Genomalicious tutorial 1: Basic ingredients

Joshua A. Thia

18 December 2022

Preamble

Every good recipe is built from basic ingredients. Likewise, bioinformatic pipelines, while complex, require some fundamental methods and data types that are the bare neccessities to any study implementing population genomic approaches.

In this tutorial, you will:

1. Become familiar with some basic features of *genomalicious*.
2. Develop familiarity with SNP data structures.
3. Learn to use *genomalicious* functions to import and manipulate SNP data structures.

Getting to know *genomalicious*

Let's start by loading the *genomalicious* library into your R session.

```
library(genomalicious)
```

Genomalicious contains a number of demonstrative toy datasets that you can experiment on. These need to be loaded in with the `data` function and typically follow the naming convention, `data_[data name]`, where `[data name]` is a unique identifier.

For example, let's take a look at a toy dataset of genotype data from four simulated populations:

```
data("data_Genos")

# Take a look at the structure of this data, the `dim` function reports the dimensions.
dim(data_Genos)
```

```
## [1] 20000    11
```

```
# The class
class(data_Genos)
```

```
## [1] "data.table" "data.frame"
```

```
# Print the data to screen.
data_Genos
```

```
##             CHROM POS          LOCUS  POP  SAMPLE GT DP AO RO ALT REF
##     1:    Contig1 437    Contig1_437 Pop1  Ind1_1  1 27 11 16   A   T
##     2:   Contig34 213   Contig34_213 Pop1  Ind1_1  1 22 17  5   G   A
##     3:   Contig38 427   Contig38_427 Pop1  Ind1_1  0 46  0 46   A   G
##     4:  Contig152 389  Contig152_389 Pop1  Ind1_1  0 19  0 19   T   A
##     5:  Contig170  36   Contig170_36 Pop1  Ind1_1  1 38 20 18   G   A
##    ---
## 19996: Contig7274 197 Contig7274_197 Pop4 Ind4_25  1 52 24 28   G   C
## 19997: Contig7287  10  Contig7287_10 Pop4 Ind4_25  0 59  0 59   T   C
## 19998: Contig7291  92  Contig7291_92 Pop4 Ind4_25  2 39 39  0   G   T
## 19999: Contig7293 124 Contig7293_124 Pop4 Ind4_25  2 30 30  0   A   T
## 20000: Contig7299 279 Contig7299_279 Pop4 Ind4_25  1 49 15 34   G   T
```

You will notice that `data_Genos` is a dual classed object: a `data.table` and a `data.frame`. This is a long-format data table, sample IDs and locus IDs are in the columns (`SAMPLE` and `LOCUS`, respectively), and the genotypes a recorded in a single column (`GT`). There is also information on the population ID (`POP`) and the chromosome (`CHROM`).

You can learn more about a dataset by using `?` and the data object name, e.g. `?data_Genos`. We will look at other data sets in later tutorials, but you can peruse the various datasets by typing `genomalicious::data_` and hitting the TAB key to get a list of options.

Loading VCFs into R

Variant call files (VCFs) are one of the typical end products from read assembly/variant calling pipelines in population genomic analyses. These files contain information about genotypes and read counts attributed to each sample and associated mapping and genotype calling statistics.

Genomalicious offers a very simple way to import VCFs into R as **long-format** data tables, whereby loci and and populations are both in rows, as described above.

We will now import a demo VCF into R using the `vcf2DT` function. First, we need to find where *genomalicious* is installed and make a path to the demo file.

```
# Create a link to raw external datasets in genomalicious
genomaliciousExtData <- paste0(find.package('genomalicious'), '/extdata')

# This command here shows you the VCF file that comes with genomalicious
list.files(path=genomaliciousExtData, pattern='indseq.vcf')
```

```
## [1] "data_indseq.vcf"
```

```
# Use this to create a path to that file
vcfPath <- paste0(genomaliciousExtData, '/data_indseq.vcf')

# The value of vcfPath will depend on your system
vcfPath
```

```
## [1] "C:/Users/Dr_Thia/AppData/Local/R/win-library/4.3/genomalicious/extdata/data_indseq.vcf"
```

You can naviagte yourself to the path stored in `vcfPath` and open the VCF using a text editor. However, we can simply read the lines of the file and print them to screen:

```r
# Read in and print first 20 lines of the demo VCF
readLines(vcfPath)[1:20]
```

```
##  [1] "##This is a toy dataset for the R package genomalicious - it emulates a VCF file"
##  [2] "##INFO=<ID=DP,Number=1,Type=Integer,Description='The total depth across samples'>"
##  [3] "##FORMAT=<ID=GT,Number=1,Type=String,Description='Genotype'>"
##  [4] "##FORMAT=<ID=DP,Number=1,Type=Integer,Description='The total depth in a sample'>"
##  [5] "##FORMAT=<ID=RO,Number=1,Type=Integer,Description='The reference allele counts in a sample'>"
##  [6] "##FORMAT=<ID=AO,Number=1,Type=Integer,Description='The alternate allele counts in a sample'>"
##  [7] "#CHROM\tPOS\tREF\tALT\tINFO\tQUAL\tFORMAT\tInd1_1\tInd1_10\tInd1_11\tInd1_12\tInd1_13\tInd1_14
##  [8] "Contig1\t437\tT\tA\tDP=3587\t30\tGT:DP:RO:AO\t0/1:27:16:11\t0/0:33:33:0\t0/1:51:33:18\t0/0:35:
##  [9] "Contig34\t213\tA\tG\tDP=3408\t30\tGT:DP:RO:AO\t0/1:22:5:17\t0/0:35:35:0\t0/0:46:46:0\t0/1:14:8
## [10] "Contig38\t427\tG\tA\tDP=3418\t30\tGT:DP:RO:AO\t0/0:46:46:0\t0/0:14:14:0\t0/0:46:46:0\t0/0:36:3
## [11] "Contig152\t389\tA\tT\tDP=3476\t30\tGT:DP:RO:AO\t0/0:19:19:0\t0/0:41:41:0\t0/0:34:34:0\t0/0:30:
## [12] "Contig170\t36\tA\tG\tDP=3517\t30\tGT:DP:RO:AO\t0/1:38:18:20\t0/1:27:9:18\t0/0:31:31:0\t1/1:31:
## [13] "Contig183\t250\tA\tC\tDP=3590\t30\tGT:DP:RO:AO\t1/1:44:0:44\t1/1:43:0:43\t0/1:41:16:25\t1/1:39
## [14] "Contig263\t267\tA\tT\tDP=3525\t30\tGT:DP:RO:AO\t1/1:24:0:24\t0/0:36:36:0\t0/0:46:46:0\t0/0:24:
## [15] "Contig346\t278\tG\tT\tDP=3393\t30\tGT:DP:RO:AO\t0/1:47:26:21\t0/1:27:14:13\t0/1:34:15:19\t0/0:
## [16] "Contig425\t300\tC\tG\tDP=3687\t30\tGT:DP:RO:AO\t0/1:38:19:19\t1/1:21:0:21\t0/1:35:24:11\t0/1:3
## [17] "Contig428\t69\tA\tC\tDP=3587\t30\tGT:DP:RO:AO\t1/1:19:0:19\t1/1:29:0:29\t1/1:28:0:28\t1/1:14:0
## [18] "Contig437\t496\tG\tT\tDP=3572\t30\tGT:DP:RO:AO\t0/1:26:13:13\t0/1:67:34:33\t0/0:30:30:0\t1/1:4
## [19] "Contig439\t463\tA\tG\tDP=3518\t30\tGT:DP:RO:AO\t0/1:33:16:17\t0/1:36:18:18\t0/1:26:13:13\t0/1:
## [20] "Contig440\t431\tT\tG\tDP=3450\t30\tGT:DP:RO:AO\t0/0:19:19:0\t0/0:21:21:0\t0/0:45:45:0\t0/0:43:
```

This demo VCF was constructed more simply for the purpose of this tutorial, but follows the same basic structure of that produced from variant calling software. You will see that the text is interspersed with `\t`: these indicate tabs that separate the various columns in the file.

A VCF has the following components:

1. A **comments section** marked with double hashes, `##`. These typically contain details about the contents of the VCF or how the reads were called.
2. A **heading section** marked with a single hash, `#`. This is effectively the column names for the wide format SNP data.
3. All lines proceeding contain SNP data.

VCFs can be a bit tricky to interpret when you first see them. All columns **before** `FORMAT` contain information about the SNP (i.e. the chromosome, its position, the alleles). `FORMAT` **itself** contains a text string detailing the contents of the sample columns. Note, the contents of `FORMAT` are typically described in the comments section. We therefore know that `DP:RO:AO` represent the total read depth (`DP`), and the counts of the reference (`RO`) and alternate (`AO`) allele. Also note the difference in `DP` stored in the `INFO` column (read depth across all samples) and that in `FORMAT` (read depth within each sample).

All sample columns occur **after** `FORMAT` (e.g. `Ind1.115`, `Ind1.243`) and continue to the end of the line. Within each sample column, the values described in `FORMAT` are stored, with each value separated by a `:`. This demo VCF contains 256 simulated individuals sampled from four populations, with naming conventions `Ind[pop]_[sample]`, were `pop` is the population ID and `sample` is the sample ID.

Note the difference in the VCF file as a **wide-format** data structure to that of the **long-format** data structure we saw earlier. In wide-format, samples are in columns, and the row represent some common measurement across those samples; in this case, genotypes at a SNP locus.

Now that we understand the data we are working with, let's import it into R.

```r
# Import VCF into R using the path name
indSnps <- vcf2DT(vcfPath)
```

```
## (1/4) Reading in VCF as a data table
## (2/4) Generating locus IDs
## (3/4) Converting from wide to long format
## (4/4) Parsing data for each sample
## All done! <3
```

```r
# First 8 rows of the imported SNP data
head(indSnps, 8)
```

```
##            LOCUS     CHROM POS REF ALT QUAL SAMPLE  GT DP RO AO
## 1:   Contig1_437   Contig1 437   T   A   30 Ind1_1 0/1 27 16 11
## 2:  Contig34_213  Contig34 213   A   G   30 Ind1_1 0/1 22  5 17
## 3:  Contig38_427  Contig38 427   G   A   30 Ind1_1 0/0 46 46  0
## 4: Contig152_389 Contig152 389   A   T   30 Ind1_1 0/0 19 19  0
## 5:  Contig170_36 Contig170  36   A   G   30 Ind1_1 0/1 38 18 20
## 6: Contig183_250 Contig183 250   A   C   30 Ind1_1 1/1 44  0 44
## 7: Contig263_267 Contig263 267   A   T   30 Ind1_1 1/1 24  0 24
## 8: Contig346_278 Contig346 278   G   T   30 Ind1_1 0/1 47 26 21
```

```r
# The imported data is stored as a data.table object
class(indSnps)
```

```
## [1] "data.table" "data.frame"
```

As you can see, the function `vcf2DT` converts the wide-format VCF data into a long-format data table. Instead of a single column for each sample (and loci in rows), all possible sample-by-locus combinations are stored in the rows of `indSnps`, with a single column each for samples and loci. Using the `$` notation, you can access these columns as vectors.

```r
# Column vector for sample and loci
head(indSnps$SAMPLE)
```

```
## [1] "Ind1_1" "Ind1_1" "Ind1_1" "Ind1_1" "Ind1_1" "Ind1_1"
```

```r
head(indSnps$LOCUS)
```

```
## [1] "Contig1_437"   "Contig34_213"  "Contig38_427"  "Contig152_389"
## [5] "Contig170_36"  "Contig183_250"
```

Typically in these tutorials, and throughout the documentation for *genomalicious*, I will use `$` to indicate nested vectors of R objects (e.g. columns, list items).

You would have noticed a column in `indSnps` called `GT`: this column contains information on each sample's genotype at a particular locus. '0' is the reference allele and '1' is the alternate allele; each allele is separated by a '/'. This is a **separated** format of genotype values as **character** class. However, another way of representing genotypes is as **counts** of one of the alleles, e.g. '0', '1', or '2', as an **integer** class.

When genotypes are represented as allele counts, we assume biallelic data because it is only possible to keep track of a maximum of two unique alleles. In reality, there are other ways to work around this, but many population genomic analyses are constrained to be on biallelic SNPs, so we will focus on those here.

The function `genoscore_converter` can be used to convert genotypes between the biallelic scoring formats. The input into this function is simply a vector of genotypes. If inputting the separated format, the vector must be a `character` class object. If inputting allele count format, the vector is ideally `integer` class object. The function bases its counts of the **alternate allele**, hence a genotype of '0/0', '0/1', and '1/1', are equivalent to 0, 1, and 2 (respectively).

```
# Practise run with simple vectors
genoscore_converter(c('0/0', '0/1', '1/1'))
```

```
## [1] 0 1 2
```

```
genoscore_converter(c(0L, 1, 2))
```

```
## [1] "0/0" "0/1" "1/1"
```

```
# Now convert the first 15 genotypes from separated to count format.
genoscore_converter(indSnps$GT[1:15])
```

```
##  [1] 1 1 0 0 1 2 2 1 1 2 1 1 0 1 0
```

```
indSnps$GT[1:15]
```

```
##  [1] "0/1" "0/1" "0/0" "0/0" "0/1" "1/1" "1/1" "0/1" "0/1" "1/1" "0/1" "0/1"
## [13] "0/0" "0/1" "0/0"
```

Data tables for storing SNP data

Genomalicious is largely built around `data.table` classed objects. Data tables feel and more-or-less function as per standard R `data.frame` objects, but they differ in two major ways. Firstly, they are much more efficient at storing large volumes of data. Secondly, they have some nifty features that facilitate easy data manipulations. Though the purpose of this tutorial is not to provide a detailed demonstration of `data.table` object features (you can find that, here), let's just take a quick look at how you can harness the power of data tables.

In the simplest case, objects of class `data.table` can be manipulated using the semantics `D[i, j, by]`. If `D` is the data table, `i` represents rows, `j` represents columns, and `by` represents an operation we would like to perform on the data.

Just like a regular object of class `data.frame` we can subset a data table using integer indexes and column names.

```
indSnps[1:5,]
```

```
##            LOCUS     CHROM POS REF ALT QUAL SAMPLE  GT DP RO AO
## 1:   Contig1_437   Contig1 437   T   A   30 Ind1_1 0/1 27 16 11
## 2:  Contig34_213  Contig34 213   A   G   30 Ind1_1 0/1 22  5 17
## 3:  Contig38_427  Contig38 427   G   A   30 Ind1_1 0/0 46 46  0
## 4: Contig152_389 Contig152 389   A   T   30 Ind1_1 0/0 19 19  0
## 5:  Contig170_36 Contig170  36   A   G   30 Ind1_1 0/1 38 18 20
```

```
indSnps[, 1:3]
```

```
##                 LOCUS      CHROM POS
##     1:    Contig1_437    Contig1 437
##     2:   Contig34_213   Contig34 213
##     3:   Contig38_427   Contig38 427
##     4:  Contig152_389  Contig152 389
##     5:   Contig170_36  Contig170  36
##    ---
## 19996: Contig7274_197 Contig7274 197
## 19997:  Contig7287_10 Contig7287  10
## 19998:  Contig7291_92 Contig7291  92
## 19999: Contig7293_124 Contig7293 124
## 20000: Contig7299_279 Contig7299 279
```

```
indSnps[1:5, 1:3]
```

```
##            LOCUS     CHROM POS
## 1:   Contig1_437   Contig1 437
## 2:  Contig34_213  Contig34 213
## 3:  Contig38_427  Contig38 427
## 4: Contig152_389 Contig152 389
## 5:  Contig170_36 Contig170  36
```

```
indSnps[1:5, c('LOCUS','SAMPLE','GT')]
```

```
##            LOCUS SAMPLE  GT
## 1:   Contig1_437 Ind1_1 0/1
## 2:  Contig34_213 Ind1_1 0/1
## 3:  Contig38_427 Ind1_1 0/0
## 4: Contig152_389 Ind1_1 0/0
## 5:  Contig170_36 Ind1_1 0/1
```

But the neat thing about data tables is that we can use expressions to manipulate rows (at position `i`) and columns (at position `j`) using some sort of grouping (at position `by`). Here are some very simple examples:

```
# Subset rows to keep only those with a depth > 15.
indSnps[DP > 15,]
```

```
##                 LOCUS      CHROM POS REF ALT QUAL SAMPLE  GT DP RO AO
##     1:    Contig1_437    Contig1 437   T   A   30 Ind1_1 0/1 27 16 11
##     2:   Contig34_213   Contig34 213   A   G   30 Ind1_1 0/1 22  5 17
##     3:   Contig38_427   Contig38 427   G   A   30 Ind1_1 0/0 46 46  0
##     4:  Contig152_389  Contig152 389   A   T   30 Ind1_1 0/0 19 19  0
##     5:   Contig170_36  Contig170  36   A   G   30 Ind1_1 0/1 38 18 20
##    ---
## 19649: Contig7274_197 Contig7274 197   C   G   30 Ind4_9 0/0 47 47  0
## 19650:  Contig7287_10 Contig7287  10   C   T   30 Ind4_9 0/0 26 26  0
## 19651:  Contig7291_92 Contig7291  92   T   G   30 Ind4_9 0/0 23 23  0
## 19652: Contig7293_124 Contig7293 124   T   A   30 Ind4_9 1/1 26  0 26
## 19653: Contig7299_279 Contig7299 279   T   G   30 Ind4_9 0/0 43 43  0
```

```
# You can apply functions to columns, for example,
# take the mean depth across all samples and loci.
indSnps[, mean(DP)]
```

```
## [1] 34.9446
```

```
# We can add a grouping to our calculation of the mean. Here, we
# calculate the mean by locus.
indSnps[, mean(DP), by=LOCUS]
```

```
##               LOCUS    V1
##   1:    Contig1_437 35.87
##   2:   Contig34_213 34.08
##   3:   Contig38_427 34.18
##   4:  Contig152_389 34.76
##   5:   Contig170_36 35.17
##  ---
## 196: Contig7274_197 36.02
## 197:  Contig7287_10 35.98
## 198:  Contig7291_92 35.48
## 199: Contig7293_124 33.77
## 200: Contig7299_279 33.58
```

```
# We can combine manipulations of rows, columns and groups. Here, we
# filter for read depth > 30, then determine the number of samples
# with that read depth at each locus.
indSnps[DP > 30, length(unique(SAMPLE)), by=LOCUS]
```

```
##               LOCUS V1
##   1:   Contig38_427 63
##   2:   Contig170_36 66
##   3:  Contig183_250 66
##   4:  Contig346_278 62
##   5:  Contig425_300 72
##  ---
## 196:  Contig3113_10 67
## 197:  Contig3113_62 67
## 198: Contig5310_284 59
## 199: Contig6679_335 69
## 200:  Contig6505_32 61
```

```
# You can also use these feature of column manipulation to
# apply a function to the data and create a new column using
# the ':=' notation. For example, let's add a column the scores
# genotypes as an integer of counts of the alternate allele,
# as opposed to the VCF standard character format (0/0, 0/1, or 1/1).
# We will use the genomalicious function, genoscore_converter.
indSnps[, GT.INT:=genoscore_converter(GT)]
indSnps[1:5]
```

```
##            LOCUS     CHROM POS REF ALT QUAL SAMPLE  GT DP RO AO GT.INT
```

```
## 1:   Contig1_437   Contig1 437   T   A   30 Ind1_1 0/1 27 16 11      1
## 2:  Contig34_213  Contig34 213   A   G   30 Ind1_1 0/1 22  5 17      1
## 3:  Contig38_427  Contig38 427   G   A   30 Ind1_1 0/0 46 46  0      0
## 4: Contig152_389 Contig152 389   A   T   30 Ind1_1 0/0 19 19  0      0
## 5:  Contig170_36 Contig170  36   A   G   30 Ind1_1 0/1 38 18 20      1
```

Many functions in *genomalicious* take long-format `data.table` objects as their direct input. From personal experience, I find data tables from 100s of individuals, and 1,000s to 10,000s of SNP loci are manageable, though they may take a moment to load into R (especially when importing from a VCF). I believe the ease, simplicity, and versatility of working with data tables makes them a great way to deal with high dimensional data (many samples, many loci, many chromosomes/contigs).

As a disclaimer, it is important to note that the utility of the data table oriented methods developed in `genomalicious` will be limited by the dimensionality of your dataset and the memory and processing power of your system. Increasingly larger sample sizes and/or numbers of SNP loci will affect performance relative to the available computing resources. This is something to keep in mind, and very large genomic data sets will probably benefit from using other more memory efficient R packages specifically designed to handleenormous data sizes.

Data structures: Long-to-wide-format and genotype values

Though many functions in R expect data structured in long-format, there are lots of others that require data to be wide-format. This is especially the case in many population genetics/genomics R packages. Remember, in long-format, loci-by-sample combinations are all in rows, whereas in wide-format, samples are in rows and loci are in columns.

There are two functions in *genomalicious* for long-to-wide conversions: `DT2Mat_genos` for **individual genotypes** and `DT2Mat_freqs` for **allele frequencies**. Both return an R `matrix` object.

Let's first try this out with the individual-level genotype data we imported at the start of this lesson. Earlier, you made a column in `indSnps`, `$GT.INT`, that contained the genotypes scored as integer counts of the alternate alleles.

Take a look at the help file for `DT2Mat_genos`:

```
?DT2Mat_genos
```

```
## starting httpd help server ... done
```

You will see that `DT2Mat_genos` requires a data table as input and specification of the sample, locus, and genotype columns in the long format data table as the arguments, `popCol`, `locusCol`, and `genoCol`, respectively. The default values of these arguments are `popCol='SAMPLE', locusCol='LOCUS', genoCol='GT'`, but remember, `GT` in our data table `indSnps` records genotypes as **characters**, but we want to create a matrix of genotypes scored as **integers**. We therefore need to manually specify `genoCol`.

```
# Convert long-format data table of genotypes to a wide-format matrix
genosMat <- DT2Mat_genos(indSnps, genoCol='GT.INT')

# First 10 individuals and first 3 loci
genosMat[1:10,1:3]
```

```
##         Contig1014_62 Contig1047_30 Contig1074_118
## Ind1_1              0             1              2
## Ind1_10             0             0              2
```

```
## Ind1_11             0             0             2
## Ind1_12             0             0             2
## Ind1_13             0             0             2
## Ind1_14             0             1             1
## Ind1_15             0             0             1
## Ind1_16             0             1             2
## Ind1_17             0             2             2
## Ind1_18             0             1             2
```

```
# Check the class
class(genosMat)
```

```
## [1] "matrix" "array"
```

```
# Sample names are stored in the rows of the matrix
rownames(genosMat)
```

```
##   [1] "Ind1_1"  "Ind1_10" "Ind1_11" "Ind1_12" "Ind1_13" "Ind1_14" "Ind1_15"
##   [8] "Ind1_16" "Ind1_17" "Ind1_18" "Ind1_19" "Ind1_2"  "Ind1_20" "Ind1_21"
##  [15] "Ind1_22" "Ind1_23" "Ind1_24" "Ind1_25" "Ind1_3"  "Ind1_4"  "Ind1_5"
##  [22] "Ind1_6"  "Ind1_7"  "Ind1_8"  "Ind1_9"  "Ind2_1"  "Ind2_10" "Ind2_11"
##  [29] "Ind2_12" "Ind2_13" "Ind2_14" "Ind2_15" "Ind2_16" "Ind2_17" "Ind2_18"
##  [36] "Ind2_19" "Ind2_2"  "Ind2_20" "Ind2_21" "Ind2_22" "Ind2_23" "Ind2_24"
##  [43] "Ind2_25" "Ind2_3"  "Ind2_4"  "Ind2_5"  "Ind2_6"  "Ind2_7"  "Ind2_8"
##  [50] "Ind2_9"  "Ind3_1"  "Ind3_10" "Ind3_11" "Ind3_12" "Ind3_13" "Ind3_14"
##  [57] "Ind3_15" "Ind3_16" "Ind3_17" "Ind3_18" "Ind3_19" "Ind3_2"  "Ind3_20"
##  [64] "Ind3_21" "Ind3_22" "Ind3_23" "Ind3_24" "Ind3_25" "Ind3_3"  "Ind3_4"
##  [71] "Ind3_5"  "Ind3_6"  "Ind3_7"  "Ind3_8"  "Ind3_9"  "Ind4_1"  "Ind4_10"
##  [78] "Ind4_11" "Ind4_12" "Ind4_13" "Ind4_14" "Ind4_15" "Ind4_16" "Ind4_17"
##  [85] "Ind4_18" "Ind4_19" "Ind4_2"  "Ind4_20" "Ind4_21" "Ind4_22" "Ind4_23"
##  [92] "Ind4_24" "Ind4_25" "Ind4_3"  "Ind4_4"  "Ind4_5"  "Ind4_6"  "Ind4_7"
##  [99] "Ind4_8"  "Ind4_9"
```

```
# Loci names are stored in the columns of the matrix (the first 20)
colnames(genosMat)[1:20]
```

```
##  [1] "Contig1014_62"  "Contig1047_30"  "Contig1074_118" "Contig1078_331"
##  [5] "Contig1109_489" "Contig1132_452" "Contig1219_313" "Contig1323_488"
##  [9] "Contig1328_374" "Contig1330_380" "Contig1342_118" "Contig1344_183"
## [13] "Contig1354_315" "Contig1375_453" "Contig1384_494" "Contig1410_473"
## [17] "Contig1422_203" "Contig1462_463" "Contig152_389"  "Contig1541_121"
```

There is also a way to go in reverse, that is, to convert the wide-format matrix back into a long-format data table. This is done by specifying the argument `flip=TRUE` (default value is `FALSE`). Doing so requires the input to be a wide-format matrix. There are again default values of `popCol='SAMPLE', locusCol='LOCUS', genoCol='GT'`, which specify the long-format columns.

```
genosDT <- DT2Mat_genos(genosMat, flip=TRUE)

genosDT
```

```
##          SAMPLE         LOCUS GT
##     1:  Ind1_1 Contig1014_62  0
##     2: Ind1_10 Contig1014_62  0
##     3: Ind1_11 Contig1014_62  0
##     4: Ind1_12 Contig1014_62  0
##     5: Ind1_13 Contig1014_62  0
##    ---
## 19996:  Ind4_5 Contig984_413  1
## 19997:  Ind4_6 Contig984_413  0
## 19998:  Ind4_7 Contig984_413  0
## 19999:  Ind4_8 Contig984_413  0
## 20000:  Ind4_9 Contig984_413  0
```

Postamble

This concludes the 'Basic Ingredients' tutorial! You should now be comfortable with the basic functionality of *genomalicious* for importing SNP data into R. You have also familiarised yourself with SNP data structures and learnt how *genomalicious* can be used to do some basic manipulations that are common in population genetic/genomic analyses.