libGWAS API Documentation

Release 1.0.0

Todd Edwards and Eric Torstenson

Nov 30, 2016

CHAPTER

ONE

LIBGWAS LIBRARIES

The following represents the API functionality associated with the meanvar application which includes a single interface for extracting data from each of the supported file types (libgwas). The contents below are only of interest for those who wish to extend MVtest or utilize libGWAS in their own GWAS analysis programs.

libGWAS is released under the Gnu Public license version 3 (http://www.gnu.org/licenses/gpl-3.0.en.html).

1.1 libGWAS package

libGWAS provides a singular interface for using several GWAS data formats such as Pedigree (ped), Translated Pedigree (tped), Binary Pedigree (bed) and two common imputed formats, IMPUTE and MACH as well as each of the accompanying files such as marker or family data. Support for plink style phenotype and covariate formatted files are also provided.

1.1.1 libgwas.bed_parser module

class `libgwas.bed_parser.Parser`(*fam*, *bim*, *bed*)

Bases: *libgwas.transposed_pedigree_parser.Parser*

`ReportConfiguration`(*file*)

Report configuration for logging purposes.

Parameters `file` -- Destination for report details

Returns None

`alleles` = **None**

Alleles for each locus

`bed_file` = **None**

Filename associated with the binary allele information (in variant major format only)

`bim_file` = **None**

filename for marker info in PLINK .bim format

`extract_genotypes`(*bytes*)

Extracts encoded genotype data from binary formatted file.

Parameters `bytes` -- array of bytes pulled from the .bed file

Returns standard python list containing the genotype data

Only ind_count genotypes will be returned (even if there are a handful of extra pairs present).

`fam_file` = **None**
Filename associated with the pedigree data (first 6 columns from standard pedigree: fid, iid, fid, mid, sex, pheno)

`families` = **None**
Pedigree information for reporting

`filter_missing()`
Filter out individuals and SNPs that have too many missing to be considered

Returns None

This must be run prior to actually parsing the genotypes because it initializes the following instance members:

- ind_mask
- total_locus_count
- locus_count
- data_parser.boundary (adds loci with too much missingness)

`geno_conversions` = **None**
Genotype conversion

`genotype_file` = **None**
Actual pedigree file being parsed (file object)

`ind_count` = **None**
Number of valid individuals

`ind_mask` = **None**
Mask indicating valid samples

`init_genotype_file()`
Resets the bed file and preps it for starting at the start of the genotype data

Returns to beginning of file and reads the version so that it points to first marker's info

Returns None

`initialize(`*map3=False*, *pheno_ covar=None*`)`

`load_bim(`*map3=False*`)`
Basic marker details loading.

(chr, rsid, gen. dist, pos, allelel 1, allele2)

Parameters `map3` -- When true, ignore the genetic distance column

Returns None

`load_fam(`*pheno_ covar=None*`)`
Load contents from the .fam file, updating the pheno_covar with family ids found.

Parameters `pheno_covar` -- Phenotype/covariate object

Returns None

`load_genotypes()`
Prepares the file for genotype parsing.

Returns None

`markers` = **None**
Valid loci to be used for analysis

`name` = **None**
Name used for reporting information about this dataset

`populate_iteration`(*iteration*)
Parse genotypes from the file and iteration with relevant marker details.

Parameters `iteration` -- ParseLocus object which is returned per iteration

Returns True indicates current locus is valid.

StopIteration is thrown if the marker reaches the end of the file or the valid genomic region for analysis.

1.1.2 libgwas.boundary module

class `libgwas.boundary.BoundaryCheck`(*bp=(None, None)*, *kb=(None, None)*, *mb=(None, None)*)
Bases: `object`

Record boundary specifications from user to control traversal.

Default boundaries are specified in numerical positions along a single chromosome. Users are permitted to provide boundaries in 3 forms: Bases, Kilobasees and Megabases. All are recorded as single base offsets from the beginning of the chromosome (starting at 1).

The valid setting doesn't mean the boundary object is invalid, only that no actual boundary ranges have been provided. This is done to allow the user interface code to be a little simpler (i.e. if the user didn't provide bounds using numerical boundaries, it can try instantiating a SnpBoundary and pass the relevant arguments to that object. If none are valid, then either can be used, at which point both act as chromosome boundaries or simple SNP filters)

If chrom is specfied, all SNPs and boundaries are expected to reside on that chromosome.

`LoadExclusions`(*snps*)
Load locus exclusions.

Parameters `snps` -- Can either be a list of rsids or a file containing rsids.

Returns None

If snps is a file, the file must only contain RSIDs separated by whitespace (tabs, spaces and return characters).

`LoadSNPs`(*snps=[]*)
Define the SNP inclusions (by RSID). This overrides true boundary definition.

Parameters `snps` -- array of RSIDs

Returns None

This doesn't define RSID ranges, so it throws InvalidBoundarySpec if it encounters what appears to be a range (SNP contains a "-")

`NoExclusions`()
Determine that there are no exclusion criterion in play

Returns True if there is no real boundary specification of any kind.

Simple method allowing parsers to short circuit the determination of missingness, which can be moderately compute intensive.

`ReportConfiguration(f)`
Report the boundary configuration details

Parameters `f` -- File (or standard out/err)

Returns None

`TestBoundary(chr, pos, rsid)`
Test if locus is within the boundaries and not to be ignored.

Parameters

- `chr` -- Chromosome of locus
- `pos` -- BP position of locus
- `rsid` -- RSID (used to check for exclusions)

Returns True if locus isn't to be ignored

`beyond_upper_bound` = **None**
Is set once the upper limit has been exceeded

`bounds` = **None**
Actual boundary details in BP

`chrom` = **-1**

`dropped_snps` = **None**
Indices of loci that are to be dropped {chr=>[pos1, pos2, ..., posN]}

`ignored_rs` = **None**
List of RS Numbers to be ignored

`target_rs` = **None**
List of RS Numbers to be targeted (ignors all but those listed)

`valid` = **None**
True if boundary conditions remain true

1.1.3 libgwas.data_parser module

class `libgwas.data_parser.DataParser`
Bases: `object`

Abstract representation of all dataset parsers

`boundary` = <**libgwas.boundary.BoundaryCheck object**>
Boundary object specifying valid region for analysis

`compressed_pedigree` = **False**
When true, assume that standard pedigree and transposed pedigree are compressed with gzip

`get_effa_freq(genotypes)`

`get_loci()`

`has_fid` = **True**
When false, pedigree header expects no family id column

`has_liability` = **False**
When false, pedigree header expects no liability column

`has_parents` = **True**
When false, pedigree header expects no parents columns

`has_pheno` = **True**
When false, pedigree header expects no phenotype column

`has_sex` = **True**
When false, pedigree header expects no sex column

`ind_exclusions` = **[]**
Filter out specific individuals by individual ID

`ind_inclusions` = **[]**
Filter in specific individuals by individual ID

`ind_miss_tol` = **1.0**
Filter individuals with too many missing

`max_maf` = **1.0**
filter out if a minor allele frequency exceeds this value

`min_maf` = **0.0**
this can be used to filter out loci with too few minor alleles

`missing_representation` = **'0'**
External representation of missingness

`missing_storage` = **-1**

`snp_miss_tol` = **1.0**
Filter SNPs with too many missing

static `valid_indid`(*indid*)

`libgwas.data_parser.check_inclusions`(*item*, *included=[]*, *excluded=[]*)
Everything passes if both are empty, otherwise, we have to check if empty or is present.

1.1.4 libgwas.exceptions module

exception `libgwas.exceptions.InvalidBoundarySpec`(*malformed_ boundary*)
Bases: *libgwas.exceptions.ReportableException*

Indicate boundary specification was malformed or non-sensical

exception `libgwas.exceptions.InvalidSelection`(*msg*)
Bases: *libgwas.exceptions.MalformedInputFile*

Indicate that the user provided input that is meaningless.

This is likely a situation where the user provided an invalid name for a phenotype or covariate. Probably a misspelling.

exception `libgwas.exceptions.InvariantVar`(*msg=''*)
Bases: *libgwas.exceptions.ReportableException*

No minor allele found

exception `libgwas.exceptions.MalformedInputFile`(*msg*)
Bases: *libgwas.exceptions.ReportableException*

Error encountered in data from an input file

exception `libgwas.exceptions.NanInResult`(*msg=''*)
Bases: *libgwas.exceptions.ReportableException*

NaN found in result

exception libgwas.exceptions.NoMatchedPhenoCovars(*msg=''*)
Bases: *libgwas.exceptions.ReportableException*

No ids matched between pheno or covar and the family data

exception libgwas.exceptions.ReportableException(*msg*)
Bases: exceptions.Exception

Simple exeception with message

exception libgwas.exceptions.TooFewAlleles(*chr=None, rsid=None, pos=None, alleles=None, index=None*)
Bases: *libgwas.exceptions.TooManyAlleles*

Indicate fixed allele was found

exception libgwas.exceptions.TooManyAlleles(*chr=None, rsid=None, pos=None, alleles=None, index=None, prefix='Too many alleles: '*)
Bases: *libgwas.exceptions.ReportableException*

Indicate locus found with more than 2 alleles

alleles = **None**
Allele 1 and 2

chr = **None**
Chromosome

index = **None**
Index of the locus within the file

pos = **None**
BP Position

rsid = **None**
RSID

exception libgwas.exceptions.UnsolvedLocus(*msg*)
Bases: *libgwas.exceptions.ReportableException*

1.1.5 libgwas.impute_parser module

class libgwas.impute_parser.Encoding
Bases: object

Simple enumeration for various model encodings

Additive = **0**

Dominant = **1**

Genotype = **3**

Raw = **4**

Recessive = **2**

class libgwas.impute_parser.Parser(*fam_details, archive_list, chroms, info_files=[]*)
Bases: *libgwas.data_parser.DataParser*

Parse IMPUTE style output.

ReportConfiguration(*file*)

Parameters file -- Destination for report details

Returns None

`archives` = **None**
This is only the list of files to be processed

`chroms` = **None**
List of chroms to match files listed in archives

`current_chrom` = **None**
This will be used to record the chromosome of the current file

`current_file` = **None**
This will be used to record the opened file used for parsing

`current_info` = **None**
This will be used to record the info file associated with quality of SNPs

`fam_details` = **None**
single file containing the subject details (similar to plink's .fam)

`gen_ext` = **'gen.gz'**
The genotype file suffix (of not following convention)

`get_effa_freq`(*genotypes*)
Returns the effect allele's frequency

`get_next_line`()
If we reach the end of the file, we simply open the next, until we run out of archives to process

`info_ext` = **'info'**
the extension associated with the .info files if not using conventions

`info_files` = **None**
array of .info files

`info_threshold` = **0.4**
The threshold associated with the .info info column

`load_family_details`(*pheno_ covar*)
Load family data updating the pheno_covar with family ids found.

Parameters `pheno_covar` -- Phenotype/covariate object

Returns None

`load_genotypes`()
Prepares the files for genotype parsing.

Returns None

`populate_iteration`(*iteration*)
Parse genotypes from the file and iteration with relevant marker details.

Parameters `iteration` -- ParseLocus object which is returned per iteration

Returns True indicates current locus is valid.

StopIteration is thrown if the marker reaches the end of the file or the valid genomic region for analysis.

`libgwas.impute_parser.SetEncoding`(*sval*)
Sets the encoding variable according to the text passed

Parameters `sval` -- text specification for the desired model

1.1.6 libgwas.locus module

class `libgwas.locus.Locus`(*other=None*)

Bases: `object`

`alleles` = **None**
List of alleles present

`chr` = **None**
Chromosome

`exp_hetero_freq`
Returns the estimated frequency of heterozygotes

`flip()`
This will switch major/minor around, regardless of frequency truth.

This is intended for forcing one of two populations to relate correctly to the same genotype definitions. When flipped, Ps and Qs will be backward, and the maf will no longer relate to the "minor" allele frequency. However, it does allow clients to use the same calls for each population without having to perform checks during those calculations.

`hetero_count` = **None**
total count of heterozygotes observed

`hetero_freq`
Returns the frequency of observed heterozygotes (not available with all parsers)

`maf`
Returns the MAF. This is valid for all parsers

`maj_allele_count` = **None**
total number of major alleles observed

`major_allele`
Sets/Returns the encoding for the major allele (A, C, G, T, etc)

`min_allele_count` = **None**
total number of minor alleles observed

`minor_allele`
Sets/Returns the encoding for minor allele

`missing_allele_count` = **None**
total number of missing alleles were observed

`p`
Frequency for first allele

`pos` = **None**
BP Position

`q`
Frequency for second allele

`rsid` = **None**
RSID

`sample_size`
Returns to total sample size

`total_allele_count`
Returns the total number of alleles

1.1.7 libgwas.mach_parser module

class libgwas.mach_parser.Encoding
Bases: object

Dosage = **0**
Currently there is only one way to interpret these values

class libgwas.mach_parser.Parser(*archive_list*, *info_files=[]*)
Bases: *libgwas.data_parser.DataParser*

Parse IMPUTE style output.

Due to the nature of the mach data format, we must load the data first into member before we can begin analyzing it. Due to the massive amount of data, SNPs are loaded in in chunks.

ISSUES:

- Currently, we will not be filtering on individuals except by explicit removal
- We are assuming that each gzip archive contains all data associated with the loci contained within (i.e. there won't be separate files with different subjects inside) ((Todd email jan-9-2015))
- There is no place to store RSID from the output that I've seen (Minimac output generated by Ben Zhang). As of Feb 2016, I've made the chr:pos ID requirment optional, and added in a 3rd column (optional for even the option) which can be rsid if present (currently, the data we have puts alleles in that column, but future imputations can be done differently). When using the mach-chrpos flag, users can produce results that appear similar to plink output with chromosome, position and optionally RSIDs in the expected columns. This is, by default, turned off, and the entire contents of that ID column is stored as simply the RSID when reporting results.

ReportConfiguration(*file*)
Report the configuration details for logging purposes.

Parameters file -- Destination for report details

Returns None

chrpos_encoding = **False**

chunk_stride = **50000**
Number of loci to parse at a time (larger stride requires more memory)

dosage_ext = **'dose.gz'**
Extension for the dosage file

get_effa_freq(*genotypes*)
Returns the frequency of the effect allele

info_ext = **'info.gz'**
Extension for the info file

load_family_details(*pheno_covar*)
Load contents from the .fam file, updating the pheno_covar with family ids found.

Parameters pheno_covar -- Phenotype/covariate object

Returns None

load_genotypes()
Actually loads the first chunk of genotype data into memory due to the individual oriented format of MACH data.

Due to the fragmented approach to data loading necessary to avoid running out of RAM, this function will initialize the data structures with the first chunk of loci and prepare it for otherwise normal iteration.

Also, because the parser can be assigned more than one .gen file to read from, it will automatically move to the next file when the first is exhausted.

`min_rsquared` = **0.3**
rsquared threshold for analysis (obtained from the mach output itself)

`openfile`(*filename*)

`parse_genotypes`(*lb*, *ub*)
Extracts a fraction of the file (current chunk of loci) loading the genotypes into memoery.

Parameters

- `lb` -- Lower bound of the current chunk
- `ub` -- Upper bound of the current chunk

Returns Dosage dosages for current chunk

`populate_iteration`(*iteration*)
Parse genotypes from the file and iteration with relevant marker details.

Parameters `iteration` -- ParseLocus object which is returned per iteration

Returns True indicates current locus is valid.

StopIteration is thrown if the marker reaches the end of the file or the valid genomic region for analysis.

This function will force a load of the next chunk when necessary.

1.1.8 libgwas.parsed_locus module

class `libgwas.parsed_locus.ParsedLocus`(*datasource*, *index=-1*)
Bases: *libgwas.locus.Locus*

Locus data representing current iteration from a dataset

Provide an iterator interface for all dataset types.

`cur_idx` = **None**
Index within the list of loci being analyzed

`genotype_data` = **None**
Actual genotype data for this locus

`next`()
Move to the next valid locus.

Will only return valid loci or exit via StopIteration exception

1.1.9 libgwas.pedigree_parser module

class `libgwas.pedigree_parser.Parser`(*mapfile*, *datasource*)
Bases: *libgwas.data_parser.DataParser*

Parse standard pedigree dataset.

Data should follow standard format for pedigree data, except alleles be either numerical (1 and 2) or as bases (A, C, T and G). All loci must have 2 alleles to be returned.

Attributes initialized to None are only available after load_genotypes() has been called.

Issues:

- Pedigree files are currently loaded in their entirety, but we could load them in according to chunks like we are doing in mach input.
- There are a bunch of legacy lists which should be reduced to a single list of Locus objects.

`ReportConfiguration(`*file*`)`
Report configuration for logging purposes.

Parameters `file` -- Destination for report details

Returns None

`alleles` = **None**
List of both alleles for each valid locus

`datasource` = **None**
Filename for the actual pedigree information

`genotypes` = **None**
Matrix of genotype data

`get_loci()`

`individual_mask` = **None**
Mask used to remove excluded and filtered calls from the genotype data (each position represents an individual)

`invalid_loci` = **None**
Loci that are being ignored due to filtration

`load_genotypes(`*pheno_ covar*`)`
Load all data into memory and propagate valid individuals to pheno_covar.

Parameters `pheno_covar` -- Phenotype/covariate object is updated with subject

information :return: None

`load_mapfile(`*map3=False*`)`
Load the marker data

Parameters `map3` -- When true, ignore the gen. distance column

Builds up the marker list according to the boundary configuration

`locus_count` = **None**
Number of valid loci

`mapfile` = **None**
Filename for the marker information

`markers` = **None**
List of valid Locus Objects

`markers_maf` = **None**
List of MAF at each locus

`name` = **None**
Name used for reporting information about this dataset

`populate_iteration`(*iteration*)
Parse genotypes from the file and iteration with relevant marker details.

Parameters `iteration` -- ParseLocus object which is returned per iteration

Returns True indicates current locus is valid.

StopIteration is thrown if the marker reaches the end of the file or the valid genomic region for analysis.

`rsids` = **None**
List of all SNP names for valid loci

1.1.10 libgwas.pheno_covar module

class `libgwas.pheno_covar.PhenoCovar`
Bases: `object`

Store both phenotype and covariate data in a single object.

Provide iterable interface to allow evaluation of multiple phenotypes easily. Covariates do not change during iteration. Missing is updated according to the missing content within the phenotype (and covariates as well).

`add_subject`(*ind_id*, *sex=None*, *phenotype=None*)
Add new subject to study, with optional sex and phenotype

Throws MalformedInputFile if sex is can't be converted to int

`covariate_data` = **None**
All covariate data [[cov1],[cov2],etc]

`covariate_labels` = **None**
List of covariate names from header, if provided SEX is implied, if sex_as_covariate is true. Covariates loaded without header are simply named Cov-N

`destandardize_variables`(*tv*, *blin*, *bvar*, *errBeta*, *nonmissing*)
Destandardize betas and other components.

`do_standardize_variables` = **None**
Allows you to turn off standardization

`freeze_subjects`()
Converts variable data into numpy arrays.

This is required after all subjects have been added via the add_subject function, since we don't know ahead of time who is participating in the analysis due to various filtering possibilities.

`individual_mask` = **None**
True indicates an individual is to be excluded

`load_covarfile`(*file*, *indices=[]*, *names=[]*, *sample_file=False*)
Load covariate data from file.

Unlike phenofiles, if we already have data, we keep it (that would be the sex covariate)

`load_phenofile`(*file*, *indices=[]*, *names=[]*, *sample_file=False*)
Load phenotype data from phenotype file

Whitespace delimited, FAMID, INDID, VAR1, [VAR2], etc

Users can specify phenotypes of interest via indices and names. Indices are 1 based and start with the first variable. names must match name specified in the header (case is ignored).

`missing_encoding` = **-9**
Internal encoding for missingness

`pedigree_data` = **None**
Pedigree information {FAMID:INDID => index, etc}

`phenotype_data` = **None**
Raw phenotype data with every possible phenotype [[ph1],[ph2],etc]

`phenotype_names` = **None**
List of phenotype names from header, if provided. If no header is found, the phenotype is simply named Pheno-N

`prep_testvars()`
Make sure that the data is in the right form and standardized as expected.

`sex_as_covariate` = **False**
Do we use sex as a covariate?

`test_variables` = **None**
finalized data ready for analysis

1.1.11 libgwas.snp_boundary_check module

class `libgwas.snp_boundary_check.SnpBoundaryCheck`(*snps=[]*)
Bases: *libgwas.boundary.BoundaryCheck*

RS (or other name) based boundary checking.

Same rules apply as those for BoundaryCheck, except users can provide multiple RS boundary regions. Though, all boundary groups must reside on a single chromosome.

Class members (these are not intended for public consumption):

- start_bounds bp location for boundary starts Currently, only one boundary is permitted. This is to remain consistant with plink
- end_bounds bp location for boundary end (inclusive)
- ignored_rs List of RS numbers to be ignored
- target_rs List of RS numbers to be targeted
- **dropped_snps indices of loci that are to be dropped** {chr=>[pos1, pos2, ...]}
- **end_rs This is used during iteration to identify when to** turn "off" the current boundary group

`NoExclusions()`
Determine that there are no exclusion criterion in play

Returns True if there is no real boundary specification of any kind.

Simple method allowing parsers to short circuit the determination of missingness, which can be moderately compute intensive.

`ReportConfiguration`(*f*)
Report the boundary configuration details

Parameters `f` -- File (or standard out/err)

Returns None

`TestBoundary`(*chr*, *pos*, *rsid*)
Test if locus is within the boundaries and not to be ignored.

Parameters

- `chr` -- Chromosome of locus
- `pos` -- BP position of locus
- `rsid` -- RSID (used to check for exclusions)

Returns True if locus isn't to be ignored

1.1.12 libgwas.standardizer module

class `libgwas.standardizer.NoStandardization`(*pc*)
Bases: *libgwas.standardizer.StandardizedVariable*

This is mostly a placeholder for standardizers. Each application will probably have a specific approach to standardizing/destandardizing the input/output.

`destandardize`(*estimates*, *se*, ***kwargs*)
When the pheno/covar data has been standardized, this can be used to rescale the betas back to a meaningful value using the original data.

For the "Un-standardized" data, we do no conversion.

`standardize`()
Standardize the variables within a range [-1.0 and 1.0]

This replaces the local copies of this data. When it's time to scale back, use destandardize from the datasource for that.

class `libgwas.standardizer.StandardizedVariable`(*pc*)
Bases: `object`

Optional plugin object that can be used to standardize covariate and phenotype data.

Many algorithms require that input be standardized in some way in order to work properly, however, rescaling the results is algorithm specific. In order to facilitate this situation, application authors can write up application specific Standardization objects for use with the data parsers.

`covar_count` = **None**
number of covars

`covariates` = **None**
Standardized covariate data

`datasource` = **None**
Reference back to the pheno_covar object for access to raw data

`destandardize`()
Stub for the appropriate destandardizer function.

Each object type will do it's own thing here.

`get_covariate_name`(*idx*)
Return label for a specific covariate

Parameters `idx` -- which covariate?

Returns string label

`get_covariate_names()`
Return all covariate labels as a list

Returns list of covariate names

`get_phenotype_name()`
Returns current phenotype name

`get_variables(missing_in_geno=None)`
Extract the complete set of data based on missingness over all for the current locus.

Parameters `missing_in_geno` -- mask associated with missingness in genotype

Returns (phenotypes, covariates, nonmissing used for this set of vars)

`idx` = **None**
index of the current phenotype

`missing` = **None**
mask representing missingness (1 indicates missing)

`pheno_count` = **None**
number of phenotypes

`phenotypes` = **None**
standardized phenotype data

`standardize()`
Stub for the appropriate standardizer function

Each Standardizer object will do it's own thing here.

`libgwas.standardizer.get_standardizer()`

`libgwas.standardizer.set_standardizer(std)`

1.1.13 libgwas.transposed_pedigree_parser module

class `libgwas.transposed_pedigree_parser.Parser(tfam, tped)`
Bases: *libgwas.data_parser.DataParser*

Parse transposed pedigree dataset

Class Members: tfam_file filename associated with the pedigree information tped_file Filename associated with the genotype data families Pedigree information for reporting genotype_file Actual pedigree file begin parsed (file object)

`ReportConfiguration(file)`

`filter_missing()`
Filter out individuals and SNPs that have too many missing to be considered

`load_genotypes()`
This really just intializes the file by opening it up.

`load_tfam(pheno_covar)`
Load the pedigree portion of the data and sort out exclusions

`name` = **None**
Name used for reporting information about this dataset

`populate_iteration(iteration)`
Pour the current data into the iteration object

`process_genotypes(data)`
Parse pedigree line and remove excluded individuals from geno

Translates alleles into numerical genotypes (0, 1, 2) counting number of minor alleles.

Throws exceptions if an there are not 2 distinct alleles

1.1.14 Module contents

`libgwas.BuildReportLine(key, value, offset=None)`
Prepare key/value for reporting in configuration report

Parameters

- `key` -- configuration 'keyword'
- `value` -- value reported to be associated with keyword

Returns formatted line starting with a comment

`libgwas.Exit(msg, code=1)`
Exit execution with return code and message :param msg: Message displayed prior to exit :param code: code returned upon exiting

`libgwas.ExitIf(msg, do_exit, code=1)`
Exit if do_exit is true

Parameters

- `msg` -- Message displayed prior to exit
- `do_exit` -- exit when true
- `code` -- application's return code upon exit

`libgwas.sys_call(cmd)`
Execute cmd and capture stdout and stderr

Parameters `cmd` -- command to be executed

Returns (stdout, stderr)

|

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V