keep a large number of files open. To see if you can handle this, check the limit within the shell:

```
ulimit -n
```

The number of currently open and maximum open files in the system:

cat /proc/sys/fs/file-nr

Changing these limits might not be easy for a user.

This class is inefficient if the number of files is larger than `maxopen` and calls to *write* do not group keys together.

close()
close all open files.

getFilename(*identifier*)
get filename for an identifier.

openFile(*filename*, *mode='w'*)
open file.

If file is in a new directory, create directories.

deleteFiles(*min_size=0*)
delete all files below a minimum size.

class IOTools.**FilePoolMemory**(**args*, ***kwargs*)
Bases: IOTools.FilePool

manage a pool of output files

The data is cached in memory before writing to disk.

close()
close all open files. writes the data to disk.

deleteFiles(*min_size=0*)
delete all files below a minimum size.

getFilename(*identifier*)
get filename for an identifier.

openFile(*filename*, *mode='w'*)
open file.

If file is in a new directory, create directories.

IOTools.**val2str**(*val*, *format='%5.2f'*, *na='na'*)
return formatted value.

If value does not fit format string, return "na"

IOTools.**str2val**(*val*, *format='%5.2f'*, *na='na'*)
guess type of value.

IOTools.**prettyFloat**(*val*, *format='%5.2f'*)
deprecated, use val2str

IOTools.**prettyPercent**(*numerator*, *denominator*, *format='%5.2f'*, *na='na'*)
output a percent value or "na" if not defined

IOTools.**prettyString**(*val*)
output val or na if val == None

class IOTools.**nested_dict**
Bases: collections.defaultdict

Auto-vivifying nested dictionaries.

For example:

```
nd= nested_dict()
nd["mouse"]["chr1"]["+"] = 311
```

iterflattened()
iterate through values with nested keys flattened into a tuple

clear() → None. Remove all items from D.

copy() → a shallow copy of D.

default_factory
Factory for default value called by __missing__().

static fromkeys(*S*[, *v*]) → New dict with keys from S and values equal to v.
v defaults to None.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_key(*k*) → True if D has a key k, else False

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → list of D's values

viewitems() → a set-like object providing a view on D's items

viewkeys() → a set-like object providing a view on D's keys

viewvalues() → an object providing a view on D's values

IOTools.**flatten**(*l*, *ltypes=(<type 'list'>, <type 'tuple'>)*)
flatten a nested list/tuple.

IOTools.**which**(*program*)
check if program is in path.

from post at http://stackoverflow.com/questions/377017/test-if-executable-exists-in-python

IOTools.**convertValue**(*value*, *list_detection=False*)
convert a value to int, float or str.

IOTools.**iterate_tabular**(*infile*, *sep='\t'*)
iterate over infile skipping comments.

IOTools.**openFile**(*filename*, *mode='r'*, *create_dir=False*)
open file in *filename* with mode *mode*.

If *create* is set, the directory containing filename will be created if it does not exist.

gzip - compressed files are recognized by the suffix `.gz` and opened transparently.

Note that there are differences in the file like objects returned, for example in the ability to seek.

returns a file or file-like object.

IOTools.**iterate**(*infile*)
iterate over infile and return a namedtuple according to first row.

Iterators.py - general purpose iterators.

Author Unknown

Release Id

Date December 09, 2013

Tags Python

A collection of useful, general purpose iterators.

This code was downloaded from an unknown source.

Iterators.**sample**(*iterable*, *sample_size=None*)
sample # copies from iterator without replacement.

Stores a temporay copy of the items in iterable. The function has thus a possibly high memory footprint and long pre-processing time to yield the first element.

If sample_size is not given, the iterator returns elements in random order (see random.shuffle())

Iterators.**group_by_distance**(*iterable*, *distance=1*)
group integers into non-overlapping intervals that are at most *distance* apart.

```
>>> list( group_by_distance( (1,1,2,4,5,7) ) )
[(1, 3), (4, 6), (7, 8)]

>>> list( group_by_distance( [] ) )
[]

>>> list( group_by_distance( [3] ) )
[(3, 4)]

>>> list( group_by_distance( [3,2] ) )
Traceback (most recent call last):
...
ValueError: iterable is not sorted: 2 < 3
```

Database.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

Database.**executewait**(*dbhandle*, *statement*, *error=None*, *retries=-1*, *wait=5*)
execute an sql `statement`.

If `error` is given, it is scanned for locking issues.

Retry `retries` times if set to a positive number. A retry of `0` indicates no retry, a negative number retries infinitely.

The process waits `wait` seconds between each retry.

Returns a cursor object.

Database.**getColumnNames**(*dbhandle*, *table*)
get column names of a table from a database.

ExternalList.py - large disk-based lists

A list class that serves as a stand-in for python lists but stores data on the file system.

Implements not all functionality of lists yet. Lists are stored as tab-separated values for unix sort functionality, so all elements in the list should have the same type.

ProgressBar.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

class `ProgressBar.`**`ProgressBar`**(*minValue=0*, *maxValue=10*, *totalWidth=12*)
progress bar class

adapted from Randy Pargman (2002) see http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/168639

2.2.7 Pipelines

PipelineTracks.py - Definition of tracks in pipelines

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Motivation

A pipeline typically processes the data streams from several experimental data sources. These data streams are usually processed separately (processing, quality control) and as aggregates. For example, consider the following experimental layout:

Filename	*Content*
liver-stimulated-R1	liver, stimulated, replicate 1
liver-stimulated-R2	liver, stimulated, replicate 2
liver-unstimulated-R1	liver, unstimulated, replicate 1
liver-unstimulated-R2	liver, unstimulated, replicate 2
heart-stimulated-R1	heart, stimulated, replicate 1
heart-stimulated-R2	heart, stimulated, replicate 2
heart-unstimulated-R1	heart, unstimulated, replicate 1
heart-unstimulated-R2	heart, unstimulated, replicate 2

The experiment measured in two tissues with two conditions with two replicates each giving eight data streams. During the analysis, the streams are merged in a variety of combinations:

- unmerged for initial processing, QC, etc.
- by replicates to assess reproducibility of measurements
- by condition to assess the size of the response to the stimulus
- by tissue to assess differences between tissue and address the biological question.

The crossing of data streams complicates the building of pipelines, especially as no two experiments are the same. The `PipelineTracks` module assists in controlling these data streams. This module provides some tools to map tracks to different representations and to group them in flexible ways in order to provide convenient short-cuts in pipelines.

There are three class within `PipelineTracks`: `Sample`, `Tracks` and `Aggregate`.

A Track The basic atomic data structure is a `Sample` or *track*. A *track* is a single measurement that can be combined with other tracks. A track identifier consists of a tuple of attributes. Each track in and experimental design has the same number of labels in the same order. In the example above, there are three attributes: tissue, condition and replicate. Identifiers are thus `('liver', 'stimulated','R1')` or `('heart','unstimulated','R2')`.

The same track can be represented by different names depending on context, for example when it is used as a filename or a database table. As filename, the track `('heart','unstimulated','R2')` is rendered as `heart-unstimulated-R2` (avoiding spaces), while as a table, it reads `heart_unstimulated_R2`, avoiding `-+.`. The `Sample` class provides convenience methods to convert names from one context to another.

Track containers A container of type `Tracks` stores one or more objects of type `Sample`.

Aggregates Tracks can be combined into aggregates. Aggregation is indicated by the `agg` keyword.

For example, the `liver-stimulated-agg` aggregate combines the tracks `liver-stimulated-R1` and `liver-stimulated-R2`. The aggregate `agg-stimulated-agg` combines all replicates and all tissues (`liver-stimulated-R1`, `liver-stimulated-R2`, `heart-stimulated-R1`, `heart-stimulated-R2`)

Usage

Defining tracks and aggregates To use tracks, you need to first define a new `Sample`. In the example above with the attributes tissue, condition and replicate, the `Sample` could be:

```
import PipelineTracks

class MySample( PipelineTracks.Sample ):
     attributes = ( "tissue", "condition", "replicate" )
```

Once defined, you can add tracks to a `tracks` container. For example:

```
TRACKS = PipelineTracks.Tracks( MySample ).loadFromDirectory( glob.glob( "*.fastq.gz" ),
                                                              pattern = "(\S+).fastq.gz" )
```

will collect all files ending in `.fastq.gz`. The track identifiers will be derived by removing the `fastq.gz` suffix. The variable `TRACKS` contains all the tracks derived from files ending in `*.fastq.gz`:

```
>>> print TRACKS
[liver-stimulated-R2, heart-stimulated-R2, liver-stimulated-R1, liver-unstimulated-R1, heart-unstimul
```

To build aggregates, use `PipelineTracks.Aggregate`. The following combines replicates for each experiment:

```
EXPERIMENTS = PipelineTracks.Aggregate( TRACKS, labels = ("condition", "tissue" ) )
```

Aggregates are simply containers of associated data sets. To get a list of experiments, type:

```
>>> EXPERIMENTS = PipelineTracks.Aggregate( TRACKS, labels = ("condition", "tissue" ) )
>>> print list(EXPERIMENT)
[heart-stimulated-agg, heart-unstimulated-agg, liver-stimulated-agg, liver-unstimulated-agg]
```

or:

```
>>> print EXPERIMENT.keys()
[heart-stimulated-agg, heart-unstimulated-agg, liver-stimulated-agg, liver-unstimulated-agg]
```

To obtain all replicates in the experiment `heart-stimulated`, use dictionary access:

```
>>> print EXPERIMENTS['heart-stimulated-agg']
[heart-stimulated-R2, heart-stimulated-R1]
```

The returned objects are tracks. To use a *track* as a tablename or as a file, use data access functions `Sample.asTable()` or `Sample.asFile()`, respectively:

```
>>> print [x.asFile() for x in EXPERIMENTS['heart-stimulated-agg'] ]
['heart-stimulated-R2', 'heart-stimulated-R1']

>>> print [str(x) for x in EXPERIMENTS['heart-stimulated-agg'] ]
['heart-stimulated-R2', 'heart-stimulated-R1']

>>> print [x.asTable() for x in EXPERIMENTS['heart-stimulated-agg'] ]
['heart_stimulated_R2', 'heart_stimulated_R1']
```

Note how the - is converted to _ as the former are illegal as SQL table names.

The default representation is file-based. By using the class method:

```
MySample.setDefault( "asTable" )
```

the default representation can be changed for all tracks simultaneously.

You can have multiple aggregates. For example, some tasks might require all conditions or all tissues:

```
CONDITIONS = PipelineTracks.Aggregate( TRACKS, labels = ("condition", ) )
TISSUES = PipelineTracks.Aggregate( TRACKS, labels = ("tissue", ) )
```

You can have several `Tracks` within a directory. `Tracks` are simply containers and as such do not have any actions associated with them.

Using tracks in pipelines Unfortunately, tracks and aggregates do not work yet directly as ruffus task lists. Instead, they need to be converted to files explicitely using list comprehensions.

If you wanted to process all tracks separately, use:

```
@files( [ ("%s.fastq.gz" % x.asFile(),
           "%s.qc" % x.asFile()) for x in TRACKS ] )
def performQC( infile, outfile ):
    ....
```

The above statement will create the following list of input/output files for the `performQC` task:

```
[ ( "liver-stimulated-R1.fastq.gz", "liver-stimulated-R1.qc" )
  ( "liver-stimulated-R2.fastq.gz" , "liver-stimulated-R2.qc" ),
  ...
]
```

Using aggregates works similarly, though you will need to create the file lists yourself using nested list comprehensions. The following creates an analysis per experimemnt:

```
@files( [( ([ "%s.fastq.gz" % y.asFile() for y in EXPERIMENTS[x]]),
                 "%s.out" % x.asFile())
                 for x in EXPERIMENTS ] )
```

```
def checkReproducibility( infiles, outfile ):
    ....
```

The above statement will create the following list of input/output files:

```
[ ( ( "liver-stimulated-R1.fastq.gz", "liver-stimulated-R2.fastq.gz" ), "liver-stimulated-agg.out" ),
  ( ( "liver-unstimulated-R1.fastq.gz", "liver-unstimulated-R2.fastq.gz" ), "liver-unstimulated-agg.
  ( ( "heart-stimulated-R1.fastq.gz", "heart-stimulated-R2.fastq.gz" ), "heart-stimulated-agg.out" ),
  ( ( "heart-unstimulated-R1.fastq.gz", "heart-unstimulated-R2.fastq.gz" ), "heart-unstimulated-agg.
]
```

The above code makes sure that the file dependencies are observed. Thus, if `heart-stimulated-R1.fastq.gz` changes, only `heart-stimulated-agg.out` will be re-computed.

Tracks and aggregates can be used within a task. The following code will collect all replicates for the experiment `liver-stimulated-agg`

```
>>> track = TRACKS.factory( filename = "liver-stimulated-agg" )
>>> replicates = PipelineTracks.getSamplesInTrack( track, TRACKS )
>>> print replicates
[liver-stimulated-R2, liver-stimulated-R1]
```

API

class `PipelineTracks.`**`Sample`**(*filename=None*)

Bases: `object`

a sample/track with one attribute called `experiment`.

create a new Sample.

If filename is given, the sample name will be derived from *filename*.

`clone`()
return a copy of self.

`asFile`()
return sample as a filename

`asTable`()
return sample as a tablename

`asR`()
return sample as valid R label

`fromFile`(*fn*)
build sample from filename *fn*

`fromTable`(*tn*)
build sample from tablename *tn*

`fromR`(*rn*)
build sample from R name *rn*

`asAggregate`(**args*)
return a new aggregate Sample.

`toLabels`()
return attributes that this track is an aggregate of.

classmethod setDefault(*representation=None*)
set default representation for tracks to *representation*. If *represenation* is None, the representation will be set to the library default (asFile()).

class PipelineTracks.**Sample3**(*filename=None*)
Bases: PipelineTracks.Sample

a sample/track with three attributes: tissue, condition and replicate.

create a new Sample.

If filename is given, the sample name will be derived from *filename*.

asAggregate(**args*)
return a new aggregate Sample.

asFile()
return sample as a filename

asR()
return sample as valid R label

asTable()
return sample as a tablename

clone()
return a copy of self.

fromFile(*fn*)
build sample from filename *fn*

fromR(*rn*)
build sample from R name *rn*

fromTable(*tn*)
build sample from tablename *tn*

classmethod setDefault(*representation=None*)
set default representation for tracks to *representation*. If *represenation* is None, the representation will be set to the library default (asFile()).

toLabels()
return attributes that this track is an aggregate of.

class PipelineTracks.**Tracks**(*factory=<class 'PipelineTracks.Sample'>*)
a collection of tracks.

create a new container.

New tracks are derived using *factory*.

factory
alias of Sample

loadFromDirectory(*files*, *pattern*, *exclude=None*)
load tracks from a list of files, applying pattern.

Pattern is a regular expression with at at least one group, for example `(.*).gz`.

If set, exclude files matching regular expression in *exclude*.

getTracks(*pattern=None*)
return all tracks in container.

PipelineTracks.**getSamplesInTrack**(*track*, *tracks*)
return all tracks in *tracks* that constitute *track*.

PipelineTracks.py - Definition of tracks in pipelines

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Motivation

A pipeline typically processes the data streams from several experimental data sources. These data streams are usually processed separately (processing, quality control) and as aggregates. For example, consider the following experimental layout:

Filename	*Content*
liver-stimulated-R1	liver, stimulated, replicate 1
liver-stimulated-R2	liver, stimulated, replicate 2
liver-unstimulated-R1	liver, unstimulated, replicate 1
liver-unstimulated-R2	liver, unstimulated, replicate 2
heart-stimulated-R1	heart, stimulated, replicate 1
heart-stimulated-R2	heart, stimulated, replicate 2
heart-unstimulated-R1	heart, unstimulated, replicate 1
heart-unstimulated-R2	heart, unstimulated, replicate 2

The experiment measured in two tissues with two conditions with two replicates each giving eight data streams. During the analysis, the streams are merged in a variety of combinations:

- unmerged for initial processing, QC, etc.
- by replicates to assess reproducibility of measurements
- by condition to assess the size of the response to the stimulus
- by tissue to assess differences between tissue and address the biological question.

The crossing of data streams complicates the building of pipelines, especially as no two experiments are the same. The `PipelineTracks` module assists in controlling these data streams. This module provides some tools to map tracks to different representations and to group them in flexible ways in order to provide convenient short-cuts in pipelines.

There are three class within `PipelineTracks`: `Sample`, `Tracks` and `Aggregate`.

A Track The basic atomic data structure is a `Sample` or *track*. A *track* is a single measurement that can be combined with other tracks. A track identifier consists of a tuple of attributes. Each track in and experimental design has the same number of labels in the same order. In the example above, there are three attributes: tissue, condition and replicate. Identifiers are thus `('liver', 'stimulated','R1')` or `('heart','unstimulated','R2')`.

The same track can be represented by different names depending on context, for example when it is used as a filename or a database table. As filename, the track `('heart','unstimulated','R2')` is rendered as `heart-unstimulated-R2` (avoiding spaces), while as a table, it reads `heart_unstimulated_R2`, avoiding `-+.`. The `Sample` class provides convenience methods to convert names from one context to another.

Track containers A container of type `Tracks` stores one or more objects of type `Sample`.

Aggregates Tracks can be combined into aggregates. Aggregation is indicated by the `agg` keyword.

For example, the `liver-stimulated-agg` aggregate combines the tracks `liver-stimulated-R1` and `liver-stimulated-R2`. The aggregate `agg-stimulated-agg` combines all replicates and all tissues (`liver-stimulated-R1`, `liver-stimulated-R2`, `heart-stimulated-R1`, `heart-stimulated-R2`)

Usage

Defining tracks and aggregates To use tracks, you need to first define a new `Sample`. In the example above with the attributes tissue, condition and replicate, the `Sample` could be:

```
import PipelineTracks

class MySample( PipelineTracks.Sample ):
     attributes = ( "tissue", "condition", "replicate" )
```

Once defined, you can add tracks to a `tracks` container. For example:

```
TRACKS = PipelineTracks.Tracks( MySample ).loadFromDirectory( glob.glob( "*.fastq.gz" ),
                                                              pattern = "(\S+).fastq.gz" )
```

will collect all files ending in `.fastq.gz`. The track identifiers will be derived by removing the `fastq.gz` suffix. The variable `TRACKS` contains all the tracks derived from files ending in `*.fastq.gz`:

```
>>> print TRACKS
[liver-stimulated-R2, heart-stimulated-R2, liver-stimulated-R1, liver-unstimulated-R1, heart-unstimul
```

To build aggregates, use `PipelineTracks.Aggregate`. The following combines replicates for each experiment:

```
EXPERIMENTS = PipelineTracks.Aggregate( TRACKS, labels = ("condition", "tissue" ) )
```

Aggregates are simply containers of associated data sets. To get a list of experiments, type:

```
>>> EXPERIMENTS = PipelineTracks.Aggregate( TRACKS, labels = ("condition", "tissue" ) )
>>> print list(EXPERIMENT)
[heart-stimulated-agg, heart-unstimulated-agg, liver-stimulated-agg, liver-unstimulated-agg]
```

or:

```
>>> print EXPERIMENT.keys()
[heart-stimulated-agg, heart-unstimulated-agg, liver-stimulated-agg, liver-unstimulated-agg]
```

To obtain all replicates in the experiment `heart-stimulated`, use dictionary access:

```
>>> print EXPERIMENTS['heart-stimulated-agg']
[heart-stimulated-R2, heart-stimulated-R1]
```

The returned objects are tracks. To use a *track* as a tablename or as a file, use data access functions `Sample.asTable()` or `Sample.asFile()`, respectively:

```
>>> print [x.asFile() for x in EXPERIMENTS['heart-stimulated-agg'] ]
['heart-stimulated-R2', 'heart-stimulated-R1']

>>> print [str(x) for x in EXPERIMENTS['heart-stimulated-agg'] ]
['heart-stimulated-R2', 'heart-stimulated-R1']

>>> print [x.asTable() for x in EXPERIMENTS['heart-stimulated-agg'] ]
['heart_stimulated_R2', 'heart_stimulated_R1']
```

Note how the - is converted to _ as the former are illegal as SQL table names.

The default representation is file-based. By using the class method:

```
MySample.setDefault( "asTable" )
```

the default representation can be changed for all tracks simultaneously.

You can have multiple aggregates. For example, some tasks might require all conditions or all tissues:

```
CONDITIONS = PipelineTracks.Aggregate( TRACKS, labels = ("condition", ) )
TISSUES = PipelineTracks.Aggregate( TRACKS, labels = ("tissue", ) )
```

You can have several `Tracks` within a directory. `Tracks` are simply containers and as such do not have any actions associated with them.

Using tracks in pipelines Unfortunately, tracks and aggregates do not work yet directly as ruffus task lists. Instead, they need to be converted to files explicitely using list comprehensions.

If you wanted to process all tracks separately, use:

```
@files( [ ("%s.fastq.gz" % x.asFile(),
           "%s.qc" % x.asFile()) for x in TRACKS ] )
def performQC( infile, outfile ):
    ....
```

The above statement will create the following list of input/output files for the `performQC` task:

```
[ ( "liver-stimulated-R1.fastq.gz", "liver-stimulated-R1.qc" )
  ( "liver-stimulated-R2.fastq.gz" , "liver-stimulated-R2.qc" ),
  ...
]
```

Using aggregates works similarly, though you will need to create the file lists yourself using nested list comprehensions. The following creates an analysis per experimemnt:

```
@files( [( ([ "%s.fastq.gz" % y.asFile() for y in EXPERIMENTS[x]]),
                  "%s.out" % x.asFile())
                  for x in EXPERIMENTS ] )
def checkReproducibility( infiles, outfile ):
    ....
```

The above statement will create the following list of input/output files:

```
[ ( ( "liver-stimulated-R1.fastq.gz", "liver-stimulated-R2.fastq.gz" ), "liver-stimulated-agg.out" ),
  ( ( "liver-unstimulated-R1.fastq.gz", "liver-unstimulated-R2.fastq.gz" ), "liver-unstimulated-agg.
  ( ( "heart-stimulated-R1.fastq.gz", "heart-stimulated-R2.fastq.gz" ), "heart-stimulated-agg.out" ),
  ( ( "heart-unstimulated-R1.fastq.gz", "heart-unstimulated-R2.fastq.gz" ), "heart-unstimulated-agg.
]
```

The above code makes sure that the file dependencies are observed. Thus, if `heart-stimulated-R1.fastq.gz` changes, only `heart-stimulated-agg.out` will be re-computed.

Tracks and aggregates can be used within a task. The following code will collect all replicates for the experiment `liver-stimulated-agg`

```
>>> track = TRACKS.factory( filename = "liver-stimulated-agg" )
>>> replicates = PipelineTracks.getSamplesInTrack( track, TRACKS )
>>> print replicates
[liver-stimulated-R2, liver-stimulated-R1]
```

API

class PipelineTracks.**Sample**(*filename=None*)
Bases: object

a sample/track with one attribute called experiment.

create a new Sample.

If filename is given, the sample name will be derived from *filename*.

clone()
return a copy of self.

asFile()
return sample as a filename

asTable()
return sample as a tablename

asR()
return sample as valid R label

fromFile(*fn*)
build sample from filename *fn*

fromTable(*tn*)
build sample from tablename *tn*

fromR(*rn*)
build sample from R name *rn*

asAggregate(**args*)
return a new aggregate Sample.

toLabels()
return attributes that this track is an aggregate of.

classmethod setDefault(*representation=None*)
set default representation for tracks to *representation*. If *represenation* is None, the representation will be set to the library default (asFile()).

class PipelineTracks.**Sample3**(*filename=None*)
Bases: PipelineTracks.Sample

a sample/track with three attributes: tissue, condition and replicate.

create a new Sample.

If filename is given, the sample name will be derived from *filename*.

asAggregate(**args*)
return a new aggregate Sample.

asFile()
return sample as a filename

asR()
return sample as valid R label

asTable()
return sample as a tablename

clone()
return a copy of self.

fromFile(*fn*)
build sample from filename *fn*

fromR(*rn*)
build sample from R name *rn*

fromTable(*tn*)
build sample from tablename *tn*

classmethod setDefault(*representation=None*)
set default representation for tracks to *representation*. If *represenation* is None, the representation will be set to the library default (asFile()).

toLabels()
return attributes that this track is an aggregate of.

class PipelineTracks.**Tracks**(*factory=<class 'PipelineTracks.Sample'>*)
a collection of tracks.

create a new container.

New tracks are derived using *factory*.

factory
alias of Sample

loadFromDirectory(*files*, *pattern*, *exclude=None*)
load tracks from a list of files, applying pattern.

Pattern is a regular expression with at at least one group, for example (.*).gz.

If set, exclude files matching regular expression in *exclude*.

getTracks(*pattern=None*)
return all tracks in container.

PipelineTracks.**getSamplesInTrack**(*track*, *tracks*)
return all tracks in *tracks* that constitute *track*.

2.2.8 Plotting

modules/GDLDraw.rst

2.2.9 Other

RLE.py - a simple run length encoder

Author

Release Id

Date December 09, 2013

Tags Python

Taken from: http://rosettacode.org/wiki/Run-length_encoding#Python

RLE.**encode**(*input_array*)
encode array or string.

return tuples of (count, value).

```
>>> encode(array.array( "i", (10,10,10,10,20,20,20,20) ) )
[(4, 10), (4, 20)]

>>> encode("aaaaahhhhhhmmmmmmmuiiiiiiiiaaaaaa")
[(5, 'a'), (6, 'h'), (7, 'm'), (1, 'u'), (7, 'i'), (6, 'a')]
```

RLE.**decode**(*lst*, *typecode*)
decode to array

```
>>> decode( [(4, 10), (4, 20)], typecode="i" )
array('i', [10, 10, 10, 10, 20, 20, 20, 20])

>>> decode( [(5, 'a'), (6, 'h'), (7, 'm'), (1, 'u'), (7, 'i'), (6, 'a')], typecode="c" )
array('c', 'aaaaahhhhhhmmmmmmmuiiiiiiiiaaaaaa')
```

RLE.**compress**(*input_string*, *bytes=1*)
return compressed stream.

SVGdraw.py - generate SVG drawings

Author Fedor Baart & Hans de Wit

Release Id

Date December 09, 2013

Tags Python

This module has been copied from 3rd party resources.

SVGdraw uses an object model drawing and a method toXML to create SVG graphics by using easy to use classes and methods usualy you start by creating a drawing eg

> d=drawing() #then you create a SVG root element s=svg() #then you add some elements eg a circle and add it to the svg root element c=circle() #you can supply attributes by using named arguments. c=circle(fill='red',stroke='blue') #or by updating the attributes attribute: c.attributes['stroke-width']=1 s.addElement(c) #then you add the svg root element to the drawing d.setSVG(s) #and finaly you xmlify the drawing d.toXml()

this results in the svg source of the drawing, which consists of a circle on a white background. Its as easy as that;) This module was created using the SVG specification of www.w3c.org and the O'Reilly (www.oreilly.com) python books as information sources. A svg viewer is available from www.adobe.com

class SVGdraw.**pathdata**(*x=None*, *y=None*)
class used to create a pathdata object which can be used for a path. although most methods are pretty straightforward it might be useful to look at the SVG specification.

closepath()
ends the path

move(*x*, *y*)
move to absolute

relmove(*x*, *y*)
move to relative

line(*x*, *y*)
line to absolute

relline(*x*, *y*)
line to relative

hline(*x*)
horizontal line to absolute

relhline(*x*)
horizontal line to relative

vline(*y*)
verical line to absolute

relvline(*y*)
vertical line to relative

bezier(*x1*, *y1*, *x2*, *y2*, *x*, *y*)
bezier with xy1 and xy2 to xy absolut

relbezier(*x1*, *y1*, *x2*, *y2*, *x*, *y*)
bezier with xy1 and xy2 to xy relative

smbezier(*x2*, *y2*, *x*, *y*)
smooth bezier with xy2 to xy absolut

relsmbezier(*x2*, *y2*, *x*, *y*)
smooth bezier with xy2 to xy relative

qbezier(*x1*, *y1*, *x*, *y*)
quadratic bezier with xy1 to xy absolut

relqbezier(*x1*, *y1*, *x*, *y*)
quadratic bezier with xy1 to xy relative

smqbezier(*x*, *y*)
smooth quadratic bezier to xy absolut

relsmqbezier(*x*, *y*)
smooth quadratic bezier to xy relative

ellarc(*rx*, *ry*, *xrot*, *laf*, *sf*, *x*, *y*)
elliptival arc with rx and ry rotating with xrot using large-arc-flag and sweep-flag to xy absolut

relellarc(*rx*, *ry*, *xrot*, *laf*, *sf*, *x*, *y*)
elliptival arc with rx and ry rotating with xrot using large-arc-flag and sweep-flag to xy relative

class SVGdraw.**SVGelement**(*type=''*, *attributes=None*, *elements=None*, *text=''*, *namespace=''*, *cdata=None*, ***args*)
SVGelement(type,attributes,elements,text,namespace,**args) Creates a arbitrary svg element and is intended to be subclassed not used on its own. This element is the base of every svg element it defines a class which resembles a xml-element. The main advantage of this kind of implementation is that you don't have to create a toXML method for every different graph object. Every element consists of a type, attribute, optional subelements, optional text and an optional namespace. Note the elements==None, if elements = None:self.elements=[] construction. This is done because if you default to elements=[] every object has a reference to the same empty list.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**tspan**(*text=None*, ***args*)
Bases: SVGdraw.SVGelement

ts=tspan(text='',**args)

a tspan element can be used for applying formatting to a textsection usage: ts=tspan('this text is bold') ts.attributes['font-weight']='bold' st=spannedtext() st.addtspan(ts) t=text(3,5,st)

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**tref**(*link*, ***args*)
Bases: SVGdraw.SVGelement

tr=tref(link='',**args)

a tref element can be used for referencing text by a link to its id. usage: tr=tref('#linktotext') st=spannedtext() st.addtref(tr) t=text(3,5,st)

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**spannedtext**(*textlist=None*)
st=spannedtext(textlist=[])

a spannedtext can be used for text which consists of text, tspan's and tref's You can use it to add to a text element or path element. Don't add it directly to a svg or a group element. usage:

ts=tspan('this text is bold') ts.attributes['font-weight']='bold' tr=tref('#linktotext') tr.attributes['fill']='red' st=spannedtext() st.addtspan(ts) st.addtref(tr) st.addtext('This text is not bold') t=text(3,5,st)

class SVGdraw.**rect**(*x=None*, *y=None*, *width=None*, *height=None*, *fill=None*, *stroke=None*, *stroke_width=None*, ***args*)
Bases: SVGdraw.SVGelement

r=rect(width,height,x,y,fill,stroke,stroke_width,**args)

a rectangle is defined by a width and height and a xy pair

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**ellipse**(*cx=None*, *cy=None*, *rx=None*, *ry=None*, *fill=None*, *stroke=None*, *stroke_width=None*, ***args*)
Bases: SVGdraw.SVGelement

e=ellipse(rx,ry,x,y,fill,stroke,stroke_width,**args)

an ellipse is defined as a center and a x and y radius.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**circle**(*cx=None*, *cy=None*, *r=None*, *fill=None*, *stroke=None*, *stroke_width=None*, ***args*)
Bases: SVGdraw.SVGelement

c=circle(x,y,radius,fill,stroke,stroke_width,**args)

The circle creates an element using a x, y and radius values eg

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**point**(*x*, *y*, *fill='black'*, ***args*)
Bases: SVGdraw.circle

p=point(x,y,color)

A point is defined as a circle with a size 1 radius. It may be more efficient to use a very small rectangle if you use many points because a circle is difficult to render.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**line**(*x1=None*, *y1=None*, *x2=None*, *y2=None*, *stroke=None*, *stroke_width=None*, ***args*)
Bases: SVGdraw.SVGelement

l=line(x1,y1,x2,y2,stroke,stroke_width,**args)

A line is defined by a begin x,y pair and an end x,y pair

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**polyline**(*points*, *fill=None*, *stroke=None*, *stroke_width=None*, ***args*)
Bases: SVGdraw.SVGelement

pl=polyline([[x1,y1],[x2,y2],...],fill,stroke,stroke_width,**args)

a polyline is defined by a list of xy pairs

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**polygon**(*points*, *fill=None*, *stroke=None*, *stroke_width=None*, ***args*)
Bases: SVGdraw.SVGelement

pl=polyline([[x1,y1],[x2,y2],...],fill,stroke,stroke_width,**args)

a polygon is defined by a list of xy pairs

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**path**(*pathdata*, *fill=None*, *stroke=None*, *stroke_width=None*, *id=None*, ***args*)
Bases: SVGdraw.SVGelement

p=path(path,fill,stroke,stroke_width,**args)

a path is defined by a path object and optional width, stroke and fillcolor

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**text**(*x=None*, *y=None*, *text=None*, *font_size=None*, *font_family=None*, *text_anchor=None*, *font_style=None*, ***args*)
Bases: SVGdraw.SVGelement

t=text(x,y,text,font_size,font_family,**args)

a text element can bge used for displaying text on the screen

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**textpath**(*link*, *text=None*, ***args*)
Bases: SVGdraw.SVGelement

tp=textpath(text,link,**args)

a textpath places a text on a path which is referenced by a link.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**pattern**(*x=None*, *y=None*, *width=None*, *height=None*, *patternUnits=None*, ***args*)
Bases: SVGdraw.SVGelement

p=pattern(x,y,width,height,patternUnits,**args)

A pattern is used to fill or stroke an object using a pre-defined graphic object which can be replicated ("tiled") at fixed intervals in x and y to cover the areas to be painted.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**title**(*text=None*, ***args*)
Bases: SVGdraw.SVGelement

t=title(text,**args)

a title is a text element. The text is displayed in the title bar add at least one to the root svg element

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**description**(*text=None*, ***args*)
Bases: SVGdraw.SVGelement

d=description(text,**args)

a description can be added to any element and is used for a tooltip Add this element before adding other elements.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**lineargradient**(*x1=None*, *y1=None*, *x2=None*, *y2=None*, *id=None*, ***args*)
Bases: SVGdraw.SVGelement

lg=lineargradient(x1,y1,x2,y2,id,**args)

defines a lineargradient using two xy pairs. stop elements van be added to define the gradient colors.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**radialgradient**(*cx=None*, *cy=None*, *r=None*, *fx=None*, *fy=None*, *id=None*, ***args*)
Bases: SVGdraw.SVGelement

rg=radialgradient(cx,cy,r,fx,fy,id,**args)

defines a radial gradient using a outer circle which are defined by a cx,cy and r and by using a focalpoint. stop elements van be added to define the gradient colors.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**stop**(*offset*, *stop_color=None*, ***args*)
Bases: SVGdraw.SVGelement

st=stop(offset,stop_color,**args)

Puts a stop color at the specified radius

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**style**(*type*, *cdata=None*, ***args*)
Bases: SVGdraw.SVGelement

st=style(type,cdata=None,**args)

Add a CDATA element to this element for defing in line stylesheets etc..

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**image**(*url*, *x=None*, *y=None*, *width=None*, *height=None*, ***args*)
Bases: SVGdraw.SVGelement

im=image(url,width,height,x,y,**args)

adds an image to the drawing. Supported formats are .png, .jpg and .svg.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**cursor**(*url*, ***args*)
Bases: SVGdraw.SVGelement

c=cursor(url,**args)

defines a custom cursor for a element or a drawing

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**marker**(*id=None*, *viewBox=None*, *refx=None*, *refy=None*, *markerWidth=None*, *markerHeight=None*, ***args*)
Bases: SVGdraw.SVGelement

m=marker(id,viewbox,refX,refY,markerWidth,markerHeight,**args)

defines a marker which can be used as an endpoint for a line or other pathtypes add an element to it which should be used as a marker.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**group**(*id=None*, ***args*)
Bases: SVGdraw.SVGelement

g=group(id,**args)

a group is defined by an id and is used to contain elements g.addElement(SVGelement)

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**symbol**(*id=None*, *viewBox=None*, ***args*)
Bases: SVGdraw.SVGelement

sy=symbol(id,viewbox,**args)

defines a symbol which can be used on different places in your graph using the use element. A symbol is not rendered but you can use 'use' elements to display it by referencing its id. sy.addElement(SVGelement)

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**defs**(***args*)
Bases: SVGdraw.SVGelement

d=defs(**args)

container for defining elements

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**switch**(***args*)
Bases: SVGdraw.SVGelement

sw=switch(**args)

Elements added to a switch element which are "switched" by the attributes requiredFeatures, requiredExtensions and systemLanguage. Refer to the SVG specification for details.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**use**(*link*, *x=None*, *y=None*, *width=None*, *height=None*, ***args*)
Bases: SVGdraw.SVGelement

u=use(link,x,y,width,height,''**args'')

references a symbol by linking to its id and its position, height and width

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**link**(*link=''*, ***args*)
Bases: SVGdraw.SVGelement

a=link(url,''**args'')

a link is defined by a hyperlink. add elements which have to be linked a.addElement(SVGelement)

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**view**(*id=None*, ***args*)
Bases: SVGdraw.SVGelement

v=view(id,''**args'')

a view can be used to create a view with different attributes

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**script**(*type*, *cdata=None*, ***args*)
Bases: SVGdraw.SVGelement

sc=script(type,type,cdata,''**args'')

adds a script element which contains CDATA to the SVG drawing

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**animate**(*attribute*, *fr=None*, *to=None*, *dur=None*, ***args*)
Bases: SVGdraw.SVGelement

an=animate(attribute,from,to,during,''**args'')

animates an attribute.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**animateMotion**(*pathdata*, *dur*, ***args*)
Bases: SVGdraw.SVGelement

an=animateMotion(pathdata,dur,''**args'')

animates a SVGelement over the given path in dur seconds

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**animateTransform**(*type=None*, *fr=None*, *to=None*, *dur=None*, ***args*)
Bases: SVGdraw.SVGelement

antr=animateTransform(type,from,to,dur,‘‘**args‘‘)

transform an element from and to a value.

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**animateColor**(*attribute*, *type=None*, *fr=None*, *to=None*, *dur=None*, ***args*)
Bases: SVGdraw.SVGelement

ac=animateColor(attribute,type,from,to,dur,‘‘**args‘‘)

Animates the color of a element

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**set**(*attribute*, *to=None*, *dur=None*, ***args*)
Bases: SVGdraw.SVGelement

st=set(attribute,to,during,‘‘**args‘‘)

sets an attribute to a value for a

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**svg**(*viewBox=None*, *width=None*, *height=None*, ***args*)
Bases: SVGdraw.SVGelement

s=svg(viewbox,width,height,‘‘**args‘‘)

a svg or element is the root of a drawing add all elements to a svg element. You can have different svg elements in one svg file s.addElement(SVGelement)

eg d=drawing() s=svg((0,0,100,100),‘100%’,‘100%’) c=circle(50,50,20) s.addElement(c) d.setSVG(s) d.toXml()

addElement(*SVGelement*)
adds an element to a SVGelement

SVGelement.addElement(SVGelement)

class SVGdraw.**drawing**
d=drawing()

this is the actual SVG document. It needs a svg element as a root. Use the addSVG method to set the svg to the root. Use the toXml method to write the SVG source to the screen or to a file d=drawing() d.addSVG(svg) d.toXml(optionalfilename)

SetTools.py - Tools for working on sets

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Many of the functions in this module precede the `set` datatype in python.

Code

`SetTools.`**`combinations`**(*list_of_sets*)
create all combinations of a list of sets

returns a list of tuples (set_composition, union, intersection)

`SetTools.`**`writeSets`**(*outfile*, *list_of_sets*, *labels=None*)
output a list of sets as a tab-separated file.

labels is a list of set labels.

`SetTools.`**`unionIntersectionMatrix`**(*list_of_sets*)
build union and intersection of a list of sets.

return a matrix with the upper diagonal the union and the lower diagonal the intersection.

`SetTools.`**`CompareSets`**(*set1*, *set2*)
returns the union and the disjoint members of two sets. The sets have to be sorted.

`SetTools.`**`MakeListComprehensionFunction`**(*name*, *nsets*)
Returns a function applicable to exactly <nsets> sets. The returned function has the signature F(set0, set1, ..., set<nsets>) and returns a list of all element combinations as tuples. A set may be any iterable object.

Sockets.py - working with sockets

This class allows you to send variable length strings over a socket. As I am new at this, it is probably not efficient and 100% fault tolerant.

Look at the end of the file for example usage.

GraphTools.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

exception `GraphTools.`**`Error`**
Bases: `exceptions.Exception`

Base class for exceptions in this module.

exception `GraphTools.`**`InputError`**(*message*)
Bases: `GraphTools.Error`

Exception raised for errors in the input.

Attributes: expression – input expression in which the error occurred message – explanation of the error

exception GraphTools.**RuntimeError**(*message*)
Bases: GraphTools.Error

Exception raised for errors in the input.

Attributes: expression – input expression in which the error occurred message – explanation of the error

Cluster.py - module for running a job in parallel on the cluster

exception Cluster.**Error**
Bases: exceptions.Exception

Base class for exceptions in this module.

exception Cluster.**ClusterError**(*message*)
Bases: Cluster.Error

Exception raised for errors in the input.

Attributes: expression – input expression in which the error occurred message – explanation of the error

2.2.10 Obsolete

Intervalls.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

Intervalls.**CombineIntervallsLarge**(*intervalls*)
combine intervalls. Overlapping intervalls are concatenated into larger intervalls.

Intervalls.**ComplementIntervalls**(*intervalls*, *first=None*, *last=None*)
complement a list of intervalls with intervalls not in list.

Intervalls.**AddComplementIntervalls**(*intervalls*, *first=None*, *last=None*)
complement a list of intervalls with intervalls not in list and return both.

Intervalls.**CombineIntervallsDistance**(*intervalls*, *min_distance*)
combine a list of non-overlapping intervalls, and merge those that are less than a certain distance apart.

Intervalls.**DeleteSmallIntervalls**(*intervalls*, *min_length*)
combine a list of non-overlapping intervalls, and delete those that are too small.

Intervalls.**CombineIntervallsOverlap**(*intervalls*)
combine intervalls. Overlapping intervalls are reduced to their intersection.

first_from, last_to contain region of current maximum overlapping segment. max_right is maximum extension of any sequence overlapping with current overlapping segment.

Intervalls.**RemoveIntervallsContained**(*intervalls*)
remove intervalls that are fully contained in another.

[(10, 100), (20, 50), (70, 120), (130, 200), (10, 50), (140, 210), (150, 200)]

results:

[(10, 100), (70, 120), (130, 200), (140, 210)]

Intervalls.**RemoveIntervallsSpanning**(*intervalls*)
remove intervalls that are full covering another, i.e. always keep the smallest.

[(10, 100), (20, 50), (70, 120), (40,80), (130, 200), (10, 50), (140, 210), (150, 200)]

result:

[(20, 50), (40, 80), (70, 120), (150, 200)]

Intervalls.**ShortenIntervallsOverlap**(*intervalls*, *to_remove*)
shorten intervalls, so that there is no overlap with another set of intervalls.

assumption: intervalls are not overlapping

Intervalls.**CalculateOverlap**(*intervalls1*, *intervalls2*)
calculate overlap between intervalls.

IntervallsWeigted.py - working with weigted intervals

Work with weighted invervalls. A weighted intervall is a tuple of the form (from,to,weight).

Funktions in this module take an optional function parameter object fct, that will give the weight if two intervalls are combined. The default is to add the weights of intervalls that are combined.

This module is work in progress. Finished are:

CombineIntervallsLarge RemoveIntervallsSpanning

IntervallsWeighted.**CombineIntervallsLarge**(*intervalls*, *fct=<function <lambda> at 0x406b0c8>*)
combine intervalls. Overlapping intervalls are concatenated into larger intervalls.

IntervallsWeighted.**ComplementIntervalls**(*intervalls*, *first=None*, *last=None*)
complement a list of intervalls with intervalls not in list.

IntervallsWeighted.**AddComplementIntervalls**(*intervalls*, *first=None*, *last=None*)
complement a list of intervalls with intervalls not in list and return both.

IntervallsWeighted.**CombineIntervallsDistance**(*intervalls*, *min_distance*)
combine a list of non-overlapping intervalls, and merge those that are less than a certain distance apart.

IntervallsWeighted.**DeleteSmallIntervalls**(*intervalls*, *min_length*)
combine a list of non-overlapping intervalls, and delete those that are too small.

IntervallsWeighted.**CombineIntervallsOverlap**(*intervalls*)
combine intervalls. Overlapping intervalls are reduced to their intersection.

first_from, last_to contain region of current maximum overlapping segment. max_right is maximum extension of any sequence overlapping with current overlapping segment.

IntervallsWeighted.**RemoveIntervallsContained**(*intervalls*)
remove intervalls that are fully contained in another.

[(10, 100), (20, 50), (70, 120), (130, 200), (10, 50), (140, 210), (150, 200)]

results:

[(10, 100), (70, 120), (130, 200), (140, 210)]

IntervallsWeighted.**RemoveIntervallsSpanning**(*intervalls*, *fct=<function <lambda> at 0x6ee39b0>*)
remove intervalls that are full covering another, i.e. always keep the smallest.

[(10, 100), (20, 50), (70, 120), (40,80), (130, 200), (10, 50), (140, 210), (150, 200)]

result:

[(20, 50), (40, 80), (70, 120), (150, 200)]

IntervallsWeighted.**ShortenIntervallsOverlap**(*intervalls*, *to_remove*)
shorten intervalls, so that there is no overlap with another set of intervalls.

assumption: intervalls are not overlapping

SaryFasta.py - index fasta files by suffix array

Subroutines for working on I/O of large genomic files.

Index a fasta file to retrieve sequences by suffix-array fragment search.

python SaryFasta.py [options] name [files]

SaryFasta.**getHID**(*sequence*)
returns a hash identifier for a sequence.

SaryFasta.**createDatabase**(*db*, *filenames*, *buf_size=400000000*, *force=False*, *regex_identifier=None*)
index files in filenames to create database.

buf_size: buffer size for a sary chunk.

Two new files are created - db.fasta and db_name.idx

regex_identifier: pattern to extract identifier from description line. If None, the part until the first white-space character is used.

SaryFasta.**benchmarkRandomFragment**(*fasta*, *size*)
returns a random fragment of size.

SaryFasta.**verify**(*reference*, *fasta*, *num_iterations*, *fragment_size*, *stdout=<open file '<stdout>', mode 'w' at 0x7f116e56d150>*, *quiet=False*)
verify two databases.

Get segment from fasta and check for presence in fasta2.

Fasta.py - Methods for dealing with fasta files.

Author

Release Id

Date December 09, 2013

Tags Python

SuffixArray.py - sarry frontend

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

2.2.11 Unsorted

Modules not sorted into categories.

BlatTest.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

exception `BlatTest.`**`Error`**
Bases: `exceptions.Exception`

Base class for exceptions in this module.

exception `BlatTest.`**`ParsingError`**(*message*, *line=None*)
Bases: `BlatTest.Error`

Exception raised for errors while parsing

Attributes: message – explanation of the error

class `BlatTest.`**`Match`**
a psl match.

Block coordinates are on the forward strand for target and on the forward/reverse strand for the query depending on the strand.

The fields mQueryFrom/To and mSbjctFrom/To are always on the forward strand.

`convertCoordinates`()
convert coordinates.

This rescales the block positions so that they start at 0 and converts the query to forward and the sbjct to forward/reverse coordinates.

About the psl psl format from the manual at http://genome.ucsc.edu/google/goldenPath/help/pslSpec.html

‘” In general the coordinates in psl files are “zero based half open.” The first base in a sequence is numbered zero rather than one. When representing a range the end coordinate is not included in the range. Thus the first 100 bases of a sequence are represented as 0-100, and the second 100 bases are represented as 100-200.

There is a another little unusual feature in the .psl format. It has to do with how coordinates are handled on the negative strand. In the qStart/qEnd fields the coordinates are where it matches from the point of view of the forward strand (even when the match is on the reverse strand). However on the qStarts[] list, the coordinates are reversed. ‘”‘

I want to work in forward coordinates for the query and forward/reverse coordinates for the sbjct.

For a negative strand match, the following is done:

- invert mSbjctFrom and mSbjctTo with mSbjctLength
- add block sizes to mQueryStarts and mSbjctStarts
- invert mQueryStarts and mSbjctStarts
- reverse blocksize, mQueryStarts and mSbjctStarts

switchTargetStrand()
switch the target strand.

Use in cases in which a feature has been defined on the negative target strand with reverse coordinates. The result will be the same alignment using forward coordinates on the target.

This method will also update the query strand and coordinates.

fromMaq(*maq*)
build BLAT entry from a MAQ match.

see Maq.py

getBlocks()
return a list of aligned blocks.

getMapQuery2Target()
return a map between query to target.

If the strand is "-", the coordinates for query are on the negative strand.

getMapTarget2Query()
return a map between target to query.

If the strand is "-", the coordinates for query are on the negative strand.

fromMap(*map_query2target*, *use_strand=None*)
return a map between query to target.

fromPair(*query_start*, *query_size*, *query_strand*, *query_seq*, *target_start*, *target_size*, *target_strand*, *target_seq*)
fill from two aligned sequences.

Note that sequences are case-sensitive.

BlatTest.**iterator**(*infile*)
iterate over the contents of a psl file.

BlatTest.**iterator_pslx**(*infile*)
iterate over the contents of a pslx file.

BlatTest.**iterator_target_overlap**(*infile*, *merge_distance*)
iterate over psl formatted infile and return blocks of target overlapping alignments.

BlatTest.**iterator_query_overlap**(*infile*, *merge_distance*)
iterate over psl formatted infile and return blocks of target overlapping alignments.

BlatTest.**iterator_test**(*infile*, *report_step=100000*)
only output parseable lines from infile.

BlatTest.**iterator_per_query**(*iterator_psl*)
iterate over the contents of a psl file per query

CBioPortal.py - Interface with the Sloan-Kettering cBioPortal webservice

Author Ian Sudbery

Release Id

Date December 09, 2013

Tags Python

The Sloan Kettering cBioPortal webservice provides access to a database of results of genomics experiments on various cancers. The database is organised into studies, each study contains a number of case lists, where each list contains the ids of a set of patients, and genetic profiles, each of which represents an assay conducted on the patients in the case list as part of the study.

The main class here is the CBioPortal class representing a connection to the cBioPortal Database. Query's are represented as methods of the class. Study ids or names or case lists can be provided to the constructor to the object, via the setDefaultStudy and setDefaultCaseList methods or to the indevidual query methods. Where ever possible the validity of parameters is checked *before* the query is executed.

Whenever a query requires a genetic profile id or a list of such ids, but none are given, the list of all profiles for which the show_in_analysis flag is set will be used.

All of the commands provided in the webservice are implemented here and as far as possible the name, syntax and paramter names of the query are identical to the raw commands to the webservice. These queries are:

- getCancerStudies,
- getCaseLists,
- getProfileData,
- getMutationData,
- getClinicalData,
- getProteinArrayInfo,
- getProteinArrayData,
- getLink,
- getOncoprintHTML.

In addition two new queries are implememented that are not part of the webservice:

- getPercentAltered and
- getTotalAltered

These emulate the function of the website where the percent of cases that show any alteration for the gene and profiles given are returned (getPercentAltered, or the percent of cases that show an alteration in any of the genes (getTotalAltered) is returned.

examples:

```
gene_list = [ "TP53",
"BCL2",
"MYC"  ]
portal = CBioPortal()
portal.setDefaultStudy(study = "prad_mskcc")
portal.setDefaultCaseList(case_set_id = "prad_all_complete")
portal.getPercentAltered(gene_list = gene_list)
```

or more tersely:

```
portal.CBioProtal()
portal.getPercentAltered(study = "prad_mskcc", case_set_id = "prad_all_complete",
                         gene_list = ["TP53","BCL2","MYC"],
                         genetic_profile_id =["prad_mskcc_mrna"])
```

Any warnings returned by the query are stored in CBioPortal.last_warnings.

Query's that would give too long an URL are split into smaller querys and the results combined transparently.

A commandline interface is provided for convenience, syntax:

```
python CBioPortal.py [options] command(s)
```

exception CBioPortal.**CDGSError**(*error*, *request*)
Bases: exceptions.Exception

exception that handles errors returned by querys in the database

CSV2DB.py - utilities for uploading a table to database

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Purpose

create a table from a csv separated file and load data into it.

This module supports backends for postgres and sqlite3. Column types are auto-detected.

Todo

Use file import where appropriate to speed up loading. Currently, this is not always the case.

Usage

Documentation

Code

CSV2DB.**executewait**(*dbhandle*, *statement*, *error*, *retry=False*, *wait=5*)
execute sql statement.

Retry on error, if retry is True. Returns a cursor object.

CSV2DB.**quoteRow**(*row*, *take*, *map_column2type*, *missing_values*, *null='NULL'*, *string_value='%s'*)
return a dictionary with properly quoted values.

GDLDraw.py -

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

Glam2.py - Parser for MAST files.

Author

Release Id

Date December 09, 2013

Tags Python

As of biopython 1.5.6, the MAST parser is broken.

`Glam2.`**`parse`**(*infile*)
parse Glam2 output.

Glam2Scan.py - Parser for MAST files

Author

Release Id

Date December 09, 2013

Tags Python

As of biopython 1.5.6, the MAST parser is broken.

class `Glam2Scan.`**`Match`**
a Glam2Scan entry.

`Glam2Scan.`**`parse`**(*infile*)
parse Glam2Scan output.

IGV.py - Simple wrapper to the IGV socket interface

Author Brent Pedersen

Release Id

Date December 09, 2013

Tags Python

This code was written by Brent Pedersen.

Downloaded from https://github.com/brentp/bio-playground/blob/master/igv/igv.py on Nov.30 2011.

class IGV.**IGV**(*host='127.0.0.1'*, *port=60151*, *snapshot_dir='/tmp/igv'*)

Bases: object

Simple wrapper to the IGV (http://www.broadinstitute.org/software/igv/home) socket interface (http://www.broadinstitute.org/software/igv/PortCommands)

requires:

1. **you have IGV running on your machine (launch with webstart here:** http://www.broadinstitute.org/software/igv/download)
2. **you have enabled port communication in** View -> Preferences... -> Advanced

Successful commands return 'OK'

example usage:

```
>>> igv = IGV()
>>> igv.genome('hg19')
'OK'
```

#>>> igv.load('http://www.broadinstitute.org/igvdata/1KG/pilot2Bams/NA12878.SLX.bam') 'OK'
>>> igv.go('chr1:45,600-45,800') 'OK'

#save as svg, png, or jpg

```
>>> igv.save('/tmp/r/region.svg')
'OK'
>>> igv.save('/tmp/r/region.png')
'OK'
```

 # go to a gene name.

```
>>> igv.go('muc5b')
'OK'
>>> igv.sort()
'OK'
>>> igv.save('muc5b.png')
'OK'
```

 # get a list of commands that will work as an IGV batch script.

```
>>> print "\n".join(igv.commands)
snapshotDirectory /tmp/igv
genome hg19
goto chr1:45,600-45,800
snapshotDirectory /tmp/r
snapshot region.svg
snapshot region.png
goto muc5b
sort base
snapshot muc5b.png
```

Note, there will be some delay as the browser has to load the annotations at each step.

sort(*option='base'*)

options is one of: base, position, strand, quality, sample, and readGroup.

Logfile.py - logfile parsing

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Purpose

Parse logfiles

Usage

Example:

```
python cgat_script_template.py --help
```

Type:

```
python cgat_script_template.py --help
```

for command line help.

Documentation

Code

class Logfile.**RuntimeInformation**

Bases: tuple

RuntimeInformation(script, options, jobid, host, has_finished, start_date, end_date, wall, utime, stime, cutime, cstime)

count(*value*) → integer – return number of occurrences of value

cstime

Alias for field number 11

cutime

Alias for field number 10

end_date

Alias for field number 6

has_finished

Alias for field number 4

host

Alias for field number 3

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

jobid

Alias for field number 2

options

Alias for field number 1

script
Alias for field number 0

start_date
Alias for field number 5

stime
Alias for field number 9

utime
Alias for field number 8

wall
Alias for field number 7

class Logfile.**LogFileDataLines**
Bases: Logfile.LogFileData

record lines.

MAST.py - Parser for MAST files

Author

Release Id

Date December 09, 2013

Tags Python

As of biopython 1.5.6, the MAST parser is broken.

class MAST.**Match**
a MAST entry.

MAST.**parse**(*infile*)
parse verbose MAST output.

MAST.**frequencies2logodds**(*counts*, *background_frequencies=None*)
write a motif from *counts* to outfile.

Counts should be a numpy matrix with *nalphabet* columns and *motif_width* rows.

MAST.**writeMast**(*outfile*, *logodds_matrix*, *alphabet*)
output logodds matrix in MAST format.

MAST.**writeTomTom**(*outfile*, *counts_matrix*, *header=False*)
output counts matrix in tomtom format.

output counts with columns as motif positions and rows as alphabet.

MAST.**sequences2motif**(*outfile*, *sequences*, *background_frequencies=None*, *format='MAST'*)
write a motif defined by a collection of sequences to outfile.

Tophat.py - working with tophat/cufflinks output files

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Code

Tophat.**parseTranscriptComparison**(*infile*)
read cufflinks 1.0.3 output in infile stream.

returns a two-level dictionary mapping with levels track and contig.

class Tophat.**Locus**
Bases: tuple

Locus(locus_id, contig, strand, start, end, transcript_ids, transcripts)

contig
Alias for field number 1

count(*value*) → integer – return number of occurrences of value

end
Alias for field number 4

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

locus_id
Alias for field number 0

start
Alias for field number 3

strand
Alias for field number 2

transcript_ids
Alias for field number 5

transcripts
Alias for field number 6

class Tophat.**Tracking**
Bases: tuple

Tracking(transfrag_id, locus_id, ref_gene_id, ref_transcript_id, code, transcripts)

code
Alias for field number 4

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

locus_id
Alias for field number 1

ref_gene_id
Alias for field number 2

ref_transcript_id
Alias for field number 3

transcripts
Alias for field number 5

transfrag_id
Alias for field number 0

Tophat.**TranscriptInfo**
alias of Transfrag

Tophat.**iterate_tracking**(*infile*)
parse .tracking output file from cuffcompare

returns iterator with list of loci.

Tophat.**iterate_locus**(*infile*)
parse .loci output file from cuffcompare

returns iterator with list of loci.

VCF.py - Tools for working with VCF files

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

The Variant Call Format (*vcf*) is described here:

http://www.1000genomes.org/wiki/doku.php?id=1000_genomes:analysis:vcf4.0

Code

MACS.py - Parser for MACS output

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

The Pipeline module contains various utility functions for parsing MACS output.

API

class WrapperMACS.**MacsPeak**
Bases: tuple

MacsPeak(contig, start, end, length, summit, tags, pvalue, fold, fdr)

contig
Alias for field number 0

count(*value*) → integer – return number of occurrences of value

end
Alias for field number 2

fdr
Alias for field number 8

fold
Alias for field number 7

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

length
Alias for field number 3

pvalue
Alias for field number 6

start
Alias for field number 1

summit
Alias for field number 4

tags
Alias for field number 5

WrapperMACS.**iterateMacsPeaks**(*infile*)
iterate over peaks.xls file and return parsed data. The fdr is converted from percent to values between 0 and 1.

class WrapperMACS.**Macs2Peak**
Bases: tuple

Macs2Peak(contig, start, end, length, summit, pileup, pvalue, fold, fdr, name)

contig
Alias for field number 0

count(*value*) → integer – return number of occurrences of value

end
Alias for field number 2

fdr
Alias for field number 8

fold
Alias for field number 7

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

length
Alias for field number 3

name
Alias for field number 9

pileup
Alias for field number 5

pvalue
Alias for field number 6

start
Alias for field number 1

summit
Alias for field number 4

WrapperMACS.**iterateMacs2Peaks**(*infile*)
iterate over peaks.xls file and return parsed data. The fdr is converted from percent to values between 0 and 1.

WrapperZinba.py - utility functions for zinba output

Author Andreas Heger

Release Id

Date December 09, 2013

Tags Python

Purpose

Usage

Documentation

Code

class `WrapperZinba.`**`ZinbaPeak`**

Bases: `tuple`

ZinbaPeak(contig, unrefined_start, unrefined_end, strand, posterior, summit, height, refined_start, refined_end, median, fdr)

`contig`
Alias for field number 0

`count`(*value*) → integer – return number of occurrences of value

`fdr`
Alias for field number 10

`height`
Alias for field number 6

`index`(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

`median`
Alias for field number 9

`posterior`
Alias for field number 4

`refined_end`
Alias for field number 8

`refined_start`
Alias for field number 7

`strand`
Alias for field number 3

`summit`
Alias for field number 5

`unrefined_end`
Alias for field number 2

`unrefined_start`
Alias for field number 1

WrapperZinba.**iteratePeaks**(*infile*)
iterate of zinba peaks in infile.

CHAPTER 3

CGAT Pipelines

CGAT pipelines perform basic tasks, are fairly generic and might be of wider interest.

3.1 CGAT Pipelines

3.1.1 Installing CGAT pipelines

The CGAT pipelines, scripts and libraries make several assumptions about the computing environment. This section describes how to install the code and set up your computing environment.

Downloading and installing the source code

To obtain the latest code, check it out from the public mercurial repository at:

```
hg clone http://www.cgat.org/hg/cgat/ cgat
```

Once checked-out, you can get the latest changes via pulling and updating:

```
hg pull
hg update
```

Some scripts contain cython code that needs to be recompiled if the script or the pysam installation has changed. To rebuild all scripts, for example after updating the repository, type:

```
python cgat/scripts/cgat_rebuild_extensions.py
```

Recompilation requires a C compiler to be installed.

Setting up the computing environment

The pipelines assume that Sun Grid Engine has been installed. Other queueing systems might work, but expect to be disappointed. The pipeline is started on a *submit host* assuming a default queue `all.q`. Other queues can be specified on the command line, for example:

```
python cgat/CGATPipelines/pipeline_<name>.py --cluster-queue=medium_jobs.q
```

A pipeline might start up to `-p/--multiprocess` processes. Preferentially, tasks are sent to the cluster, but for some tasks this is not possible. These might thus run on the *submit host*, so make sure it is fairly powerful.

Pipelines expects that the *working directory* is accessible with the same path both from the submit and the *execution host*.

Software requirements

On top of pipeline specific bioinformatics software, CGAT pipelines make use a variety of software. Unfortunately we can't support many versions. The following table gives a list software we have currently installed:

Section	*Software*	*Version*
apps	java	jre1.6.0_26
apps	gccxml	0.9
apps	R	2.14.1
bio	alignlib	0.4.4
apps	python	2.7.1
apps	perl	5.12.3
apps	graphlib	0.1
bio	abiwtap	1.2.1
bio	bamstats	1.22
bio	batman	0.2.3
bio	bedtools	2.13.3
bio	belvu	2.16
bio	bfast	0.6.5a
bio	bioprospector	2004
bio	bowtie	0.12.7
bio	bwa	0.5.9
bio	cdhit	4.3
bio	clustalw	2.1
bio	cufflinks	1.3.0
bio	cpc	0.9-r2
bio	dialign	2.2.1
bio	ensembl	62
bio	ensembl-variation	62
bio	exonerate	2.2.0
bio	fastqc	0.9.2
bio	fastx	0.0.13
bio	gatk	1.0.5506
bio	gblocks	0.91b
bio	gcprofile	1.0
bio	gmap	2011.03.28
bio	galaxy	dist
bio	IGV	2.0.23
bio	IGVTools	1.5.12
bio	kent	1.0
bio	hmmer	3.0
bio	leotools	0.1
bio	meme	4.7.0
bio	muscle	3.8.31
bio	mappability_map	1.0

Continued on next page

Table 3.1 – continued from previous page

bio	ncbiblast	2.2.25+
bio	newickutils	1.3.0
bio	novoalign	2.07.11
bio	novoalignCS	1.01.11
bio	paml	4.4c
bio	picard-tools	1.48
bio	phylip	3.69
bio	polyphen	2.0.23
bio	samtools	0.1.18
bio	shrimp	2.1.1
bio	sicer	1.1
bio	sift	4.0.3
bio	simseq	72ce499
bio	soap	2.21
bio	soapsplice	1.0
bio	sratoolkit	2.1.7
bio	SpliceMap	3.3.5.2
bio	stampy	1.0.17
bio	statgen	0.1.4
bio	storm	0.1
bio	tabix	0.2.5
bio	tophat	1.4.1
bio	treebest	0.1
bio	tv	0.5
bio	vcftools	0.1.8a
bio	emboss	6.3.1
bio	velvet	1.1.04
bio	perm	0.3.5
bio	lastz	1.02.00
bio	hpeak	2.1
bio	boost	1.46.1
bio	Trinity	2012-01-25
bio	bowtie2	2.0.0-beta5
bio	tophat2	2.0.0
bio	all	1.0

What exactly is required will depend on the particular pipeline. The pipeline assumes that the executables are in the users `PATH` and that the rest of the environment has been set up for each tool.

Additionally, there is a list of additional software that is required that are usually shipped as a source package with the operating system. These are:

sqlite

Python libraries

CGAT uses python extensively and is currently developed against python 2.7.1. Python 2.6 should work as well, but some libraries present in 2.7.1 but missing in 2.6 might need to be installed. Scripts have not yet been ported to python 3.

CGAT requires the following in-house python libraries to be installed:

Library	*Version*	*Purpose*	*Download*
pysam	0.6.0	python bindings for samtools	hg clone https://code.google.com/p/pysam/ pysam
alignlib	0.4.5	C++ sequence alignment library with python bindings.	wget http://downloads.sourceforge.net/project/alignlib/alignlib/alignlib-0.4.5.tar.gz
sphinxreport	latest	report generator	svn checkout https://sphinx-report.googlecode.com/svn/trunk/ sphinx-report

In addition, CGAT scripts make extensive use of the following python libraries (list below might not be complete):

Library	*Version*	*Purpose*
numpy		
scipy		
rpy2		
matplotlib		
ruffus		
drmaa_python		

The full list of modules installed at CGAT is:

Module	Version	Method
pycairo	01/08/06	S
pygjobject	2.20.0	S
pygtk	2.16.0	S
wxPython	2.9.1.1	S
matplotlib	1	S
numpy	01/05/01	E
scipy	0.8.0	S
rpy	1.0.3	S
rpy2	02/02/00	S
networkx	1.3	E
pytables	2.2	
pygccxml	1	S
pyplusplus	1	S
bx.python		
pygresql	4	E
myqsl-python	01/02/03	E
biopython	1.56	E
ply	3.3	E
psyco		
pyrex	0.9.9	E
cython	0.13	E
sphinx	1.0.5	E
reportlab	2.5	E
guppy	0.1.9	E
pil	01/01/07	E
threadpool	01/02/07	E
progressbar	2.3	E
virtualenv	01/05/01	E
sqlalchemy	0.6.5	E
ruffus	2.2	E

Continued on next page

Table 3.2 – continued from previous page

drmaa	0.4b3	E
bx.python	12/01/10	S
corebio	0.5.0	E
weblogolib	3	E
mercurial	01/07/03	E
scikits.learn	0.7.1	E
web.py	0.34	E
pandas	0.5.0	E
pybedtools	0.6	E

Method : Installation method (E = easy_install/setuptools, S = setup.py/distutils, C = CGAT)

3.1.2 Using CGAT pipelines

This section provides a tutorial-like introduction to CGAT pipelines.

Introduction

A pipeline takes input data and performs a series of automated steps (*task*) on it to produce some output data.

Each pipeline is usually coupled with a *SphinxReport* document to summarize and visualize the results.

It really helps if you are familiar with following:

- the unix command line to run and debug the pipeline
- python in order to understand what happens in the pipeline
- ruffus in order to understand the pipeline code
- sge in order to monitor your jobs
- mercurial in order to up-to-date code

Setting up a pipeline

Before starting, check that your computing environment is appropriate (see *Installing CGAT pipelines*). Once all components are in place, setting up a pipeline involves the following steps:

Step 1: Get the latest clone of the cgat script repository:

```
hg clone http://www.cgat.org/hg/cgat/ src
```

Note: You need to have mercurial installed.

The directory `src` is the *source directory*. It will be abbreviated `<src>` in the following commands. This directory will contain the pipeline master script named `pipeline_<name>.py`, the default configuration files and all the helper scripts and libraries to run the pipeline.

Step 2: Create a *working directory* and enter it. For example:

```
mkdir version1
cd version1
```

The pipeline will live there and all subsequent steps should be executed from within this directory.

Step 3: Obtain and edit an initial configuration file. Ruffus pipelines are controlled by a configuration file. A configuration file with all the default values can be obtained by running:

```
python <src>/pipeline_<name>.py config
```

This will create a new `pipeline.ini` file. **YOU MUST EDIT THIS FILE**. The default values are likely to use the wrong genome or point to non-existing locations of indices and databases. The configuration file should be well documented and the format is simple. The documenation for the ConfigParser python module contains the full specification.

Step 4: Add the input files. The required input is specific for each pipeline; read the pipeline documentation to find out exactly which files are needed. Commonly, a pipeline works from input files copied or linked into the *working directory* and named following pipeline specific conventions.

Running a pipeline

Pipelines are controlled by a single python script called `pipeline_<name>.py` that lives in the *source directory*. Command line usage information is available by running:

```
python <src>/pipeline_<name>.py --help
```

The basic syntax for `pipeline_<name>.py` is:

```
python <src>/pipeline_<name>.py [options] _COMMAND_
```

`COMMAND` can be one of the following:

make <task> run all tasks required to build *task*

show <task> show tasks required to build *task* without executing them

plot <task> plot image (requires inkscape) of pipeline state for *task*

touch <task> touch files without running *task* or its pre-requisites. This sets the timestamps for files in *task* and its pre-requisites such that they will seem up-to-date to the pipeline.

config write a new configuration file `pipeline.ini` with default values. An existing configuration file will not be overwritten.

clone <srcdir> clone a pipeline from `srcdir` into the current directory. Cloning attempts to conserve disk space by linking.

In case you are running a long pipeline, make sure you start it appropriately, for example:

```
nice -19 nohup <src>/pipeline_<name>.py make full
```

This will keep the pipeline running if you close the terminal.

Troubleshooting

Many things can go wrong while running the pipeline. Look out for

- **bad input format. The pipeline does not perform sanity checks on the input format.** If the input is bad, you might see wrong or missing results or an error message.
- **pipeline disrutions. Problems with the cluster, the file system or the controlling terminal** might all cause the pipeline to abort.

- **bugs. The pipeline makes many implicit assumptions about the input files and the programs it** runs. If program versions change or inputs change, the pipeline might not be able to deal with it. The result will be wrong or missing results or an error message.

If the pipeline aborts, locate the step that caused the error by reading the logfiles and the error messages on stderr (`nohup.out`). See if you can understand the error and guess the likely problem (new program versions, badly formatted input, ...). If you are able to fix the error, remove the output files of the step in which the error occured and restart the pipeline. It should continue from the appropriate location.

Note: Look out for upstream errors. For example, the pipeline might build a geneset filtering by a certain set of contigs. If the contig names do not match, the geneset will be empty, but the geneset building step might conclude successfully. However, you might get an error in any of the downstream steps complaining that the gene set is empty. To fix this, fix the error and delete the files created by the geneset building step and not just the step that threw the error.

Updating to the latest code version

To get the latest bugfixes, go into the *source directory* and type:

```
hg pull
hg update
```

The first command retrieves the latest changes from the master repository and the second command updates your local version with these changes.

Building pipeline reports

Some of the pipelines are associated with an automated report generator to display summary information as a set of nicely formatted html pages. In order to build the documentation, drop the appropriate `conf.py` and `sphinxreport.ini` configuration files into the *working directory* and run the pipeline command:

    nice -19 pipeline_<name>.py make build_report

This will create the report from scratch in the current directory. The report can be viewed opening the file `<work>/report/html/contents.html` in your browser.

Sphinxreport is quite powerful, but also runs quite slowly on large projects that need to generate a multitude of plots and tables. In order to speed up this process, there are some advanced features that Sphinxreport offers:

- caching of results
- multiprocessing
- incremental builds
- separate build directory

Please see the sphinxreport documentation for more information.

3.1.3 Building CGAT pipelines

The best way to build a pipeline is to start from an example. There are several pipelines available, see *CGAT Pipelines*. To start a new project, use `pipeline_quickstart.py`:

```
python <srcdir>pipeline_quickstart.py --name=test
```

This will create a new directory called `test` in the current directory.

Another source of information is the script `pipeline_template.py` in the *source directory*.

This section describes how CGAT pipelines can be constructed using the `Pipeline` module. The Pipeline.py module contains a variety of useful functions for pipeline construction.

Overview

Pipelines generally have a similar structure. Pipelines are implemented as a pipeline script in the *source directory* called `pipeline_<somename>.py` and a file `pipeline_<somename>.ini` with default configuration values.

Pipeline input

Pipelines are executed within a dedicated *working directory*. They usually require the following files within this directory:

- a pipeline configuration file `pipeline.ini`
- input data files, usually linked in from a data repository

Other files that might be used in a pipeline are:

- external data files such as genomes that a referred to by they their full path name.
- sphinxreport.ini and conf.py for automated reports.

The pipelines will work from the input files in the *working directory*, usually identified by their suffix. For example, a ChIP-Seq pipeline might look for any `*.fastq.gz` files in the directory, run QC on these, map the reads to a genome sequence, call peaks, do motif analyses, etc.

Pipeline output

The pipeline will create files and database tables in the *working directory*. When building a pipeline, you can choose any file/directory layout that suits your needs. Some prefer flat hierarchies with many files, while others prefer deep directories.

Two directories have a special function and can be used for exporting pipeline results (see PipelinePublishing):

The `export` directory contains all files that will be referred to directly in the report or that later should be published by the pipeline. For example, pdf documents created by the peak caller or logo images created by a motif tool should go there.

The directory `report` will contain the automatically generated report.

Guidelines

To preserve disk space, please always work use compressed files as much as possible. Most data files compress very well, for example fastq files often compress by a factor of 80% or more: a 10Gb file will use just 2Gb.

Working with compressed files is straight-forward using unix pipes and the commands `gzip`, `gunzip` or `zcat`.

If you require random access to a file, load the file into the database and index it appropriately. Genomic interval files can be indexed with tabix to allow random access.

Running commands within tasks

To run a command line program within a pipeline task, build a statement and call the `Pipeline.run()` method:

```
@files( '*.unsorted', suffix('.unsorted'), '.sorted')
def sortFile( infile, outfile ):

    statement = '''sort %(infile)s > %(outfile)s'''
    P.run()
```

On calling the `Pipeline.run()` method, the environment of the caller is examined for a variable called `statement`. The variable is subjected to string substitution from other variables in the local namespace. In the example above, `%(infile)s` and `%(outfile)s` are substituted with the values of the variables `infile` and `outfile`, respectively.

The same mechanism also permits setting configuration parameters, for example:

```
@files( '*.unsorted', suffix('.unsorted'), '.sorted')
def sortFile( infile, outfile ):

    statement = '''sort -t %(tmpdir)s %(infile)s > %(outfile)s'''
    P.run()
```

will automatically substitute the configuration parameter `tmpdir` into the command. See ConfigurationValues for more on using configuration parameters.

The pipeline will stop and return an error if the command exits with an error code.

If you chain multiple commands, only the return value of the last command is used to check for an error. Thus, if an upstream command fails, it will go unnoticed. To detect these errors, insert the `checkpoint` statement between commands. For example:

```
@files( '*.unsorted.gz', suffix('.unsorted.gz'), '.sorted)
def sortFile( infile, outfile ):

    statement = '''gunzip %(infile)s %(infile)s.tmp;
                   checkpoint;
                   sort -t %(tmpdir)s %(infile)s.tmp > %(outfile)s;
                   checkpoint;
                   rm -f %(infile)s.tmp
    P.run()
```

Of course, the statement aboved could be executed more efficiently using pipes:

```
@files( '*.unsorted.gz', suffix('.unsorted.gz'), '.sorted.gz')
def sortFile( infile, outfile ):

    statement = '''gunzip < %(infile)s
                   | sort -t %(tmpdir)s
                   | gzip > %(outfile)s'''
    P.run()
```

The pipeline inserts code automatically to check for error return codes if multiple commands are combined in a pipe.

Running commands on the cluster

In order to run commands on cluster, use `to_cluster=True`.

To run the command from the previous section on the cluster:

```
@files( '*.unsorted.gz', suffix('.unsorted.gz'), '.sorted.gz')
def sortFile( infile, outfile ):

    to_cluster = True
    statement = '''gunzip < %(infile)s
                   | sort -t %(tmpdir)s
                   | gzip > %(outfile)s'''
    P.run()
```

The pipeline will automatically create the job submission files, submit the job to the cluster and wait for its return.

Pipelines will use the command line options `--cluster-queue`, `--cluster-priority`, etc. for global job control. For example, to change the priority when starting the pipeline, use:

```
python <pipeline_script.py> --cluster-priority=-20
```

To set job options specific to a task, you can define additional variables:

```
@files( '*.unsorted.gz', suffix('.unsorted.gz'), '.sorted.gz')
def sortFile( infile, outfile ):

    to_cluster = True
    job_queue = 'longjobs.q'
    job_priority = -10
    job_options= "-pe dedicated 4 -R y"

    statement = '''gunzip < %(infile)s
                   | sort -t %(tmpdir)s
                   | gzip > %(outfile)s'''
    P.run()
```

The above statement will be run in the queue `longjobs.q` at a priority of `-10`. Additionally, it will be executed in the parallel environment `dedicated` with at least 4 cores.

Array jobs can be controlled through the `job_array` variable:

```
@files( '*.in', suffix('.in'), '.out')
def myGridTask( infile, outfile ):

    job_array=(0, nsnps, stepsize)

    statement = '''grid_task.bash %(infile)s %(outfile)s
       > %(outfile)s.$SGE_TASK_ID 2> %(outfile)s.err.$SGE_TASK_ID
    '''
    P.run()
```

Note that the `grid_task.bash` file must be grid engine aware. This means it makes use of the `SGE_TASK_ID`, `SGE_TASK_FIRST`, `SGE_TASK_LAST` and `SGE_TASK_STEPSIZE` environment variables to select the chunk of data it wants to work on.

The job submission files are files called *tmp** in the *working directory*. These files will be deleted automatically. However, the files will remain after aborted runs to be cleaned up manually.

Tracks

A pipeline typically processes the data streams from several experimental data sources. These data streams are usually processed separately (processing, quality control) and as aggregates. The module `PipelineTracks` helps implementing this.

Databases

Loading data into the database

`Pipeline.py` offers various tools for working with databases. By default, it is configured to use an sqlite3 database in the *working directory* called `csvdb`.

Tab-separated output files can be loaded into a table using the `Pipeline.load()` function. For example:

```
@transform( 'data_*.tsv.gz', suffix('.tsv.gz'), '.load' )
def loadTables( infile, outfile ):
   P.load( infile, outfile )
```

The task above will load all tables ending with `tsv.gz` into the database Table names are given by the filenames, i.e, the data in `data_1.tsv.gz` will be loaded into the table `data_1`.

The load mechanism uses the script `csv2db.py` and can be configured using the configuration options `database` and `csv2db_options`. Additional options can be given via the optional *options* argument:

```
@transform( 'data_*.tsv.gz', suffix('.tsv.gz'), '.load' )
def loadTables( infile, outfile ):
   P.load( infile, outfile, "--index=gene_id" )
```

Connecting to a database

To use data in the database in your tasks, you need to first connect to the database. It helps to encapsulate the connection in a separate function. For example:

```
def connect():
    dbh = sqlite3.connect( PARAMS["database"] )
    statement = '''ATTACH DATABASE '%s' as annotations''' % (PARAMS["annotations_database"])
    cc = dbh.cursor()
    cc.execute( statement )
    cc.close()

    return dbh
```

The above function will connect to the database. It will also attach a secondary database `annotations`.

The following example illustrates how to use the connection:

```
@transform( ... )
def buildCodingTranscriptSet( infile, outfile ):

    dbh = connect()

    statement = '''SELECT DISTINCT transcript_id FROM transcript_info WHERE transcript_biotype = 'pro
    cc = dbh.cursor()
    transcript_ids = set( [x[0] for x in cc.execute(statement)] )
    ...
```

Reports

The `Pipeline.run_report()` method builds or updates reports using SphinxReport. Usually, a pipeline will simply contain the following:

```
@follows( mkdir( "report" ) )
def build_report():
    '''build report from scratch.'''

    E.info( "starting report build process from scratch" )
    P.run_report( clean = True )

@follows( mkdir( "report" ) )
def update_report():
    '''update report.'''

    E.info( "updating report" )
    P.run_report( clean = False )
```

This will add the two tasks `build_report` and `update_report` to the pipeline. The former completely rebuilds a report, while the latter only updates changed pages. The report will be in the directory `report`.

Note that report building requries two files in the *working directory*:

- `sphinxreport.ini` - configuration values for Sphinxreport.
- `conf.py` - configuration values for sphinx.

The section *Writing pipeline reports* contains more information.

Configuration values

Setting up configuration values

Pipelines are configured via a configuration script. The following snippet can be included at the beginning of a pipeline to set it all up:

```
# load options from the config file
import Pipeline as P
P.getParameters(
       ["%s.ini" % __file__[:-len(".py")],
       "../pipeline.ini",
       "pipeline.ini" ] )
PARAMS = P.PARAMS
```

Configuration parameters will be read first from the file named `pipeline_<pipeline_name>.ini` in the *source directory*. These sets all configuration values to default paramteres.

Next, the file `../pipeline.ini` will be read (if it exists) and configuration values that are specific to a certain project will overwrite default values.

Finally, run specific configuration will be read from the file `pipeline.ini` in the *working directory*.

The method `Pipeline.getParameters()` reads parameters and updates a global dictionary of parameter values. It automatically guesses the type of parameters in the order of `int()`, `float()` or `str()`.

If a configuration variable is empty (`var=`), it will be set to `None`.

Configuration values from another pipeline can be added in a separate namespace:

```
PARAMS_ANNOTATIONS = P.peekParameters( PARAMS["annotations_dir"],
                                       "pipeline_annotations.py" )
```

The statement above will load the parameters from a `pipeline_annotations` pipeline with *working directory* `annotations_dir`.

Using configuration values

Configuration values are accessible via the `PARAMS` variable. The `PARAMS` variable is a dictionary mapping configuration parameters to values. Keys are in the format `section_parameter`. For example, the key `bowtie_threads` will provide the configuration value of:

```
[bowtie]
threads=4
```

In a script, the value can be accessed via `PARAMS["bowtie_threads"]`.

Undefined configuration values will throw a `ValueError`. To test if a configuration variable exists, use:

```
if 'bowtie_threads' in PARAMS: pass
```

To test, if it is unset, use:

```
if 'bowie_threads' in PARAMS and not PARAMS['botwie_threads']: pass
```

Task specific parameters

Task specific parameters can be set by creating a task specific section in the `pipeline.ini`. The task is identified by the output filename. For example, given the following task:

```
@files( '*.fastq', suffix('.fastq'), '.bam')
def mapWithBowtie( infile, outfile ):
    ...
```

and the files `data1.fastq` and `data2.fastq` in the *working directory*, two output files `data.bam` and `data2.bam` will be created on executing `mapWithBowtie`. Both will use the same parameters. To set parameters specific to the execution of `data1.fastq`, add the following to `pipeline.ini`:

```
[data1.fastq]
bowtie_threads=16
```

This will set the configuration value `bowtie_threads` to 16 when using the command line substitution method in `Pipeline.run()`. To get an task-specific parameter values in a python task, use:

```
@files( '*.fastq', suffix('.fastq'), '.bam')
def mytask( infile, outfile ):
    MY_PARAMS = P.substituteParameters( locals() )
```

Thus, task specific are implemented generically using the `Pipeline.run()` mechanism, but pipeline authors need to explicitely code for track specific parameters.

Documentation

Up-to-date and accurate documentation is crucial for writing portable and maintainable pipelines. To document your pipelines write documentation as you would for a module. See `pipeline_template.py` and other pipelines for an example.

To rebuild all documentation, enter the `doc` directory in the *source directory* and type:

```
cd doc
python collect.py
```

This will collect all new scripts to the documentation.

Next, edit the file `contents.rst` and add your pipeline to the table of pipelines. Finally, type:

```
make html
```

to rebuild the documentation.

Using other pipelines

You can use the output of other pipelines within your own pipelines. `pipeline_annotations` is an example - it provides often used annotation data sets for an analysis. How to load another pipelines parameters, connect to its database and write a modular report have been discussed above.

If you write a pipeline that is likely to be used by others, it is best to provide an interface. For example, the `pipeline_annotations` pipeline has an interface section that list all the files that are produced by the pipeline. Other pipelines can refer to the interface section without having to be aware of the actual file names:

```
filename_cds = os.path.join( PARAMS["annotations_dir"],
                         PARAMS_ANNOTATIONS["interface_geneset_cds_gtf"] )
```

Running other pipelines within your pipeline *should* be possible as well - provided they are within their own separate *working directory*.

Publishing data

To publish data and a report, use the `Pipeline.publish_report()` method, such as in the following task:

```
@follows( update_report )
def publish_report():
    '''publish report.'''

    E.info( "publishing report" )
    P.publish_report()
```

On publishing a report, the report (in the directory `report`, specified by `report_dir`) will get copied to the directory specified in the configuration value `web_dir`. Also, all files in the `export` directory will get copied over and links pointing to such files will be automatically corrected.

The report will then be available at `http://www.cgat.org/downloads/%(project_id)s/report` where project_id is the unique identifier given to each project. It is looked up automatically, but the automatic look-up requires that the pipeline is executed within the `/ifs/proj` directory.

If the option *prefix* is given to publish_report, all output directories will be output prefixed by *prefix*. This is very useful if there is more than one report per project.

See `Pipeline.publish_report()` for more options.

Checking requisites

TODO

3.1.4 Writing pipeline reports

CGAT pipelines use SphinxReport to report the outcome of a pipeline run. Conceptually, the workflow is that a CGAT pipeline creates data and uploads it into a database. SphinxReport then creates a report from the database.

Background

Todo

Some text here about why sphinxreport

Advanced topics

Conditional content

The ifconfig extension allows to include content depending on configuration values. To use this extension you will need to modify `conf.py`. The example below shows the modifications implemented in *<no title>* to permit the conditional inclusion of sections of the report depending on the mapper chosen:

```
# add sphinx.ext.ifconfig to the list of extensions
extensions.append( 'sphinx.ext.ifconfig' )

# define a new configuration variable
##############################################################
# Add custom configuration variables for ifconfig extension
def setup(app):
    app.add_config_value('MAPPERS', '', True)

# Set the value of custom configuration variables
import CGAT.Pipeline as P
P.getParameters(
    ["%s/pipeline.ini" % os.path.splitext(__file__)[0],
         "../pipeline.ini",
         "pipeline.ini" ] )

MAPPERS = P.asList( P.PARAMS["mappers" ] )
```

The thus defined and set custom configuration value `MAPPERS` can now be used inside an rst document:

```
.. toctree::
   :maxdepth: 2

   pipeline/Methods.rst
   pipeline/Status.rst
   pipeline/Mapping.rst
   pipeline/MappingSummary.rst
   pipeline/MappingContext.rst
   pipeline/MappingAlignmentStatistics.rst
   pipeline/MappingComplexity.rst

.. ifconfig:: "tophat" in MAPPERS

   .. toctree::
      pipeline/MappingTophat.rst

.. ifconfig:: "star" in MAPPERS

   .. toctree::
      pipeline/MappingStar.rst

.. ifconfig:: "tophat" in MAPPERS or "star" in MAPPERS or "gsnap" in MAPPERS
```

```
.. toctree::
   pipeline/Validation.rst
```

Note that `.. ifconfig` needs to be a first level directive and can not be include into another directive such as `.. toctree`.

Referring to other reports

The intersphinx extension permits referring to other sphinxreport documents. To use this extension you will need to modify your `conf.py` configuration file. For example:

```
# add sphinx.ext.ifconfig to the list of extensions
extensions.append( 'sphinx.ext.intersphinx' )

# add mapping information
intersphinx_mapping = {
   'readqc': ('/ifs/projects/proj013/readqc/report/html', None) ,
   'mapping1': ('/ifs/projects/proj013/mapping1/report/html', None),
   'mapping2': ('/ifs/projects/proj013/mapping2/report/html', None),
    }
```

This will link to three other reports. The three reports are abbreviated as `readqc`, `mapping1` and `mapping2`. The paths need to be the absolute location of the html build of the sphinx documents you created previously. These directories should contain a `objects.inv` file which is usually automatically created by sphinx.

To refer to the other documentation, type:

```
:ref:`My link to another documentation <identifier:label>`
```

`label` is a valid identifier in the referred to document. For example:

```
:ref:`ReadQC <readqc:readqcpipeline>`

    ReadQC pipeline - fastqc

:ref:`Unique Mapping  <mapping1:mappingpipeline>`

    Mapping pipeline - short read mapping with bwa. Only
    uniquely mapping reads are kept.

:ref:`Non-unique mapping <mapping2:mappingpipeline>`

    Mapping pipeline - short read mapping with bwa with same
    parameters as above, but all reads are kept.
```

This section provides some background on CGAT pipelines.

3.1.5 Background

There really are two types of pipelines. In `production pipelines` the inputs are usually the same every time the pipeline is run and the output is known beforehand. For example, read mapping and quality control is a typical pipeline. These pipelines can be well optimized and can be re-used with little change in configuration.

`analysis pipelines` control scientific analyses and are much more in a state of flux. Here, the input might change over time as the analysis expands and the output will change with every new insight or new direction a project takes. It will be still a pipeline as long as the output can be generated from the input without manual intervention.

These pipelines leave less scope for optimization compared to `production pipelines` and adapting a pipeline to a new project will involve significant refactoring.

In CGAT, we are primarily concerned with `analysis pipelines`, though we have some `production pipelines` for common tasks.

There are several ways to build pipelines. For example, there are generic workflow systems like taverna which even provide GUIs for connecting tasks. A developer writes some glue code permitting the output of one application to be used as input for another application. Also, there are specialized workflow systems for genomics, for example galaxy, which allows you to save and share analyses. New tools can be added to the system and new data imported easily for example from the UCSC genome browser.

Flexibility There always new tools and insights. A pipeline should be ultimately flexible and not constraining us in the things we can do.

Scriptability The pipeline should be scriptable, i.e, the whole pipeline can be run within another pipeline. Similarly, parts of a pipeline can be duplicated to process several data streams in parallel. This is a crucial feature in genome studies as a single analysis will not permit making inferences by itself. For example, consider you find in ChIP-Seq data from a particular transcription factor that it binds frequently in introns. You will need to run the same analysis on data from other transcription factors in order to assess if intronic binding is remarkable.

Reproducibility The pipeline is fully automated. The same inputs and configuration will produce the same outputs.

Reusability The pipeline should be able to be re-used on similar data, maybe only requiring changes to a configuration file.

Archivability Once finished, the whole project should be able to archived without too many major dependencies on external data. This should be a simple process and hence all project data should be self-contained. It should not involve going through various directories or databases to figure out which files and tables belong to a project or a project depends on.

There probably is not one toolset to satisfy all these criteria.. We use the following tools to build a pipeline:

- ruffus to control the main computational steps
- sqlite to store the results of the computational steps
- sphinxreport to visualize the data in the sqlite database

3.1.6 NGS Pipelines

Genomics Pipelines

NGS Pipelines

Other pipelines

pipeline_quickstart.py - setup a new pipeline

Author

Release Id

Date December 09, 2013

Tags Python

Purpose

Usage Example:

```
python pipeline_quickstart.py --name=chipseq
```

Type:

```
python pipeline_quickstart.py --help
```

for command line help.

Documentation

Code

Obsolete pipelines

3.1.7 Lecgacy pipelines

Within of the Ponting group we have developed a few other pipelines that are based on the GNU `make` utility. These pipelines are listed below:

Transcript comparison pipeline

Purpose

Map 454 reads onto a genome and assemble overlapping transcripts into transcript models.

The pipeline currently does not use base quality information during mapping and does not consider alternative transcripts.

Setting up

To set up the pipeline in the current directory run:

```
python setup.py --method=compare_transcripts > setup.log
```

Link towards the genome from /net/cpp-data/backup/databases/indexed_fasta and call the files genome.fasta and genome.idx. For example:

```
ln -s /net/cpp-data/backup/databases/indexed_fasta/hs_ncbi36_softmasked.fasta genome.fasta
ln -s /net/cpp-data/backup/databases/indexed_fasta/hs_ncbi36_softmasked.idx genome.idx
```

Input (required):

%.gtf gtf files with (experimental) transcripts. The % denotes the track name, for example heart.gtf, kidney.gtf, sample1.gtf, ...

genome.fasta, genome.idx an indexed genome `PARAM_GENOME`. See also index_fasta.py.

ensembl.gtf a gtf file with a reference sequence set: the default is `ensembl`, but can be changed in `PARAM_MASTER_SET_GENES`.

annotations.gff a gff file with annotated genomic regions. See `PARAM_GENOME_REGIONS`. Use gtf2gff.py to create this file.

The pipeline includes additional information if it is present:

%.coverage table with coverage information for a track. The output is from blat2assembly.py

%.polyA information about polyA tails. The output is from blat2assembly.py

%.readstats a table with read alignment statistics after filtering (see output from MapTranscripts)

%.readmap a table mapping gene_ids to read_ids after filtering (see output from MapTranscripts)

%.readinfo a table with read information.

%.readgtf mapped locations of reads after filtering.

PARAM_FILE_REPEATS_RATES gff formatted file of ancesctral repeats. The score field contains the rate (see Makefile.ancestral_repeats)

PARAM_FILE_REPEATS gff file with repeats in genome. These are used for masking in coding potential predictions.

PARAM_FILE_REPEATS_GC gff formatted file of ancesctral repeats. The score field contains the G+C content (see Makefile.ancestral_repeats)

PARAM_FILE_ALIGNMENTS psl formatted file with genomic alignments between this species in query and another at appropriate evolutionary distance in target.

PARAM_FILENAME_GO (PARAM_FILENAME_GOSLIM) GO annotations for genes in the reference set. Example format is:

```
cell_location   ENSPPYG00000000676      GO:0016020      membrane        NA
```

PARAM_FILENAME_TERRITORIES gene territories. GTF formatted file, an example entry would be:

```
chr1    protein_coding  exon    3979975 4199559 .       -       .       transcript_id "ENSPPYG00
```

PARAM_CPC_UNIREF uniref database to use for coding potential predictions.

PARAM_DATABASE database name

Output from the mapTranscripts454 project can be imported with a single command:

```
make PATH_TO_MAPPING_DIR.add-tracks
```

Configuration

Edit the `Makefile` to configure the pipeline. See Parameters below.

Usage

The pipeline is controlled by running make targets. The results of the pipeline computation are stored as tab separated tables in the working directory. Most of these tables are then imported into an sqlite database called `csvdb` (see *PARAM_DATABASE*).

Annotation Type:

```
make all
```

to do all.

Fine grained control A more complete list of targets:

all make all

build only build, but do not import.

import import

Visualization The following targets aid visualizatiov:

ucsc-tracks-gtf export the segments as compressed gtf files. Can be viewed as user tracks in the ucsc genome browser.

GO analysis GO analysis will compute the relative enrichment/depletion gene sets.

Requires `PARAM_FILENAME_TERRITORIES`, `PARAM_FILENAME_GO` and `PARAM_FILENAME_GOSLIM` to be set.

There are two counting methods. The first method (`go`) assigns GO terms associated with the reference gene set to TLs and counts these. The second method (`territorygo`) assigns TLs to genes in the reference set and then does a GO analysis on theses.

Note: The convential GO analysis based on gene list is the `territorygo` method.

Usage Usage:

```
make <track>:<slice>:<subset>:<background>.<go>.<method>analysis
```

The fields are:

track the data track to be chosen.

slice the slices correspond to flags in the table <track>_annotation. Use `all` to use all segments in a `track`.

subset the subset corresponds to a table that is joined with <track>_annotation to restrict segments to a user-specified set. Use `all` for no restriction.

background the background gene set

go either `go` or `goslim`

method either `go` or `goterritory`

Results will be in the directory `<track>:<slice>:<subset>:<background>.<go>.<method>analysis.dir`.

For example:

```
make thoracic:known:all:thoracic.go.goanalysis
```

will compute the enrichment of protein coding TL in the track `thoracic` using all `thoracic` genes as the background.

The command:

```
make thoracic:known:all:ensembl.goslim.territorygoanalysis
```

will compute `goslim` term enrichment. The foreground set are genes from the reference set (`ensembl`) overlapping protein coding TL in the track `thoracic`. The background is the complete reference gene set (`ensembl`).

Annotator analysis Annotator computes the statistical significance of enrichment/depletion of genomic features (called segments) within genomic regions (called annotations).

To run annotator analysis, two files need to be present:

1. A workspace
2. A collection of annotations on the genome

Building workspaces

Workspaces are built using makefile targets. For example to build :file:`genome.workspace`, type:: make genome.workspace

All workspaces exclude contigs called matching `random`.

genome.workspace full genome

intergenic.workspace only intergenic regions

intronic.workspace only intronic regions

unknown.workspace both intergenic and intronic regions

territories.workspace workspace of territories

alignable.workspace only segments that can be aligned to a reference genome.

There is a convenience target:

```
make annotator-workspaces
```

that will build all available workspaces.

Annotations Annotations are built using makefile targets.

all.annotations: all subsets (all/known/unknown) for each track.

architecture.annotations: annotations according to genes (intronic, intergenic, ...).

{all,known,unknown}_sets.annotations annotations of known, unknown, all transcripts

allgo_territories.annotations territories annotation with GO categories

allgoslim_territories.annotations territories annotation with GOSlim categories

intronicgo_territories.annotations territories annotation with GO categories

intronicgoslim_territories.annotations territories annotation with GOSlim categories

intergenicgo_territories.annotations territories annotation with GO categories

intergenicgoslim_territories.annotations territories annotation with GOSlim categories

There is a convenience target:

```
make annotator-annotations
```

that will build all available annotations.

Usage In order to perform `Annotator` analyses, you run a make target:

```
make <track>:<slice>:<subset>:<workspace>:<workspace2>_<annotations>.annotators
```

The fields determine which segments are used for the enrichment analysis.

track the data track to be chosen.

slice the slices correspond to flags in the table <track>_annotation. Use `all` to use all segments in a `track`.

subset the subset corresponds to a table that is joined with <track>_annotation to restrict segments to a user-specified set. Use `all` for no restriction.

workspace the workspace to be used

workspace2 a second workspace. The actual workspace will be the intersection of both workspaces.

annotations annotations to use.

Note: Annotations, segments and the workspace need to be chosen carefully for each experiment. For example, failing to use territories for goterritory analysis will measure enrichment of segments within goterritories in general, and not necessarily relative enrichment between go territories.

The results will be in the file `<track>:<slice>:<subset>:<workspace>:<workspace2>_<annotations>.annotato`

Examples The command:

```
make thoracic:unknown:all:intergenic:all_unknownsets.annotators
```

will test for enrichment among `unknown` transcripts in the track `thoracic` with intergenic segments the other sets. The command:

```
make thoracic:intronic:all:intronic:territories_intronicgoslimterritories.annotators
```

will check for enrichment of `intronic` transcripts from the track `merged` within intronic genomic segments that also have GO assignments (intersection of workspaces `intronic` and `territories`. It will label GO territories by GOslim territories.

Association analysis Association analysis computes the significance of finding segments close to annotations.

Type:

```
make annotator-distance-run
```

to run all association analyses.

Parameters

The following parameters can be set in the `Makefile`:

GPipe - Gene prediction pipeline

Introduction

This document describes the pipeline of the Chris Ponting group for predicting genes by homology. The input is a set of known transcripts from a reference genome and the masked genomic sequence of a target genome.

The pipeline predicts genes in a two-step procedure:

1. Regions of similarity between the known transcripts and the reference genome are identified using a quick heuristic search.
2. The regions of similarity of step 1 are submitted to a sensitive, but slower gene prediction program.

The pipeline contains many options to mask sequences, analyse and quality control the predictions and store the results in a relational database. This manual describes how to setup up the pipeline, run it on our cluster, and analyse the results.

Note: The pipeline is tailored to the computational setup within the Ponting group. Porting it elsewhere will require a significant amount of software installation and configuration.

Setting up

The pipeline consists of a set of scripts and a makefile, that glues together the various scripts. This section tells you what programs are needed to be installed (Requirements), how to install the software from the pipeline (Installation).

Next, the input files need to prepared (Input files).

Before starting the pipeline, you need to configure your environment and the pipeline (Configuration).

Requirements Gpipe requires following software to be installed:

postgres Gpipe currently requires a postgres installation.

seq Seg is a program to mask low complexity regions in protein sequences.

exonerate Exonerate is a program to align a peptide sequence to a genomic sequence (other alignment modes are possible). It offers heuristic modes, that allow for fast scanning of large chunks of genomic DNA, and exhaustive modes, that do a full dynamic programming mode. It is available at http://www.ebi.ac.uk/~guy/exonerate.

python Most scripts require python...

perl ...unless they are written in perl

alignlib a library for sequence alignments and its python interface. See http://sourceforge.net/projects/alignlib.

SGE Sun Grid Engine. Other schedulers might work.

seq_pairs_kaks Leo's wrapper around PAML (optional, only required for step12).

Installation Untar and unpack the source code tarball. The directory in which it ends up is the *source directory*.

Gpipe works in a *working directory*. To set up the pipeline with the current directory as the *working directory*, run:

```
python <src>setup.py --method=gpipe --project=project_name > setup.log
```

`src` is the location of the *source directory*.

This will create a makefile in the current directory and give the project the name `project_name`. The latter will be used as the name of the database schema in postgres in which data will be stored.

Note that gpipe assumes that you use `bash` as you shell. In particular, it requires that two functions are in your environment:

```
#-----------------------------------
# helper functions for detecting errors in pipes
#-----------------------------------
detect_pipe_error_helper()
{
 while [ "$#" != 0 ] ; do
     # there was an error in at least one program of the pipe
     if [ "$1" != 0 ] ; then return 1 ; fi
     shift 1
 done
 return 0
}

detect_pipe_error()
{
 detect_pipe_error_helper "${PIPESTATUS[@]}"
 return $?
}
```

Once the code is in place, add the input files to the working and make sure that all the other requirements are fulfilled.

Input files Gpipe requires 5 files to run. These input files contain the reference gene set to predict with and the genome sequence to predict in. Sample data is available at http://genserv.anat.ox.ac.uk/downloads/software/gpipe/sampledata/gpipe_sample_data.tar. To drop the sample data into your *working directory*, type:

```
wget ttp://genserv.anat.ox.ac.uk/downloads/software/gpipe/sampledata/gpipe_sample_data.tar
tar -xf gpipe_sample_data.tar
gunzip *
```

The filenames and their contents are:

`peptides.fasta` A fasta-formatted file with peptide sequences. Each sequence is on a single line. The identifier of a sequence is taken from the description line with the pattern `>(\S+)` (characters between > and first white-space). For example:

```
>CG11023-RA
MGERDQPQSSERISIFNPPVYTQHQVRNEAPYIPTTFDLLSDDEESSQRVANAGPSFRPL...
>CG2671-RA
MLKFIRGKGQQPSADRHRLQKDLFAYRKTAQHGFPHKPSALAYDPVLKLMAIGTQTGALK...
...
```

`genome.fasta` and `genome.idx` A fasta-formatted file with the genomic sequence together with its index. This file can be created from a collection of individual fasta formatted files (for example, per chromosome files) using the command:

```
python <src>index_fasta.py genome <somedir>my_dna_sequences*.fa.gz > genome.log
```

`reference.exons` A table with gene models from the reference gene set. This is a tab-formatted table with the following columns:

transcript name name of the transcript consistent with `peptides.fasta`

contig name name of the DNA segment the transcript is located on

strand strand

phase phase of that particular exon

exon-id numerical number of exon

peptide_start start of exon in transcript sequence

peptide_end end of exon in transcript sequence

genome_start start of exon on contig

genome_end end of exon on contig

Coordinates are 0-based, half-open intervals. Genomic coordinates are forward/reverse strand coordinates.

For example:

```
CG10000-RA chr3R   -1      0       1       0       126     24577165        24577291
CG10000-RA chr3R   -1      0       2       126     287     24576946        24577107
CG10000-RA chr3R   -1      1       3       287     466     24576706        24576885
CG10000-RA chr3R   -1      2       4       466     930     24576187        24576651
CG10000-RA chr3R   -1      0       5       930     1100    24575892        24576062
CG10000-RA chr3R   -1      1       6       1100    1280    24575573        24575753
CG10000-RA chr3R   -1      1       7       1280    1677    24574936        24575330
CG10001-RA chr3R   -1      0       1       0       540     24569207        24569747
CG10001-RA chr3R   -1      0       2       540     819     24566427        24566706
CG10001-RA chr3R   -1      0       3       819     978     24566193        24566352
```

map_rep2mem A table linking genes to transcripts. This tab-formatted table contains the following columns

rep A gene identifier

mem A transcript identifier

size Transcript size

Configuration To configure the pipeline, options can be set in the `Makefile` in the *working directory*.

Options that might need to be changed:

PARAM_PSQL_DATABASE The psql database

PARAM_PSQL_HOST The psql host

PARAM_PSQL_USER The psql username

Running the pipeline

The pipeline uses makefiles to control script logic. Before executing any make commands, run:

```
source setup.csh
```

to update your paths and other environment variables.

Before first running the pipeline, some maintenance work need to be performed like creating the database schema and the tables. To prepare the pipeline, run:

```
make prepare
```

This needs to be done only once.

To run the pipeline, type:

```
make all
```

Gpipe writes status messages to the file `log` in the *working directory*.

Results

Results of the gpipe run are stored in the psql database.

The view `overview` aggregates most results into a single table for easy access.

Troubleshooting

If something goes wrong, the first step is to look at the command line that caused the problem. To see the command executed, run:

```
make -n <target>
```

We use Sun Grid Engine as job queueing system and assume that for all nodes the code and data can be reached via the same mount point. All jobs that are run on the cluster are prefixed by the MAKE variable $(CMD_REMOTE_SUBMIT). You can set this variable to the empty string to run everything locally or on a mosix cluster:

make all CMD_REMOTE_SUBMIT=

Steps

The pipeline proceeds in 12 steps, which are:

Step1 Masking of protein sequences

Step2 Selecting representative transcipts to search with

Step3 Running exonerate

Step4 Running TBLASTN (disabled)

Step5 Collating putative gene-containing regions

Step6 Predicting genes for representative transcripts.

Step7 Predicting genes for redundant (alternative) transcripts

Step8 Predicting genes for member sequences

Step9 Analysing the predictions

Step10 Quality control of predictions

Step11 Removing redundant/erroneous predictions

Step12 Filter by ks (optional)

Glossary

working directory The working directory. Location of the data files and results. All commands in this tutorial are executed in the working directory.

source directory The location of the source code. The place where the script `setup.py` resides.

454 Transcript mapping pipeline

Purpose

Map 454 reads onto a genome and assemble overlapping transcripts into transcript models.

The pipeline currently does not use base quality information during mapping and does not consider alternative transcripts.

Setting up

To set up the pipeline in the current directory run:

```
python setup.py --method=map_transcripts_454 > setup.log
```

Add or link fasta files of reads into directory. These should end with the suffix .fasta. The pipeline will process several files at the same time. For example:

```
tissue1.fasta
tissue2.fasta
tissue3.fasta
```

Link towards the genome from /net/cpp-data/backup/databases/indexed_fasta and call the files genome.fasta and genome.idx. For example:

```
ln -s /net/cpp-data/backup/databases/indexed_fasta/hs_ncbi36_softmasked.fasta genome.fasta
ln -s /net/cpp-data/backup/databases/indexed_fasta/hs_ncbi36_softmasked.idx genome.idx
```

Build the index for gmap by running gmap_setup. By default, gmap indices should be put in `/net/cpp-mirror/databases/gmap`. Provide the location to the indices using the variable `PARAM_GMAP_OPTIONS`.

Note: Indices on networked disks are slow to load up. For performance reasons work with local indices.

Configuration

Edit the `Makefile` to configure the pipeline. See Parameters below.

Parameters

The following parameters can be set in the `Makefile`:

OPTIC

Purpose

The optic pipeline performs orthology assignment in a group of species.

Setting up

Download CGAT code CGAT code can be obtained by checking the latest version out from mercurial:

hg clone http://www.cgat.org/hg/cgat

In the following, the location of the checked out code will be referred to as <src>.

Requirements Optic requires the following tools to be installed.

Program	*Version*	*Purpose*
postgres		database
muscle	>=3.81.3	Multiple alignment
phyop		read mapping
treebest	>=0.1	bam/sam files
paml	>=4.4c	evolutionary rate estimation

Genomes The first step is to build the files with the genomic sequences. Download fasta files with genomic sequences and index them using the <src>/IndexedFasta.py tools.

Store the genomes in a separate directory (referred to later as <genome>).

Genesets The second step is to upload the genesets into the database. The pipeline expects protein coding transcripts from ENSEMBL and requires two files that can be downloaded from the ENSEMBL ftp site:

- File with peptide sequences, usually called `species`.pep.all.fa.gz
- **File with exon coordinates in gtf format, usually called** `species`.gtf.gz

The following example shows how to upload the human gene set. We are going to be using ensembl version 62 on hg19:

```
mkdir -p genesets/hs62

cd genesets/hs62

ln -s <genomes>hg19.fasta genome.fasta

ln -s <genomes>hg19.idx genome.idx

ln -s <mirror>Homo_sapiens.GRCh37.62.gtf.gz reference.gtf.gz

ln -s <mirror>Homo_sapiens.GRCh37.62.pep.all.fa.gz reference.pep.fa.gz
```

Create the makefile:

```
python <src>setup.py -m ensembl -p cgat_hs62
```

Create database tables:

```
make prepare
```

Upload data:

```
make all
```

To verify all was ok, look at the file `predictions.check`. This file compares the ENSEMBL supplied peptide sequences with those that have been uploaded into the database.

The data is now stored in the database schema cgat_hs62.

You need to do this for all species that you want to run OPTIC on.

Make sure that the naming is consistent with the genomes. Thus, `hg19` should both refer to the gene set for human, but also to the genomic sequence files (`hg19.fasta`) in the <genomes> directory.

Running OPTIC

The optic pipeline works from several directories.

Create a working directory:

```
mkdir optic
```

Create a makefile:

```
python <src>setup.py -m optic -p optic
cd optic
```

Enter the `data` directory and create the following files:

Makefile.inc common makefile file options. For example:

```
## Global configuration options for OPTIC
PARAM_PROJECT_NAME=cgat_proj007

DIR_TMP=/tmp/

PARAM_DIR_DATA=<optic>data/

PARAM_SRC_SCHEMAS=cgat_hs62 cgat_mm62 cgat_gg62 cgat_ac62 cgat_xt62
cgat_dr62 cgat_oa65

PARAM_SPECIES_TREE=((((((cgat_hs62,cgat_mm62),cgat_oa65),cgat_gg62),cgat_ac62),cgat_xt62),cgat_d

PARAM_ANALYSIS_DUPLICATIONS_OUTGROUPS=cgat_dr62

## CGAT cluster params

DIR_SCRIPTS=<src>/

PARAM_QUEUE=all.q
PARAM_QUEUE_LOCAL=all.q
PARAM_QUEUE_SERVER=all.q
```

schema2sp tsv file mapping species in database schema to swissprot name (required for treebest):

```
# map of species names to swiss prot taxonomic names
# used for njtree
schema  sp
cgat_hs62       HUMAN
cgat_mm62       MOUSE
cgat_xt62       XENTR
cgat_dr62       DANRE
cgat_ac62       ANOCA
cgat_gg62       CHICK
cgat_oa65       ORNAN
```

species_tree the phylogeny of the species in newick format, for example:

```
((((((cgat_hs62,cgat_mm62),cgat_oa65),cgat_gg62),cgat_ac62),cgat_xt62),cgat_dr62);
```

species_tree_permutations the phylogeny of the species and permutations of it. Usually just a copy of species_tree files needed for web server:

schema2colour map species name to colour:

```
cgat_hs62       255,204,0
cgat_dr62       255,102,204
cgat_xt62       204,204,0
cgat_mm62       204,102,255
cgat_ac62       102,255,255
cgat_gg62       125,125,255
cgat_oa65       255,255,102
```

schema2name map species name to real name:

```
schema  name
cgat_hs62       H. sapiens
cgat_mm62       M. musculus
cgat_ac62       A. carolinensis
cgat_xt62       X. tropicalis
cgat_dr62       D. rerio
cgat_gg62       G. gallus
cgat_oa65       O. anatinus
```

schema2url map species name to ENSEMBL URL:

```
cgat_hs62       displayGene?schema=%(species)s&gene_id=%(gene)s
cgat_mm62       displayGene?schema=%(species)s&gene_id=%(gene)s
cgat_ac62       displayGene?schema=%(species)s&gene_id=%(gene)s
cgat_xt62       displayGene?schema=%(species)s&gene_id=%(gene)s
cgat_dr62       displayGene?schema=%(species)s&gene_id=%(gene)s
cgat_gg62       displayGene?schema=%(species)s&gene_id=%(gene)s
cgat_oa65       displayGene?schema=%(species)s&gene_id=%(gene)s
```

Preparing data Create export data files. This will create fasta file and exon boundary files for all species:

```
make -C export export_clustering
```

Phyop Setup pairwise phyop runs and run them:

```
make -C orthology_pairwise prepare
```

```
nice -19 nohup make -C orthology_pairwise all
```

Wait a while...

Clustering The clustering step combines pairwise orthology assignments into clusters of potential orthologs:

```
make -C orthology_multiple prepare
make -C orthology_mulitple all
```

Multiple alignment Next, multiple alignments are built for each cluster:

```
make -C malis prepare
make -C malis all
make -C malis summary.dir
make -C malis summary
```

Orthologous groups Based on the multiple alignments, trees are built within each cluster and the trees are split into orthologous groups:

```
make -C paralogy_trees prepare
make -C paralogy_trees all
make -C paralogy_trees analysis
make -C paralogy_trees summary
```

Configuration

Edit the `Makefile` to configure the pipeline.

Some of these pipelines are still being used, though they are not actively supported any more.

CHAPTER 4

Developer's guide

4.1 Contributing to CGAT code

We encourage everyone who uses parts of the CGAT code collection to contribute. Contributions can take many forms: bugreports, bugfixes, new scripts and pipelines, documentation, tests, etc. All contributions are welcome.

4.1.1 Checklist for new scripts/modules

Before adding a new scripts to the repository, please check if the following are true:

1. The script performs a non-trivial task. If a one-line command line entry using standard unix commands can give the same effect, avoid adding a script to the repository.
2. The script has a clear purpose. Scripts should follow the unix philosophy. They should concentrate on one task and do it well. Ideally, the major input and output can be read from and written to standard input and standard output, respectively.
3. The script follows the naming convention of CGAT scripts.
4. The scripts follows the *Style Guide*.
5. The script implements the `-h/--help` options. Ideally, the script has been derived from `scripts/cgat_script_template.py`.
6. The script can be imported. Ideally, it imports without performing any actions or writing output.
7. The script is well documented and the documentation has been added to the CGAT documentation. There should be an entry in `doc/scripts.rst` and a file `doc/scripts/newscript.py`.
8. The script has at least one test case added to `tests` - and the test works (see *Testing*).

4.1.2 Building extensions

Using pyximport, it is (relatively) straight-forward to add optimized C-code to python scripts and, for example, access pysam internals and the underlying samtools library. See for example *<no title>*.

To add an extension, the following needs to be in place:

1. The main script (`scripts/bam2stats.py`). The important lines in this script are:

```
try:
    import pyximport
    pyximport.install()
    import _bam2stats
except ImportError:
    import CGAT._bam2stats as _bam2stats
```

The snippet first attempts to build and import the extension by setting up pyximport and then importing the cython module as `_bam2stats`. In case this fails, as is the case for an installed code, it looks for a pre-built extension (by `setup.py`) in the CGAT pacakge.

2. The cython implementation `_bam2stats.pyx`. This script imports the pysam API via:

```
from csamtools cimport *
```

This statement imports, amongst others, `AlignedRead` into the namespace. Speed can be gained from declaring variables. For example, to efficiently iterate over a file, an `AlignedRead` object is declared:

```
# loop over samfile
cdef AlignedRead read
for read in samfile:
    ...
```

3. A `pyxbld` providing pyximport with build information. Required are the locations of the samtools and pysam header libraries of a source installation of pysam plus the `csamtools.so` shared library. For example:

```
def make_ext(modname, pyxfilename):
    from distutils.extension import Extension
    import pysam, os
    dirname = os.path.dirname( pysam.__file__ )[:-len("pysam")]
    return Extension(name = modname,
                     sources=[pyxfilename],
                     extra_link_args=[ os.path.join( dirname,
                            "csamtools.so")],
                     include_dirs =  pysam.get_include(),
                     define_macros = pysam.get_defines() )
```

If the script `bam2stats.py` is called the first time, pyximport will compile the cython extension `_bam2stats.pyx` and make it available to the script. Compilation requires a working compiler and cython installation. Each time `_bam2stats.pyx` is modified, a new compilation will take place.

pyximport comes with cython.

4.2 Testing

This module describes the implementation of unit tests for the CGAT code collection.

4.2.1 Testing scripts

Scripts are tested by comparing the expected output with the latest output. The tests are implemented in the script `test_scripts.py`.

This script collects tests from subdirectories in the `tests` directory. Each test is named by the name of the script it tests.

Adding a new test manually

To add a new test for a CGAT script, create a new *test directory* in the directory `tests`. The name of the *test directory* has to correspond to the name of the script the tests will tested.

In this directory, create a file called `tests.yaml`. This file is in *yaml* format, a simple text-based format to describe nested data structures.

The `tests.yaml` file contains the descriptions of the individual tests to run. Each test is a separate data structure in this file. The fields are:

options Command line options for running the test. If you need to provide additional files as input, use the `%DIR%` place holder for the *test directory*.

stdin Filename of file to use as stdin to the script. If no stdin is required, set to `null` or omit.

outputs A list of output files obtained by running the script that should be compared to the list of files in `references`. `stdout` signifies the standard output.

references A list of expected output files. The order of `outputs` and `references` should be the same. The reference files are expected to be found in the directory *test directory* and thus need not prefixed with a directory place holder.

description A description of test.

To illustrate, we will be creating tests for the scripts `fasta2counts.py`. First we create the *test directory* `tests/fasta2counts.py`. Next we create a file `tests/fasta2counts.py/tests.yaml` with the following content:

```
basic_test:
    outputs: [stdout]
    stdin: null
    references: [test1.tsv]
    options: --genome-file=<DIR>/small_genome
```

`basic_test` is the name of the test. There is no standard input and the output of the script goes to stdout. Stdout will be compared to the file `test1.tsv`. The script requires the `--genome-file` option, which we supply in the `options` field. The `<DIR>` prefix will be expanded to the directory that contains the file `tests.yaml`.

Finally, we create the required input and reference files in the *test directory*. Our directory structure looks thus:

```
|___tests
  |___fasta2counts.py
  | |___small_genome.fasta
  | |___small_genome.idx
  | |___test1.tsv
  | |___tests.yaml
```

Multiple tests per script can be defined by adding additional data structures in the `tests.yaml` file.

Please write abundant tests, but keep test data to a minimum. Thus, instead of running on a large bam file, create stripped down versions containing only relevant data that is sufficient for the test at hand.

Re-use test data as much as possible. Some generic test data used by multiple tests is in the `tests/data` directory.

Creating a test

The script `tests/setup_test.py` can be used to set up a testing stub. For example:

```
python tests/setup_test.py scripts/bam2bam.py
```

will add a new test for the script `bam2bam.py`.

The script will create a new testing directory for each script passed on the command line and create a simple `tests.yaml` file. The basic test will simply call a script to check if starts without error and returns a version string.

Running tests

In order to run the tests on CGAT scripts, type:

```
nosetests tests/test_scripts.py
```

In order to get more information, type:

```
nosetests -v tests/test_scripts.py
```

To run individual tests, edit the file `tests/test_scripts.yaml`. In order to restrict testing to a single script, for example `beds2counts.py`, add the following:

```
restrict:
      regex: beds2counts.py
```

4.2.2 Testing modules

TODO e

4.2.3 Testing pipelines

TODO - describe pipeline_testing

4.3 Style Guide

4.3.1 Coding style

This style guide lays down coding conventions in the CGAT repository. For new scripts, follow the guidelines below.

As the repository has grown over years and several people contributed, the style between scripts can vary. For older scripts, follow the style within a script/module. If you want to apply the newer style, make consistent changes across the script.

In general, we want to adhere to the following conventions:

- *Variable names* are lower case throughout with underscores to separate words, such as `peaks_in_interval = 0`
- ***Function names* start with a lower case character and a** verb. Additional words start in upper case, such as `doSomethingWithData()`
- ***Class names* start with an upper case character, additional words** start again in upper case, such as `class AFancyClass():`
- ***Class methods* follow the same convention as functions, such as** `self.calculateFactor()`
- ***Class attributes* follow the same convention as variables, such** as `self.factor`

- ***Global variables* - in the rare cases they are used, are upper case** throughout such as `DEBUG=False`
- ***Module names* should start with an uppercase letter, for example,** `TreeTools.py` in order to distinguish them from built-in and third-party python modules.
- *Script names* are lower-case throughout with underscores to separate words, for example, `bam2geneprofile.py` or `join_table.py`.
- *Cython extensions* to scripts (via pyximport) should be put into the script name starting with an underscore. For example, The extensions to `bam2geneprofile.py` are in `_bam2geneprofile.pyx`.

For new scripts, use the template `script_template.py`.

The general rule is to write easily readable and maintainable code. Thus, please

- document code liberally and accurately
- **make use of whitespaces and line-breaks to break long statements** into easily readable statements.

In case of uncertainty, follow the python style guides as much as possible. The relevant documents are:

- PEP0008 - Style Guide for Python Code
- PEP0257 - Docstring Conventions

For documenting CGAT code, we follow the conventions for documenting python code:

- Python Developer's guide

In terms of writing scripts, we follow the following conventions:

- Each script should define the `-h` and `--help` options to give command line help usage.
- For tabular output, scripts should output *tsv* formatted tables. In these tables, records are separated by new-line characters and fields by tab characters. Lines with comments are started by the # character and are ignored. The first uncommented line should contain the column headers. For example:

```
# This is a comment
gene_id length
gene1   1000
gene2   2000
# Another comment
```

- Scripts should follow the unix philosophy. They should concentrate on one task and do it well. Ideally, the major input and output can be read from and written to standard input and standard output, respectively.
- The names of scripts should be meaningful. Most of our scripts perform data transformation of one kind of another, these are often called `a2b.py`. The distinctions can be subtle. Examples are:

  <no title> Input is *gtf*, output is *gtf*. This script manipulates gene sets (filtering, merging, ...).

  <no title> Input is *gtf*, output is *gff*. This script takes gene sets and changes the hierarchical description within a *gtf* file to the flat description of features in a *gff* file. For example, this script can define gene territories, regulatory domains or genomic annotations based on a gene set.

  <no title> Input is *bed*, output is *gff*. As both formats describe intervals in the genome, this script basically does a conversion between the two formats.

  Quite a few scripts contain the `2table` or `2stats`. These compute, respectively, properties or summary statistics for entries in a file. For example:

  <no title> Input is *gtf*. For each gene or transcript, compute selected properties. If there are 10,000 genes in the input, the output table will contain 10,000 rows.

<no title> Input is *gff*. Compute summary statistics across all features in the file. Here, aggregate sizes or similar by feature type or name per chromosome. No matter if there are 10,000 or 100,000 interval is the input, the output will be have the same number of rows.

4.3.2 Where to put code

Different parts of the code base go into separate directories.

Scripts Scripts are python code that contains a main() function and are intended to be executed. Scripts go into the directory `/scripts`

Modules Modules contain supporting code and are imported by scripts or other modules. Modules go into the directory `/CGAT`.

Pipelines Pipeline scripts and modules go into the directory `/CGATPipelines`.

4.3.3 Pipelines

All components of a pipeline should go into the `CGATPipelines` directory. The basic layout of a pipeline is:

```
CGATPipelines/pipeline_example.py
             /PipelineExample.py
             /PipelineExample.R
             /pipeline_example/pipeline.ini
                              /conf.py
                              /sphinxreport.ini
```

pipeline_example.py The main pipeline code. Pipelines start with the word `pipeline` and follow the conventions for *script names*, all lower case with underscores separating words.

pipeline_example/pipeline.ini Default values for pipeline configuration values.

pipeline_example/conf.py Configuration script for sphinxreport.

pipeline_example/sphinxreport.ini Configuration script for sphinxreport.

pipeline_docs/pipeline_example Sphinxreport for pipeline.

PipelineExample.py Python utility methods and classes specific to this pipeline. Once methods and classes are shared between pipelines, consider moving them to a separate module.

PipelineExample.R R utility functions specific to this pipeline.

- Make sure that the pipeline.ini file exists and contains example/default values with annotation.
- Make sure that the pipeline can be imported from any directory, especially those not containing any data files or configuration files. This is important for the documentation of the pipeline to be built.

4.3.4 Other guidelines

- Only add source code and required data to the repository. Do not add .pyc files, backup files created by your editor or other files.
- In order to build documentation, each script, module and pipeline needs to be importable. Thus, make sure that when your pipeline depends on specific files, it does not fail when imported but not executed.

4.3.5 Documentation

Writing doc-strings

Functions should be documented through their doc-string using restructured text. For example:

```
def computeValue( name, method, accuracy=2):

    :param name: The name to use.
    :type name: str.
    :param method: method to use.
    :type state: choice of ('empirical', 'parametric')
    :param accuracy:
    :type accuracy: integer
    :returns:  int -- the value
    :raises: AttributeError, KeyError
```

Writing documentation for scripts

Please follow the example in *<no title>* for documenting scripts. In addition, please pay attention to the following:

- Declare input data types for genomic data sets in optparse using the *metavar* keyword. For example:

  ```
  parser.add_option( "--extra-intervals", dest = "extra_intervals",
                  metavar="bed", help = "..." )
  ```

  Setting the type permits the script to be integrated into workflow sytemns such as galaxy.

- Please provide a meaningful example in the command line help.
- Be verbose. Something that is not documented within a script will not be used.
- Add meaningful tags to your scripts (`:Tags:`) so that they can be grouped into categories. Please choose from the following controlled vocabulary. If needed, additional terms can be added to this list.
  - Broad Themes
    - Genomics
    - NGS
    - MultipleAlignment
    - GenomeAlignment
    - Intervals
    - Genesets
    - Sequences
    - Variants
    - Protein
  - Formats
    - BAM
    - BED
    - GFF
    - GTF

    * FASTA
    * FASTQ
    * WIGGLE
    * PSL
    * CHAIN
  - Actions
    * Summary - summarizing entities within a file, such as counting the number of intervals within a file, etc.
    * Annotation - annotating individual entities within a file, such as adding length, composition, etc. to intervals.
    * Comparison - comparing the same type of entities, such as overlapping to sets of intervals.
    * Conversion - converting between different formats for the similar types of objects (Intervals in gff/bed format).
    * Transformation - transforming one entity into another, such as transforming intervals into sequences.
    * Manipulation - changing entities within a file, such as filtering sequences.

4.4 Documentation

4.4.1 Overview

CGAT scripts and modules use sphinx for documentation. The philosophy is to maintain documentation and code together. Thus, most documentation will be kept inside the actual scripts and modules, supported by overview documents explaining usage and higher level concepts.

4.4.2 Building the documentation

CGAT's documentation lives in the `doc` directory of the repository. To build the documentation, enter the `doc` directory and type:

```
make html
```

The output will be in the directory `_build/html`.

Note: Each script, module and pipeline needs to be importable, i.e, the following must work:

```
python -c "import pipeline_mapping"
```

Especially in pipelines some care is necessary to avoid failing with an error if no input or configuration files are present.

4.4.3 Writing documentation

sphinx documentation is written in Restructured Text. A useful primer is here.

Some specifics for the CGAT code base are:

- Refering to a separate script can be done using the `:doc:` directive, for example:

```
:doc:`scripts/bed2summary`
```

Note that the path relative to the current directory needs to be supplied.

- Glossary terms (`:term:`) are defined in `glossary.rst`.

4.4.4 Adding documentation

In order to add a new script, module or pipeline to the documentation documement, perform the following steps.

Here, we will be adding the script `bed2summary.py` to the documentation.

1. Create a file `doc/scripts/bed2summary.rst` with the following contents:

   ```
   .. automodule:: bed2summary

   .. program-output:: python ../scripts/bed2summary.py --help
   ```

   This will build the documentation within the bed2summary script and add the command line help to the document.

2. Add an entry to `doc/scripts.rst`. For example:

   ```
   .. toctree::

      scripts/bed2summary
   ```

   Please add your script to the toctree of an existing group.

3. For scripts that are part of the CGAT code collection, also add an entry into `doc/CGATReference.rst`.

Adding a module or pipeline is similar to adding a script, except that:

1. the `.rst` file should be in `doc/modules` or `doc/pipelines`, respectively.
2. The entry needs to be added to `modules.rst` or `CGATPipelines.rst`, respectively.
3. no `program-output` is necessary.

4.4.5 Requisites

Building the documentation requires the following components:

sphinx The documenation building system.

sphinxcontrib-programoutput Adding command line output to documenation.

4.5 Release notes

The CGAT code is currently unreleased. It is available through repository access only.

4.5.1 Contributions

We included publicly and freely available code into the tool collection for convenience.

- IGV.py was written by Brent Pedersen.
- SVGdraw.py was written by ...

- The NCL module draws from code written by ...
- list_overlap.py
- Iterators.py

Contributors

Andreas Heger Antonio Berlanga-Taylor Martin Dienstbier Nicholas Ilott Jethro Johnson Katherine Fawcett Stephen Sansom David Sims Ian Sudbery Hu Xiaoming

4.6 Importing CGAT scripts into galaxy

4.6.1 General Preparation

Add `/ifs/devel/cgat` to `PYTHONPATH`.

Make sure that extensions have been built:

```
python setup.py develop --multi-version
```

The following directories are important:

galaxy-dist Location of the galaxy distribution

cgat-xml CGAT directory within the galaxy distribution. Create by typing:

```
mkdir <galaxy-dist>/tools/cgat
```

cgat-scripts The CGAT scripts directory.

4.6.2 Adding a script manually

The following instructions describe the steps necessary to add a cgat script to galaxy.

For example, we want to publish the `bam2stats.py` script. First, create a file in `<galaxy-dist>/tools/cgat` called `bam2stats.xml` with the following contents:

```
<tool id="bam2stats.py" name="Compute Stats from BAM file">
  <description>Compute stats for a bam file</description>
    <command
      interpreter="python">/ifs/devel/cgat/scripts/bam2stats.py -v 0 &lt; $input &gt; $output
    </command>
    <inputs>
       <param format="bam" name="input" type="data" label="Source file"/>
    </inputs>
    <outputs>
       <data format="tabular" name="output" />
    </outputs>
  <help>
  Compute statistics for a bam file.
  </help>
</tool>
```

Add an entry to `tool_conf.xml` for the script:

```
<section name="CGAT Tools" id="cgat_tools">
  <tool file="cgat/bam2stats.xml" />
</section>
```

After restarting galaxy, the `bam2stats` command should now be visible in the `CGAT` section.

4.6.3 Automatic conversion of scripts

The CGAT tool collection contains a script called *<no title>* that can create and xml file for inclusion into galaxy. To create a wrapper for *<no title>*, run:

```
python <cgat-scripts>cgat2rdf.py --format=galaxy <cgat-scripts>bam2stats.py > <cgat-xml>bam2stats.xml
```

As before, add an entry to `tool_conf.xml` for the script.

For automatted conversion, a few rules need to be followed (see below).

Writing galaxy compatible scripts

CGAT scripts have generally a call interface that is compatible with galaxy and can thus be easily integrated. However, to make automatic conversion as easy as possible, conforming to a few coding conventions help.

1. Assign a metavar type to command line options of genomic file formats. For example:

   ```
   parser.add_option("-b", "--bam-file", dest="bam_files", type="string", metavar="bam",
                     help="filename with read mapping"
                          " information. Multiple files can be "
                          " submitted in a comma-separated list"  )
   ```

2. Use Experiment.OptionParser instead of optparse.OptionParser. The former has some extensions that make creating galaxy xml files easier. In particular, Experiment.OptionParser permits supplying a list of ‘,’-separated values to options that accept multiple values.

3. Follow the CGAT script naming convention. If possible, scripts should be named `<format_in>2<format_out>.py`. Formats can be mapped to other types in *<no title>*. For example, `stats` and `table` are both mapped to the format `tabular`.

CHAPTER 5

Glossary

5.1 File formats

yaml Language to serialize objects. Used in the CGAT testing framework. (YAML).

bam Format to store genomic alignments in a compressed format. (BAM).

bed File containing genomic intervals. (BED).

vcf Variant call format.

gtf General transfer format. Format to store genes and transcripts.

gff General feature format.

bigwig Compressed format for displaying numerical values across genomic ranges (BIGWIG).

fasta Sequence format.

wiggle Format for displaying numerical values across genomic ranges (Wiggle).

psl Genomic alignment format. The format is described in detail (PSL.

sam Format to store genomic alignments (SAM).

gdl gdl

tsv Tab separated values. In these tables, records are separated by new-line characters and fields by tab characters. Lines with comments are started by the # character and are ignored. The first uncommented line should contain the column headers. For example:

```
# This is a comment
gene_id       length
gene1 1000
gene2 2000
# Another comment
```

svg pass

edge list pass

fastq Sequence format containing quality scores, more background is here

sra sra

axt axt

maf maf

rdf Resource description framework

5.2 Other terms

test directory Directory that contains the `test.yaml`, input and reference files for testing scripts.

experiment experiment

replicate replicate

graph graph

track track

graph graph

submit host pass

execution host pass

edge list pass

task pass

sphinxreport sphinxreport

query pass

target pass

code directory pass

go pass

goslim pass

fastq pass

tss Transcription start site

production pipeline A pipeline that performs common tasks on a certain type of data. The idea of a production pipeline is to provide common preprocessing of data and a first look. A *project pipeline* might then take data from one or more *production pipeline* to glean biological insight.

project pipeline A pipeline that is project specific. Usually code is developed first inside a project pipeline. When it becomes generally useful, it may be refactored into a production pipeline.

stdin Unix standard input. Most CGAT tools read data from stdin.

stdout Unix standard output. Most CGAT tools output data to stdout.

stderr Unix standard error. This is where errors go.

loglevel Verbosity of logging information. The logging level can be determined by the `--verbose` option. A level of `0` means no logging output, while `1` is information messages only, while `2` outputs also debugging information.

CHAPTER 6

Disclaimer

The collection of scripts and tools is the outcome of 10 years working in various fields in bioinformatics. It contains both the good, the bad and the ugly. Use at your own risk.

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

t

V

W