simuPOP Reference Manual

Release 1.0.5 (Rev: 3918)

Bo Peng

December 2004

Last modified
March 25, 2011

Department of Epidemiology, U.T. M.D. Anderson Cancer Center
Email: bpeng@mdanderson.org
URL: http://simupop.sourceforge.net
Mailing List: simupop-list@lists.sourceforge.net

Abstract

simuPOP is a general-purpose individual-based forward-time population genetics simulation environment. Unlike coalescent-based programs, simuPOP evolves populations forward in time, subject to arbitrary number of genetic and environmental forces such as mutation, recombination, migration and population/subpopulation size changes. In contrast to competing applications that use command-line options or configuration files to direct the execution of a limited number of predefined evolutionary scenarios, users of simuPOP's scripting interface could make use of many of its unique features, such as customized chromosome types, arbitrary nonrandom mating schemes, virtual subpopulations, information fields and Python operators, to construct and study almost arbitrarily complex evolutionary scenarios.

simuPOP is provided as a number of Python modules, which consist of a large number of Python objects and functions, including population, mating schemes, operators (objects that manipulate populations) and simulators to coordinate the evolutionary processes. It is the users' responsibility to write a Python script to glue these pieces together and form a simulation. At a more user-friendly level, an increasing number of functions and scripts contributed by simuPOP users is available in the online simuPOP cookbook. They provide useful functions for different applications (e.g. load and manipulate HapMap samples, import and export files from another application) and allow users who are unfamiliar with simuPOP to perform a large number of simulations ranging from basic population genetics models to generating datasets under complex evolutionary scenarios.

This document provides complete references to all classes and functions of simuPOP and its utility modules. Please refer to the *simuPOP user's guide* for a detailed introduction to simuPOP concepts, and a number of examples on how to use simuPOP to perform various simulations. All resources, including a pdf version of this guide and a mailing list can be found at the simuPOP homepage `http://simupop.sourceforge.net`.

How to cite simuPOP:

> Bo Peng and Marek Kimmel (2005) simuPOP: a forward-time population genetics simulation environment. *bioinformatics*, **21** (18): 3686-3687.
>
> Bo Peng and Christopher Amos (2008) Forward-time simulations of nonrandom mating populations using simuPOP. *bioinformatics*, **24** (11): 1408-1409.

Contents

Chapter 1

simuPOP Components

1.1 Individual, Population, pedigree and Simulator

1.1.1 Class `GenoStruTrait`

All individuals in a population share the same genotypic properties such as number of chromosomes, number and position of loci, names of markers, chromosomes, and information fields. These properties are stored in this `GenoStruTrait` class and are accessible from both `Individual` and `Population` classes. Currently, a genotypic structure consists of

- Ploidy, namely the number of homologous sets of chromosomes, of a population. Haplodiploid population is also supported.
- Number of chromosomes and number of loci on each chromosome.
- Positions of loci, which determine the relative distance between loci on the same chromosome. No unit is assumed so these positions can be ordinal (`1`, `2`, `3`, ..., the default), in physical distance (`bp`, `kb` or `mb`), or in map distance (e.g. `centiMorgan`) depending on applications.
- Names of alleles, which can either be shared by all loci or be specified for each locus.
- Names of loci and chromosomes.
- Names of information fields attached to each individual.

In addition to basic property access functions, this class provides some utility functions such as `locusByName`, which looks up a locus by its name.

class `GenoStruTrait()`

A `GenoStruTrait` object is created with the construction of a `Population` object and cannot be initialized directly.

`absLocusIndex(`*chrom, locus*`)`

Return the absolute index of locus *locus* on chromosome *chrom*. c.f. `chromLocusPair`.

`alleleName(`*allele, locus=0*`)`

Return the name of allele *allele* at *lcous* specified by the *alleleNames* parameter of the `Population` function. *locus* could be ignored if alleles at all loci share the same names. If the name of an allele is unspecified, its value (`'0'`, `'1'`, `'2'`, etc) is returned.

`alleleNames(`*locus=0*`)`

Return a list of allele names at given by the *alleleNames* parameter of the `Population` function. *locus* could be ignored if alleles at all loci share the same names. This list does not have to cover all possible allele states of a population so `alleleNames()[`*allele*`]` might fail (use `alleleNames(`*allele*`)` instead).

chromBegin(*chrom*)
Return the index of the first locus on chromosome *chrom*.

chromByName(*name*)
Return the index of a chromosome by its *name*.

chromEnd(*chrom*)
Return the index of the last locus on chromosome *chrom* plus 1.

chromLocusPair(*locus*)
Return the chromosome and relative index of a locus using its absolute index *locus*. c.f. `absLocusIndex`.

chromName(*chrom*)
Return the name of a chromosome *chrom*.

chromNames()
Return a list of the names of all chromosomes.

chromType(*chrom*)
Return the type of a chromosome *chrom* (`CUSTOMIZED`, `AUTOSOME`, `CHROMOSOME_X`, or `CHROMOSOME_Y`).

chromTypes()
Return the type of all chromosomes (`CUSTOMIZED`, `AUTOSOME`, `CHROMOSOME_X` or `CHROMOSOME_Y`).

infoField(*idx*)
Return the name of information field *idx*.

infoFields()
Return a list of the names of all information fields of the population.

infoIdx(*name*)
Return the index of information field *name*. Raise an `IndexError` if *name* is not one of the information fields.

lociByNames(*names*)
Return the indexes of loci with names *names*. Raise a `ValueError` if any of the loci cannot be found.

lociDist(*locus1, locus2*)
Return the distance between loci *locus1* and *locus2* on the same chromosome. A negative value will be returned if *locus1* is after *locus2*.

lociNames()
Return the names of all loci specified by the *lociNames* parameter of the `Population` function. An empty list will be returned if *lociNames* was not specified.

lociPos()
Return the positions of all loci, specified by the *lociPos* prameter of the `Population` function. The default positions are 1, 2, 3, 4, ... on each chromosome.

locusByName(*name*)
Return the index of a locus with name *name*. Raise a `ValueError` if no locus is found. Note that empty strings are used for loci without name but you cannot lookup such loci using this function.

locusName(*locus*)
Return the name of locus *locus* specified by the *lociNames* parameter of the `Population` function. An empty string will be returned if no name has been given to locus *locus*.

locusPos(*locus*)
Return the position of locus *locus* specified by the *lociPos* parameter of the `Population` function.

numChrom()
Return the number of chromosomes.

numLoci(*chrom*)
Return the number of loci on chromosome *chrom*.

numLoci()
Return a list of the number of loci on all chromosomes.

ploidy()
Return the number of homologous sets of chromosomes, specified by the *ploidy* parameter of the `Population` function. Return 2 for a haplodiploid population because two sets of chromosomes are stored for both males and females in such a population.

ploidyName()
Return the ploidy name of this population, can be one of `haploid`, `diploid`, `haplodiploid`, `triploid`, `tetraploid` or `#-ploid` where `#` is the ploidy number.

totNumLoci()
Return the total number of loci on all chromosomes.

1.1.2 Class `Individual`

A `Population` consists of individuals with the same genotypic structure. An `Individual` object cannot be created independently, but refences to inidividuals can be retrieved using member functions of a `Population` object. In addition to structural information shared by all individuals in a population (provided by class `GenoStruTrait`), the `Individual` class provides member functions to get and set *genotype*, *sex*, *affection status* and *information fields* of an individual.

Genotypes of an individual are stored sequentially and can be accessed locus by locus, or in batch. The alleles are arranged by position, chromosome and ploidy. That is to say, the first allele on the first chromosome of the first homologous set is followed by alleles at other loci on the same chromsome, then markers on the second and later chromosomes, followed by alleles on the second homologous set of the chromosomes for a diploid individual. A consequence of this memory layout is that alleles at the same locus of a non-haploid individual are separated by `Individual::totNumLoci()` loci. It is worth noting that access to invalid chromosomes, such as the Y chromosomes of female individuals, is not restricted.

class Individual()
An `Individual` object cannot be created directly. It has to be accessed from a `Population` object using functions such as `Population::Individual(idx)`.

affected()
Return `True` if this individual is affected.

allele(*idx, ploidy=-1, chrom=-1*)
Return the current allele at a locus, using its absolute index *idx*. If a ploidy *ploidy* and/or a chromosome indexes is given, *idx* is relative to the beginning of specified homologous copy of chromosomes (if *chrom=-1*) or the beginning of the specified homologous copy of specified chromosome (if *chrom* >= 0).

alleleChar(*idx, ploidy=-1, chrom=-1*)
Return the name of `allele(idx, ploidy, chrom)`. If idx is invalid (e.g. second homologus copy of chromosome Y), '_' is returned.

__cmp__(*rhs*)
A python function used to compare the individual objects

genotype(*ploidy=ALL_AVAIL, chroms=ALL_AVAIL*)
Return an editable array (a `carray` object) that represents all alleles of an individual. If *ploidy* or *chroms* is given, only alleles on the specified chromosomes and homologous copy of chromosomes will be returned. If multiple chromosomes are specified, there should not be gaps between chromosomes.

info(*field*)
Return the value of an information field *filed* (by index or name). `ind.info(name)` is equivalent to `ind.name` although the function form allows the use of indexes of information fieldes.

setAffected(*affected*)
Set affection status to *affected* (`True` or `False`).

setAllele(*allele, idx, ploidy=-1, chrom=-1*)
Set allele *allele* to a locus, using its absolute index *idx*. If a ploidy *ploidy* and/or a chromosome indexes

are given, *idx* is relative to the beginning of specified homologous copy of chromosomes (if *chrom=-1*) or the beginning of the specified homologous copy of specified chromosome (if *chrom* >= 0).

setGenotype(*geno, ploidy=ALL_AVAIL, chroms=ALL_AVAIL*)
Fill the genotype of an individual using a list of alleles *geno*. If parameters *ploidy* and/or *chroms* are specified, alleles will be copied to only all or specified chromosomes on selected homologous copies of chromosomes. `geno` will be reused if its length is less than number of alleles to be filled.

setInfo(*value, field*)
Set the value of an information field *field* (by index or name) to *value*. `ind.setInfo(value, field)` is equivalent to `ind.field = value` although the function form allows the use of indexes of information fieldes.

setSex(*sex*)
Set individual sex to `MALE` or `FEMALE`.

sex()
Return the sex of an individual, `1` for male and `2` for female.

1.1.3 Class `Population`

A simuPOP population consists of individuals of the same genotypic structure, organized by generations, subpopulations and virtual subpopulations. It also contains a Python dictionary that is used to store arbitrary population variables.

In addition to genotypic structured related functions provided by the `GenoStruTrait` class, the population class provides a large number of member functions that can be used to

- Create, copy and compare populations.
- Manipulate subpopulations. A population can be divided into several subpopulations. Because individuals only mate with individuals within the same subpopulation, exchange of genetic information across subpopulations can only be done through migration. A number of functions are provided to access subpopulation structure information, and to merge and split subpopulations.
- Define and access virtual subpopulations. A *virtual subpopulation splitter* can be assigned to a population, which defines groups of individuals called *virtual subpopulations* (VSP) within each subpopulation.
- Access individuals individually, or through iterators that iterate through individuals in (virtual) subpopulations.
- Access genotype and information fields of individuals at the population level. From a population point of view, all genotypes are arranged sequentially individual by individual. Please refer to class `Individual` for an introduction to genotype arragement of each individual.
- Store and access *ancestral generations*. A population can save arbitrary number of ancestral generations. It is possible to directly access an ancestor, or make an ancestral generation the current generation for more efficient access.
- Insert or remove loci, resize (shrink or expand) a population, sample from a population, or merge with other populations.
- Manipulate population variables and evaluate expressions in this *local namespace*.
- Save and load a population.

class **Population**(*size=[], ploidy=2, loci=[], chromTypes=[], lociPos=[], ancGen=0, chromNames=[], alleleNames=[], lociNames=[], subPopNames=[], infoFields=[]*)
The following parameters are used to create a population object:

size: A list of subpopulation sizes. The length of this list determines the number of subpopulations of this population. If there is no subpopulation, `size=[popSize]` can be written as `size=popSize`.

ploidy: Number of homologous sets of chromosomes. Default to `2` (diploid). For efficiency considerations, all chromosomes have the same number of homologous sets, even if some customized chromosomes or some individuals (e.g. males in a haplodiploid population) have different numbers of homologous sets. The first case is handled by setting *chromTypes* of each chromosome. Only the haplodiploid populations are handled for the second case, for which `ploidy=HAPLODIPLOID` should be used.

loci: A list of numbers of loci on each chromosome. The length of this parameter determines the number of chromosomes. If there is only one chromosome, `numLoci` instead of `[numLoci]` can be used.

chromTypes: A list that specifies the type of each chromosome, which can be `AUTOSOME`, `CHROMOSOME_X`, `CHROMOSOME_Y`, or `CUSTOMIZED`. All chromosomes are assumed to be autosomes if this parameter is ignored. Sex chromosome can only be specified in a diploid population where the sex of an individual is determined by the existence of these chromosomes using the `XX` (`FEMALE`) and `XY` (`MALE`) convention. Both sex chromosomes have to be available and be specified only once. Because chromosomes `X` and `Y` are treated as two chromosomes, recombination on the pseudo-autosomal regions of the sex chromsomes is not supported. `CUSTOMIZED` chromosomes are special chromosomes whose inheritance patterns are undefined. They rely on user-defined functions and operators to be passed from parents to offspring. Multiple customized chromosomes have to be arranged consecutively.

lociPos: Positions of all loci on all chromosome, as a list of float numbers. Default to `1`, `2`, ... etc on each chromosome. *lociPos* should be arranged chromosome by chromosome. If `lociPos` are not in order within a chromosome, they will be re-arranged along with corresponding *lociNames* (if specified).

ancGen: Number of the most recent ancestral generations to keep during evolution. Default to `0`, which means only the current generation will be kept. If it is set to `-1`, all ancestral generations will be kept in this population (and exhaust your computer RAM quickly).

chromNames: A list of chromosome names. Default to `''` for all chromosomes.

alleleNames: A list or a nested list of allele names. If a list of alleles is given, it will be used for all loci in this population. For example, `alleleNames=('A','C','T','G')` gives names `A`, `C`, `T`, and `G` to alleles `0`, `1`, `2`, and `3` respectively. If a nested list of names is given, it should specify alleles names for all loci.

lociNames: A list of names for each locus. It can be empty or a list of unique names for each locus. If loci are not specified in order, loci names will be rearranged according to their position on the chromosome.

subPopNames: A list of subpopulation names. All subpopulations will have name `''` if this parameter is not specified.

infoFields: Names of information fields (named float number) that will be attached to each individual.

absIndIndex(*idx, subPop*)
Return the absolute index of an individual *idx* in subpopulation *subPop*.

addChrom(*lociPos, lociNames=[], chromName="", alleleNames=[], chromType=AUTOSOME*)
Add chromosome *chromName* with given type *chromType* to a population, with loci *lociNames* inserted at position *lociPos*. *lociPos* should be ordered. *lociNames* and *chromName* should not exist in the current population. Allele names could be specified for all loci (a list of names) or differently for each locus (a nested list of names), using parameter *alleleNames*. Empty loci names will be used if *lociNames* is not specified.

addChromFrom(*pop*)
Add chromosomes in population *pop* to the current population. population *pop* should have the same number of individuals as the current population in the current and all ancestral generations. This function merges genotypes on the new chromosomes from population `pop` individual by individual.

addIndFrom(*pop*)
Add all individuals, including ancestors, in *pop* to the current population. Two populations should have the same genotypic structures and number of ancestral generations. Subpopulations in population *pop* are kept.

addInfoFields(*fields, init=0*)
Add a list of information fields *fields* to a population and initialize their values to *init*. If an information field alreay exists, it will be re-initialized.

addLoci(*chrom, pos, lociNames=[], alleleNames=[]*)

Insert loci *lociNames* at positions *pos* on chromosome *chrom*. These parameters should be lists of the same length, although *names* may be ignored, in which case empty strings will be assumed. Single-value input is allowed for parameter *chrom* and *pos* if only one locus is added. Alleles at inserted loci are initialized with zero alleles. Note that loci have to be added to existing chromosomes. If loci on a new chromosome need to be added, function `addChrom` should be used. Optionally, allele names could be specified either for all loci (a single list) or each loci (a nested list). This function returns indexes of the inserted loci.

addLociFrom(*pop*)

Add loci from population *pop*, chromosome by chromosome. Added loci will be inserted according to their position. Their position and names should not overlap with any locus in the current population. population *pop* should have the same number of individuals as the current population in the current and all ancestral generations.

ancestor(*idx, gen, subPop=[]*)

Return a reference to individual `idx` in ancestral generation `gen`. The correct individual will be returned even if the current generation is not the present one (see also `useAncestralGen`). If a valid *subPop* is specified, *index* is relative to that *subPop*. Virtual subpopulation is not supported. Note that a float *idx* is acceptable as long as it rounds closely to an integer.

ancestralGens()

Return the actual number of ancestral generations stored in a population, which does not necessarily equal to the number set by `setAncestralDepth()`.

clone()

Create a cloned copy of a population. Note that Python statement `pop1 = pop` only creates a reference to an existing population `pop`.

__cmp__(*rhs*)

A python function used to compare the population objects

dvars(*subPop=[]*)

Return a wrapper of Python dictionary returned by `vars(subPop)` so that dictionary keys can be accessed as attributes.

extractIndividuals(*indexes=[], IDs=[], idField="ind_id", filter=None*)

Extract individuals with given absolute indexes (parameter *indexes*), IDs (parameter *IDs*, stored in information field *idField*, default to `ind_id`), or a filter function (parameter *filter*). If a list of absolute indexes are specified, the present generation will be extracted and form a one-generational population. If a list of IDs are specified, this function will look through all ancestral generations and extract individuals with given ID. Individuals with shared IDs are allowed. In the last case, a user-defined Python function should be provided. This function should accept parameter `"ind"` or one or more of the information fields. All individuals, including ancestors if there are multiple ancestral generations, will be passed to this function. Individuals that returns `True` will be extracted. Extracted individuals will be in their original ancestral generations and subpopulations, even if some subpopulations or generations are empty. An `IndexError` will be raised if an index is out of bound but no error will be given if an invalid ID is encountered.

extractSubPops(*subPops=ALL_AVAIL, rearrange=False*)

Extract a list of (virtual) subpopulations from a population and create a new population. If *rearrange* is `False` (default), structure and names of extracted subpopulations are kept although extracted subpopulations can have fewer individuals if they are created from extracted virtual subpopulations. (e.g. it is possible to extract all male individuals from a subpopulation using a `SexSplitter()`). If *rearrange* is `True`, each (virtual) subpopulation in *subPops* becomes a new subpopulation in the extracted population in the order at which they are specified. Because each virtual subpopulation becomes a subpopulation, this function could be used, for example, to separate male and female individuals to two subpopulations (`subPops=[(0,0), (0,1)]`). If overlapping (virtual) subpopulations are specified, individuals will be copied multiple times. This function only extract individuals from the present generation.

genotype(*subPop=[]*)

Return an editable array of the genotype of all individuals in a population (if `subPop=[]`, default), or individuals in a subpopulation *subPop*. Virtual subpopulation is unsupported.

indByID(*id, ancGens=ALL_AVAIL, idField="ind_id"*)

Return a reference to individual with *id* stored in information field *idField* (default to `ind_id`). This function by default search the present and all ancestral generations (`ancGen=ALL_AVAIL`), but you can limit the search in specific generations if you know which generations to search (`ancGens=[0,1]` for present and parental generations) or `UNSPECIFIED` to search only the current generation. If no individual with *id* is found, an `IndexError` will be raised. A float *id* is acceptable as long as it rounds closely to an integer. Note that this function uses a dynamic searching algorithm which tends to be slow. If you need to look for multiple individuals from a static population, you might want to convert a population object to a pedigree object and use function `Pedigree.indByID`.

indInfo(*field, subPop=[]*)

Return the values (as a list) of information field `field` (by index or name) of all individuals (if `subPop=[]`, default), or individuals in a (virtual) subpopulation (if `subPop=sp` or `(sp, vsp)`).

individual(*idx, subPop=[]*)

Return a refernce to individual *idx* in the population (if `subPop=[]`, default) or a subpopulation (if `subPop=sp`). Virtual subpopulation is not supported. Note that a float *idx* is acceptable as long as it rounds closely to an integer.

individuals(*subPop=[]*)

Return an iterator that can be used to iterate through all individuals in a population (if `subPop=[]`, default), or a (virtual) subpopulation (`subPop=spID` or `(spID, vspID)`). If you would like to iterate through multiple subpopulations in multiple ancestral generations, please use function `Population.allIndividuals()`.

mergeSubPops(*subPops=ALL_AVAIL, name=""*)

Merge subpopulations *subPops*. If *subPops* is `ALL_AVAIL` (default), all subpopulations will be merged. *subPops* do not have to be adjacent to each other. They will all be merged to the subpopulation with the smallest subpopulation ID. Indexes of the rest of the subpopulation may be changed. A new name can be assigned to the merged subpopulation through parameter *name* (an empty *name* will be ignored). This function returns the ID of the merged subpopulation.

numSubPop()

Return the number of subpopulations in a population. Return 1 if there is no subpopulation structure.

numVirtualSubPop()

Return the number of virtual subpopulations (VSP) defined by a VSP splitter. Return `0` if no VSP is defined.

popSize(*ancGen=-1*)

Return the total number of individuals in all subpopulations of the current generation (default) or the an ancestral generation *ancGen*.

push(*pop*)

Push population *pop* into the current population. Both populations should have the same genotypic structure. The current population is discarded if *ancestralDepth* (maximum number of ancestral generations to hold) is zero so no ancestral generation can be kept. Otherise, the current population will become the parental generation of *pop*. If *ancGen* of a population is positive and there are already *ancGen* ancestral generations (c.f. `ancestralGens()`), the greatest ancestral generation will be discarded. In any case, Population*pop* becomes invalid as all its individuals are absorbed by the current population.

recodeAlleles(*alleles, loci=ALL_AVAIL, alleleNames=[]*)

Recode alleles at *loci* (can be a list of loci indexes or names, or all loci in a population (`ALL_AVAIL`)) to other values according to parameter *alleles*. This parameter can a list of new allele numbers for alleles `0`, `1`, `2`, ... (allele `x` will be recoded to `newAlleles[x]`, `x` outside of the range of *newAlleles* will not be recoded, although a warning will be given if `DBG_WARNING` is defined) or a Python function, which should accept one or both parameters `allele` (existing allele) and `locus` (index of locus). The return value will become the new allele. This function is intended to recode some alleles without listing all alleles in a list. It will be called once for each existing allele so it is not possible to recode an allele to different alleles. A new list of allele names could be specified for these *loci*. Different sets of names could be specified for each locus if a nested list of names are given. This function recode alleles for all subpopulations in all ancestral generations.

removeIndividuals(*indexes=[], IDs=[], idField="ind_id", filter=None*)

Remove individual(s) by absolute indexes (parameter *index*) or their IDs (parameter *IDs*), or using a filter function (paramter *filter*). If indexes are used, only individuals at the current generation will be removed. If IDs are used, all individuals with one of the IDs at information field *idField* (default to `"ind_id"`) will be removed. Although `"ind_id"` usually stores unique IDs of individuals, this function is frequently used to remove groups of individuals with the same value at an information field. An `IndexError` will be raised if an index is out of bound, but no error will be given if an invalid ID is specified. In the last case, a user-defined function should be provided. This function should accept parameter `"ind"` or one or more of the information fields. All individuals, including ancestors if there are multiple ancestral generations, will be passed to this function. Individuals that returns `True` will be removed. This function does not affect subpopulation structure in the sense that a subpopulation will be kept even if all individuals from it are removed.

removeInfoFields(*fields*)

Remove information fields *fields* from a population.

removeLoci(*loci=UNSPECIFIED, keep=UNSPECIFIED*)

Remove *loci* (absolute indexes or names) and genotypes at these loci from the current population. Alternatively, a parameter *keep* can be used to specify loci that will not be removed.

removeSubPops(*subPops*)

Remove (virtual) subpopulation(s) *subPops* and all their individuals. This function can be used to remove complete subpopulations (with shifted subpopulation indexes) or individuals belonging to virtual subpopulations of a subpopulation. In the latter case, the subpopulations are kept even if all individuals have been removed. This function only handles the present generation.

resize(*sizes, propagate=False*)

Resize population by giving new subpopulation sizes *sizes*. individuals at the end of some subpopulations will be removed if the new subpopulation size is smaller than the old one. New individuals will be appended to a subpopulation if the new size is larger. Their genotypes will be set to zero (default), or be copied from existing individuals if *propagate* is set to `True`. More specifically, if a subpopulation with `3` individuals is expanded to `7`, the added individuals will copy genotypes from individual `1`, `2`, `3`, and `1` respectively. Note that this function only resizes the current generation.

save(*filename*)

Save population to a file *filename*, which can be loaded by a global function `loadPopulation(filename)`.

setAncestralDepth(*depth*)

Set the intended ancestral depth of a population to *depth*, which can be `0` (does not store any ancestral generation), `-1` (store all ancestral generations), and a positive number (store *depth* ancestral generations. If there exists more than *depth* ancestral generations (if $depth > 0$), extra ancestral generations are removed.

setGenotype(*geno, subPop=[]*)

Fill the genotype of all individuals in a population (if `subPop=[]`) or in a (virtual) subpopulation *subPop* (if `subPop=sp` or `(sp, vsp)`) using a list of alleles *geno*. *geno* will be reused if its length is less than `subPopSize(subPop)*totNumLoci()*ploidy()`.

setIndInfo(*values, field, subPop=[]*)

Set information field `field` (specified by index or name) of all individuals (if `subPop=[]`, default), or individuals in a (virtual) subpopulation (`subPop=sp` or `(sp, vsp)`) to *values*. *values* will be reused if its length is smaller than the size of the population or (virtual) subpopulation.

setInfoFields(*fields, init=0*)

Set information fields *fields* to a population and initialize them with value *init*. All existing information fields will be removed.

setSubPopByIndInfo(*field*)

Rearrange individuals to their new subpopulations according to their integer values at information field *field* (value returned by `Individual::info(field)`). individuals with negative values at this *field* will be removed. Existing subpopulation names are kept. New subpopulations will have empty names.

setSubPopName(*name, subPop*)
Assign a name *name* to subpopulation *subPop*. Note that subpopulation names do not have to be unique.

setVirtualSplitter(*splitter*)
Set a VSP *splitter* to the population, which defines the same VSPs for all subpopulations. If different VSPs are needed for different subpopulations, a `CombinedSplitter` can be used to make these VSPs available to all subpopulations.

sortIndividuals(*infoFields*)
Sort individuals according to values at specified information fields (*infoFields*). Individuals will be sorted at an increasing order.

splitSubPop(*subPop, sizes, names=[]*)
Split subpopulation *subPop* into subpopulations of given *sizes*, which should add up to the size of subpopulation *subPop* or *1*, in which case *sizes* are treated as proportions. If *subPop* is not the last subpopulation, indexes of subpopulations after *subPop* are shifted. If *subPop* is named, the same name will be given to all new subpopulations unless a new set of *names* are specified for these subpopulations. This function returns the IDs of split subpopulations.

subPopBegin(*subPop*)
Return the index of the first individual in subpopulation *subPop*.

subPopByName(*name*)
Return the index of the first subpopulation with name *name*. An `IndexError` will be raised if subpopulations are not named, or if no subpopulation with name *name* is found. Virtual subpopulation name is not supported.

subPopEnd(*subPop*)
Return the index of the last individual in subpopulation *subPop* plus `1`, so that `range(subPopBegin(subPop), subPopEnd(subPop)` can iterate through the index of all individuals in subpopulation *subPop*.

subPopIndPair(*idx*)
Return the subpopulation ID and relative index of an individual, given its absolute index `idx`.

subPopName(*subPop*)
Return the "spName - vspName" (virtual named subpopulation), "" (unnamed non-virtual subpopulation), "spName" (named subpopulation) or "vspName" (unnamed virtual subpopulation), depending on whether subpopulation is named or if *subPop* is virtual.

subPopNames()
Return the names of all subpopulations (excluding virtual subpopulations). An empty string will be returned for unnamed subpopulations.

subPopSize(*subPop=[], ancGen=-1*)
Return the size of a subpopulation (`subPopSize(sp)`) or a virtual subpopulation (`subPopSize([sp, vsp])`) in the current generation (default) or a specified ancestral generation *ancGen*. If no *subpop* is given, it is the same as `popSize(ancGen)`. Population and virtual subpopulation names can be used.

subPopSizes(*ancGen=-1*)
Return the sizes of all subpopulations at the current generation (default) or specified ancestral generation *ancGen*. Virtual subpopulations are not considered.

swap(*rhs*)
Swap the content of two population objects, which can be handy in some particular circumstances. For example, you could swap out a population in a simulator.

updateInfoFieldsFrom(*fields, pop, fromFields=[], ancGens=ALL_AVAIL*)
Update information fields *fields* from *fromFields* of another population (or Pedigree) *pop*. Two populations should have the same number of individuals. If *fromFields* is not specified, it is assumed to be the same as *fields*. If *ancGens* is not `ALL_AVAIL`, only the specified ancestral generations are updated.

useAncestralGen(*idx*)
Making ancestral generation *idx* (`0` for current generation, `1` for parental generation, `2` for grand-parental generation, etc) the current generation. This is an efficient way to access Population properties of an

ancestral generation. `useAncestralGen(0)` should always be called afterward to restore the correct order of ancestral generations.

vars(*subPop=[]*)
Return variables of a population as a Python dictionary. If a valid subpopulation *subPop* is specified, a dictionary `vars()["subPop"][subPop]` is returned. A `ValueError` will be raised if key *subPop* does not exist in `vars()`, or if key *subPop* does not exist in `vars()["subPop"]`.

virtualSplitter()
Return the virtual splitter associated with the population, `None` will be returned if there is no splitter.

asPedigree(*idField='ind_id', fatherField='father_id', motherField='mother_id'*)
Convert the existing population object to a pedigree. After this function pedigree function should magically be usable for this function.

allIndividuals(*subPops=ALL_AVAIL, ancGens=True*)
Return an iterator that iterat through all (virtual) subpopulations in all ancestral generations. A list of (virtual) subpopulations (*subPops*) and a list of ancestral generations (*ancGens*, can be a single number) could be specified to iterate through only selected subpopulation and generations. Value "ALL_AVAIL" is acceptable in the specification of "sp" and/or "vsp" in specifying a virtual subpopulation "(sp, vsp)" for the iteration through all or specific virtual subpopulation in all or specific subpopulations.

evolve(*initOps=[], preOps=[], matingScheme=MatingScheme(), postOps=[], finalOps=[], gen=-1, dryrun=False*)
Evolve the current population *gen* generations using mating scheme *matingScheme* and operators *initOps* (applied before evolution), *preOps* (applied to the parental population at the beginning of each life cycle), *postOps* (applied to the offspring population at the end of each life cycle) and *finalOps* (applied at the end of evolution). More specifically, this function creates a *Simulator* using the current population, call its *evolve* function using passed parameters and then replace the current population with the evolved population. Please refer to function "Simulator.evolve" for more details about each parameter.

1.1.4 Class `Pedigree`

The pedigree class is derived from the population class. Unlike a population class that emphasizes on individual properties, the pedigree class emphasizes on relationship between individuals. An unique ID for all individuals is needed to create a pedigree object from a population object. Compared to the `Population` class, a `Pedigree` object is optimized for access individuals by their IDs, regardless of population structure and ancestral generations. Note that the implementation of some algorithms rely on the fact that parental IDs are smaller than their offspring because individual IDs are assigned sequentially during evolution. Pedigrees with manually assigned IDs should try to obey such a rule.

class Pedigree(*pop, loci=[], infoFields=[], ancGens=ALL_AVAIL, idField="ind_id", fatherField="father_id", motherField="mother_id", stealPop=False*)
Create a pedigree object from a population, using a subset of loci (parameter *loci*, can be a list of loci indexes, names, or `ALL_AVAIL`, default to no locus), information fields (parameter *infoFields*, default to no information field besides *idField*, *fatherField* and *motherField*), and ancestral generations (parameter *ancGens*, default to all ancestral generations). By default, information field `father_id` (parameter *fatherField*) and `mother_id` (parameter *motherField*) are used to locate parents identified by `ind_id` (parameter *idField*), which should store an unique ID for all individuals. Multiple individuls with the same ID are allowed and will be considered as the same individual, but a warning will be given if they actually differ in genotype or information fields. Operators `IdTagger` and `PedigreeTagger` are usually used to assign such IDs, although function `sampling.indexToID` could be used to assign unique IDs and construct parental IDs from index based relationship recorded by operator `ParentsTagger`. A pedigree object could be constructed with one or no parent but certain functions such as relative tracking will not be available for such pedigrees. In case that your are no longer using your population object, you could steal the content from the population by setting *stealPop* to `True`.

clone()
Create a cloned copy of a Pedigree.

identifyAncestors(*IDs=ALL_AVAIL, subPops=ALL_AVAIL, ancGens=ALL_AVAIL*)

If a list of individuals (*IDs*) is given, this function traces backward in time and find all ancestors of these individuals. If *IDs* is `ALL_AVAIL`, ancestors of all individuals in the present generation will be located. If a list of (virtual) subpopulations (*subPops*) or ancestral geneartions (*ancGens*) is given, the search will be limited to individuals in these subpopulations and generations. This could be used to, for example, find all fathers of *IDs*. This function returns a list of IDs, which includes valid specified IDs. Invalid IDs will be silently ignored. Note that parameters *subPops* and *ancGens* will limit starting IDs if `IDs` is set to `ALL_AVAIL`, but specified IDs will not be trimmed according to these parameters.

identifyFamilies(*pedField="", subPops=ALL_AVAIL, ancGens=ALL_AVAIL*)

This function goes through all individuals in a pedigree and group related individuals into families. If an information field *pedField* is given, indexes of families will be assigned to this field of each family member. The return value is a list of family sizes corresponding to families 0, 1, 2, ... etc. If a list of (virtual) subpopulations (parameter *subPops*) or ancestral generations are specified (parameter *ancGens*), the search will be limited to individuals in these subpopulations and generations.

identifyOffspring(*IDs=[], subPops=ALL_AVAIL, ancGens=ALL_AVAIL*)

This function traces forward in time and find all offspring of individuals specified in parameter *IDs*. If a list of (virtual) subpopulations (*subPops*) or ancestral geneartions (*ancGens*) is given, the search will be limited to individuals in these subpopulations and generations. This could be used to, for example, find all male offspring of *IDs*. This function returns a list of IDs, which includes valid starting *IDs*. Invalid IDs are silently ignored. Note that parameters *subPops* and *ancGens* will limit search result but will not be used to trim specified *IDs*.

indByID(*id*)

Return a reference to individual with *id*. An `IndexError` will be raised if no individual with *id* is found. An float *id* is acceptable as long as it rounds closely to an integer.

individualsWithRelatives(*infoFields, sex=[], affectionStatus=[], subPops=ALL_AVAIL, ancGens=ALL_AVAIL*)

Return a list of IDs of individuals who have non-negative values at information fields *infoFields*. Additional requirements could be specified by parameters *sex* and *affectionStatus*. *sex* can be `ANY_SEX` (default), `MALE_ONLY`, `FEMALE_ONLY`, `SAME_SEX` or `OPPOSITE_SEX`, and *affectionStatus* can be `AFFECTED`, `UNAFFECTED` or `ANY_AFFECTION_STATUS` (default). This function by default check all individuals in all ancestral generations, but you could limit the search using parameter *subPops* (a list of (virtual) subpopulations) and ancestral generations *ancGens*. Relatives fall out of specified subpopulations and ancestral generaions will be considered invalid.

locateRelatives(*relType, resultFields=[], sex=ANY_SEX, affectionStatus=ANY_AFFECTION_STATUS, ancGens=ALL_AVAIL*)

This function locates relatives (of type *relType*) of each individual and store their IDs in information fields *relFields*. The length of *relFields* determines how many relatives an individual can have.

Parameter *relType* specifies what type of relative to locate, which can be

- `SPOUSE` locate spouses with whom an individual has at least one common offspring.
- `OUTBRED_SPOUSE` locate non-slibling spouses, namely spouses with no shared parent.
- `OFFSPRING` all offspring of each individual.
- `COMMON_OFFSPRING` common offspring between each individual and its spouse (located by `SPOUSE` or `OUTBRED_SPOUSE`). *relFields* should consist of an information field for spouse and `m-1` fields for offspring where `m` is the number of fields.
- `FULLSIBLING` siblings with common father and mother,
- `SIBLING` siblings with at least one common parent.

Optionally, you can specify the sex and affection status of relatives you would like to locate, using parameters *sex* and *affectionStatus*. *sex* can be `ANY_SEX` (default), `MALE_ONLY`, `FEMALE_ONLY`, `SAME_SEX` or `OPPOSITE_SEX`, and *affectionStatus* can be `AFFECTED`, `UNAFFECTED` or `ANY_AFFECTION_STATUS` (default). Only relatives with specified properties will be located.

This function will by default go through all ancestral generations and locate relatives for all individuals. This can be changed by setting parameter *ancGens* to certain ancestral generations you would like to process.

save(*filename, infoFields=[], loci=[]*)

Save a pedigree to file *filename*. This function goes through all individuals of a pedigree and outputs in each line the ID of individual, IDs of his or her parents, sex (`'M'` or `'F'`), affection status (`'A'` or `'U'`), values of specified information fields *infoFields* and genotypes at specified loci (parameter `loci`, which can be a list of loci indexes, names, or `ALL_AVAIL`). Allele numbers, instead of their names are outputed. Two columns are used for each locus if the population is diploid. This file can be loaded using function `loadPedigree` although additional information such as names of information fields need to be specified. This format differs from a .ped file used in some genetic analysis software in that there is no family ID and IDs of all individuals have to be unique. Note that parental IDs will be set to zero if the parent is not in the pedigree object. Therefore, the parents of individuals in the top-most ancestral generation will always be zero.

traceRelatives(*fieldPath, sex=[], affectionStatus=[], resultFields=[], ancGens=ALL_AVAIL*)

Trace a relative path in a population and record the result in the given information fields *resultFields*. This function is used to locate more distant relatives based on the relatives located by function `locateRelatives`. For example, after siblings and offspring of all individuals are located, you can locate mother's sibling's offspring using a *relative path*, and save their indexes in each individuals information fields *resultFields*.

A *relative path* consits of a *fieldPath* that specifies which information fields to look for at each step, a *sex* specifies sex choices at each generation, and a *affectionStatus* that specifies affection status at each generation. *fieldPath* should be a list of information fields, *sex* and *affectionStatus* are optional. If specified, they should be a list of `ANY_SEX`, `MALE_ONLY`, `FEMALE_ONLY`, `SAME_SEX` and `OppsiteSex` for parameter *sex*, and a list of `UNAFFECTED`, `AFFECTED` and `ANY_AFFECTION_STATUS` for parameter *affectionStatus*.

For example, if `fieldPath = [['father_id', 'mother_id'], ['sib1', 'sib2'], ['off1', 'off2']]`, and `sex = [ANY_SEX, MALE_ONLY, FEMALE_ONLY]`, this function will locate `father_id` and `mother_id` for each individual, find all individuals referred by `father_id` and `mother_id`, find informaton fields `sib1` and `sib2` from these parents and locate male individuals referred by these two information fields. Finally, the information fields `off1` and `off2` from these siblings are located and are used to locate their female offspring. The results are father or mother's brother's daughters. Their indexes will be saved in each individuals information fields *resultFields*. If a list of ancestral generations is given in parameter *ancGens* is given, only individuals in these ancestral generations will be processed.

asPopulation()

Convert the existing pedigree object to a population. This function will behave like a regular population after this function call.

1.1.5 Class `Simulator`

A simuPOP simulator is responsible for evolving one or more populations forward in time, subject to various *operators*. Populations in a simulator are created from one or more replicates of specified populations. A number of functions are provided to access and manipulate populations, and most importantly, to evolve them.

class **Simulator**(*pops, rep=1, stealPops=True*)

Create a simulator with *rep* (default to `1`) replicates of populations *pops*, which is a list of populations although a single population object is also acceptable. Contents of passed populations are by default moved to the simulator to avoid duplication of potentially large population objects, leaving empty populations behind. This behavior can be changed by setting *stealPops* to `False`, in which case populations are copied to the simulator.

add(*pop, stealPop=True*)

Add a population *pop* to the end of an existing simulator. This function by default moves *pop* to the simulator, leaving an empty population for passed population object. If *steal* is set to `False`, the population will be copied to the simulator, and thus unchanged.

clone()

Clone a simulator, along with all its populations. Note that Python assign statement `simu1 = simu` only creates a symbolic link to an existing simulator.

__cmp__(*rhs*)
A Pyton function used to compare the simulator objects Note that mating schemes are not tested.

dvars(*rep, subPop=[]*)
Return a wrapper of Python dictionary returned by `vars(rep, subPop)` so that dictionary keys can be accessed as attributes.

evolve(*initOps=[], preOps=[], matingScheme=MatingScheme, postOps=[], finalOps=[], gen=-1, dryrun=False*)
Evolve all populations *gen* generations, subject to several lists of operators which are applied at different stages of an evolutionary process. Operators *initOps* are applied to all populations (subject to applicability restrictions of the operators, imposed by the *rep* parameter of these operators) before evolution. They are used to initialize populations before evolution. Operators *finalOps* are applied to all populations after the evolution.

Operators *preOps*, and *postOps* are applied during the life cycle of each generation. These operators can be applied at all or some of the generations, to all or some of the evolving populations, depending the *begin*, *end*, *step*, *at* and *reps* parameters of these operators. These operators are applied in the order at which they are specified. populations in a simulator are evolved one by one. At each generation, operators *preOps* are applied to the parental generations. A mating scheme is then used to populate an offspring generation. For each offspring, his or her sex is determined before during-mating operators of the mating scheme are used to transmit parental genotypes. After an offspring generation is successfully generated and becomes the current generation, operators *postOps* are applied to the offspring generation. If any of the *preOps* and *postOps* fails (return `False`), the evolution of a population will be stopped. The generation number of a population, which is the variable `"gen"` in each populations local namespace, is increased by one if an offspring generation has been successfully populated even if a post-mating operator fails. Another variable `"rep"` will also be set to indicate the index of each population in the simulator. Note that populations in a simulator does not have to have the same generation number. You could reset a population's generation number by changing this variable.

Parameter *gen* can be set to a positive number, which is the number of generations to evolve. If a simulator starts at the beginning of a generation `g` (for example 0), a simulator will stop at the beginning (instead of the end) of generation `g + gen` (for example gen). If *gen* is negative (default), the evolution will continue indefinitely, until all replicates are stopped by operators that return `False` at some point (these operators are called *terminators*). At the end of the evolution, the generations that each replicates have evolved are returned. Note that *finalOps* are applied to all applicable population, including those that have stopped before others.

If parameter *dryrun* is set to `True`, this function will print a description of the evolutionary process generated by function `describeEvolProcess()` and exits.

extract(*rep*)
Extract the *rep-th* population from a simulator. This will reduce the number of populations in this simulator by one.

numRep()
Return the number of replicates.

population(*rep*)
Return a reference to the *rep-th* population of a simulator. The reference will become invalid once the simulator starts evolving or becomes invalid (removed). If an independent copy of the population is needed, you can use `population.clone()` to create a cloned copy or `simulator.extract()` to remove the population from the simulator.

populations()
Return a Python iterator that can be used to iterate through all populations in a simulator.

vars(*rep, subPop=[]*)
Return the local namespace of the *rep-th* population, equivalent to `x.Population(rep).vars(subPop)`.

1.2 Virtual splitters

1.2.1 Class `BaseVspSplitter`

This class is the base class of all virtual subpopulation (VSP) splitters, which provide ways to define groups of individuals in a subpopulation who share certain properties. A splitter defines a fixed number of named VSPs. They do not have to add up to the whole subpopulation, nor do they have to be distinct. After a splitter is assigned to a population, many functions and operators can be applied to individuals within specified VSPs.

Each VSP has a name. A default name is determined by each splitter but you can also assign a name to each VSP. The name of a VSP can be retrieved by function `BaseVspSplitter.name()` or `Population.subPopName()`.

Only one VSP splitter can be assigned to a population, which defined VSPs for all its subpopulations. If different splitters are needed for different subpopulations, a `CombinedSplitter` can be used.

class `BaseVspSplitter`(*names=[]*)

This is a virtual class that cannot be instantiated.

`clone`()

All VSP splitter defines a `clone()` function to create an identical copy of itself.

`name`(*vsp*)

Return the name of VSP *vsp* (an index between `0` and `numVirtualSubPop()`).

`numVirtualSubPop`()

Return the number of VSPs defined by this splitter.

`vspByName`(*name*)

Return the index of a virtual subpopulation from its name. If multiple virtual subpopulations share the same name, the first vsp is returned.

1.2.2 Class `SexSplitter`

This splitter defines two VSPs by individual sex. The first VSP consists of all male individuals and the second VSP consists of all females in a subpopulation.

class `SexSplitter`(*names=[]*)

Create a sex splitter that defines male and female VSPs. These VSPs are named `Male` and `Female` unless a new set of names are specified by parameter *names*.

`name`(*vsp*)

Return `"Male"` if *vsp=0* and `"Female"` otherwise, unless a new set of names are specified.

`numVirtualSubPop`()

Return `2`.

1.2.3 Class `AffectionSplitter`

This class defines two VSPs according individual affection status. The first VSP consists of unaffected invidiauls and the second VSP consists of affected ones.

class `AffectionSplitter`(*names=[]*)

Create a splitter that defined two VSPs by affection status.These VSPs are named `Unaffected` and `Affected` unless a new set of names are specified by parameter *names*.

`name`(*vsp*)

Return `"Unaffected"` if *vsp=0* and `"Affected"` if *vsp=1*, unless a new set of names are specified.

`numVirtualSubPop`()

Return 2.

1.2.4 Class `InfoSplitter`

This splitter defines VSPs according to the value of an information field of each indivdiual. A VSP is defined either by a value or a range of values.

class `InfoSplitter`(*field, values=[], cutoff=[], ranges=[], names=[]*)

Create an infomration splitter using information field *field*. If parameter *values* is specified, each item in this list defines a VSP in which all individuals have this value at information field *field*. If a set of cutoff values are defined in parameter *cutoff*, individuals are grouped by intervals defined by these cutoff values. For example, `cutoff=[1,2]` defines three VSPs with `v < 1`, `1 <= v < 2` and `v >=2` where `v` is the value of an individual at information field *field*. If parameter `ranges` is specified, each range defines a VSP. For example, `ranges=[[1, 3], [2, 5]]` defines two VSPs with `1 <= v < 3` and `2 <= 3 < 5`. Of course, only one of the parameters *values*, *cutoff* and *ranges* should be defined, and values in *cutoff* should be distinct, and in an increasing order. A default set of names are given to each VSP unless a new set of names is given by parameter *names*.

name(*vsp*)

Return the name of a VSP *vsp*, which is `field = value` if VSPs are defined by values in parameter *values*, or `field < value` (the first VSP), `v1 <= field < v2` and `field >= v` (the last VSP) if VSPs are defined by cutoff values. A user-specified name, if specified, will be returned instead.

numVirtualSubPop()

Return the number of VSPs defined by this splitter, which is the length parameter *values* or the length of *cutoff* plus one, depending on which parameter is specified.

1.2.5 Class `ProportionSplitter`

This splitter divides subpopulations into several VSPs by proportion.

class `ProportionSplitter`(*proportions=[], names=[]*)

Create a splitter that divides subpopulations by *proportions*, which should be a list of float numbers (between `0` and `1`) that add up to `1`. A default set of names are given to each VSP unless a new set of names is given by parameter *names*.

name(*vsp*)

Return the name of VSP *vsp*, which is `"Prop p"` where `p=propotions[vsp]`. A user specified name will be returned if specified.

numVirtualSubPop()

Return the number of VSPs defined by this splitter, which is the length of parameter *proportions*.

1.2.6 Class `RangeSplitter`

This class defines a splitter that groups individuals in certain ranges into VSPs.

class `RangeSplitter`(*ranges, names=[]*)

Create a splitter according to a number of individual ranges defined in *ranges*. For example, `RangeSplitter(ranges=[[0, 20], [40, 50]])` defines two VSPs. The first VSP consists of individuals `0`, `1`, ..., `19`, and the sceond VSP consists of individuals `40`, `41`, ..., `49`. Note that a nested list has to be used even if only one range is defined. A default set of names are given to each VSP unless a new set of names is given by parameter *names*.

name(*vsp*)

Return the name of VSP *vsp*, which is `"Range [a, b)"` where `[a, b)` is range `ranges[vsp]`. A user specified name will be returned if specified.

numVirtualSubPop()

Return the number of VSPs, which is the number of ranges defined in parameter *ranges*.

1.2.7 Class `GenotypeSplitter`

This class defines a VSP splitter that defines VSPs according to individual genotype at specified loci.

class `GenotypeSplitter`(*loci, alleles, phase=False, names=[]*)

Create a splitter that defines VSPs by individual genotype at *loci* (can be indexes or names of one or more loci). Each list in a list *allele* defines a VSP, which is a list of allowed alleles at these *loci*. If only one VSP is defined, the outer list of the nested list can be ignored. If phase if true, the order of alleles in each list is significant. If more than one set of alleles are given, Individuals having either of them is qualified.

For example, in a haploid population, `loci=1, alleles=[0, 1]` defines a VSP with individuals having allele `0` or `1` at locus `1`, `alleles=[[0, 1], [2]]` defines two VSPs with indivdiuals in the second VSP having allele `2` at locus `1`. If multiple loci are involved, alleles at each locus need to be defined. For example, VSP defined by `loci=[0, 1], alleles=[0, 1, 1, 1]` consists of individuals having alleles `[0, 1]` or `[1, 1]` at loci `[0, 1]`.

In a haploid population, `loci=1, alleles=[0, 1]` defines a VSP with individuals having genotype `[0, 1]` or `[1, 0]` at locus `1`. `alleles[[0, 1], [2, 2]]` defines two VSPs with indivdiuals in the second VSP having genotype `[2, 2]` at locus `1`. If *phase* is set to `True`, the first VSP will only has individuals with genotype `[0, 1]`. In the multiple loci case, alleles should be arranged by haplotypes, for example, `loci=[0, 1], alleles=[0, 0, 1, 1], phase=True` defines a VSP with individuals having genotype `-0-0-, -1-1-` at loci `0` and `1`. If `phase=False` (default), genotypes `-1-1-, -0-0-`, `-0-1-` and `-1-0-` are all allowed.

A default set of names are given to each VSP unless a new set of names is given by parameter *names*.

`name`(*vsp*)

Return name of VSP *vsp*, which is `"Genotype loc1,loc2:genotype"` as defined by parameters *loci* and *alleles*. A user provided name will be returned if specified.

`numVirtualSubPop`()

Number of virtual subpops of subpopulation sp

1.2.8 Class `CombinedSplitter`

This splitter takes several splitters and stacks their VSPs together. For example, if the first splitter defines `3` VSPs and the second splitter defines `2`, the two VSPs from the second splitter become the fourth (index `3`) and the fifth (index `4`) VSPs of the combined splitter. In addition, a new set of VSPs could be defined as the union of one or more of the original VSPs. This splitter is usually used to define different types of VSPs to a population.

class `CombinedSplitter`(*splitters=[], vspMap=[], names=[]*)

Create a combined splitter using a list of *splitters*. For example, `CombinedSplitter([SexSplitter(), AffectionSplitter()])` defines a combined splitter with four VSPs, defined by male (vsp `0`), female (vsp `1`), unaffected (vsp `2`) and affected individuals (vsp `3`). Optionally, a new set of VSPs could be defined by parameter *vspMap*. Each item in this parameter is a list of VSPs that will be combined to a single VSP. For example, `vspMap=[(0, 2), (1, 3)]` in the previous example will define two VSPs defined by male or unaffected, and female or affected individuals. VSP names are usually determined by splitters, but can also be specified using parameter *names*.

`name`(*vsp*)

Return the name of a VSP *vsp*, which is the name a VSP defined by one of the combined splitters unless a new set of names is specified. If a *vspMap* was used, names from different VSPs will be joined by `"or"`.

`numVirtualSubPop`()

Return the number of VSPs defined by this splitter, which is the sum of the number of VSPs of all combined splitters.

1.2.9 Class `ProductSplitter`

This splitter takes several splitters and take their intersections as new VSPs. For example, if the first splitter defines `3`

VSPs and the second splitter defines 2, 6 VSPs will be defined by splitting 3 VSPs defined by the first splitter each to two VSPs. This splitter is usually used to define finer VSPs from existing VSPs.

class `ProductSplitter`(*splitters=[], names=[]*)

Create a product splitter using a list of *splitters*. For example, `ProductSplitter([SexSplitter(), AffectionSplitter()])` defines four VSPs by male unaffected, male affected, female unaffected, and female affected individuals. VSP names are usually determined by splitters, but can also be specified using parameter *names*.

`name`(*vsp*)

Return the name of a VSP *vsp*, which is the names of indivdual VSPs separated by a comma, unless a new set of names is specified for each VSP.

`numVirtualSubPop`()

Return the number of VSPs defined by this splitter, which is the sum of the number of VSPs of all combined splitters.

1.3 Mating Schemes

1.3.1 Class `MatingScheme`

This mating scheme is the base class of all mating schemes. It evolves a population generation by generation but does not actually transmit genotype.

class `MatingScheme`(*subPopSize=[]*)

Create a base mating scheme that evolves a population without transmitting genotypes. At each generation, this mating scheme creates an offspring generation according to parameter *subPopSize*, which can be a list of subpopulation sizes (or a number if there is only one subpopulation) or a Python function which will be called at each generation, just before mating, to determine the subpopulation sizes of the offspring generation. The function should be defined with one or both parameters of `gen` and `pop` where `gen` is the current generation number and `pop` is the parental population just before mating. The return value of this function should be a list of subpopulation sizes for the offspring generation. A single number can be returned if there is only one subpopulation. The passed parental population is usually used to determine offspring population size from parental population size but you can also modify this population to prepare for mating. A common practice is to split and merge parental populations in this function so that you demographic related information and actions could be implemented in the same function.

1.3.2 Class `HomoMating`

A homogeneous mating scheme that uses a parent chooser to choose parents from a prental generation, and an offspring generator to generate offspring from chosen parents. It can be either used directly, or within a heterogeneous mating scheme. In the latter case, it can be applied to a (virtual) subpopulation.

class `HomoMating`(*chooser, generator, subPopSize=[], subPops=ALL_AVAIL, weight=0*)

Create a homogeneous mating scheme using a parent chooser *chooser* and an offspring generator *generator*.

If this mating scheme is used directly in a simulator, it will be responsible for creating an offspring population according to parameter *subPopSize*. This parameter can be a list of subpopulation sizes (or a number if there is only one subpopulation) or a Python function which will be called at each generation to determine the subpopulation sizes of the offspring generation. Please refer to class `MatingScheme` for details about this parameter.

If this mating shcme is used within a heterogeneous mating scheme. Parameters *subPops* and *weight* are used to determine which (virtual) subpopulations this mating scheme will be applied to, and how many offspring this mating scheme will produce. Please refer to mating scheme `HeteroMating` for the use of these two parameters.

1.3.3 Class `HeteroMating`

A heterogeneous mating scheme that applies a list of homogeneous mating schemes to different (virtual) subpopulations.

class `HeteroMating`(*matingSchemes, subPopSize=[], shuffleOffspring=True*)

Create a heterogeneous mating scheme that will apply a list of homogeneous mating schemes *matingSchemes* to different (virtual) subpopulations. The size of the offspring generation is determined by parameter *subPopSize*, which can be a list of subpopulation sizes or a Python function that returns a list of subpopulation sizes at each generation. Please refer to class `MatingScheme` for a detailed explanation of this parameter.

Each mating scheme defined in *matingSchemes* can be applied to one or more (virtual) subpopulation. If parameter *subPops* is not specified, a mating scheme will be applied to all subpopulations. If a list of (virtual) subpopulation is specified, the mating scheme will be applied to specific (virtual) subpopulations.

If multiple mating schemes are applied to the same subpopulation, a weight (parameter *weight*) can be given to each mating scheme to determine how many offspring it will produce. The default for all mating schemes are `0`. In this case, the number of offspring each mating scheme produces is proportional to the size of its parental (virtual) subpopulation. If all weights are negative, the numbers of offspring are determined by the multiplication of the absolute values of the weights and their respective parental (virtual) subpopulation sizes. If all weights are positive, the number of offspring produced by each mating scheme is proportional to these weights. Mating schemes with zero weight in this case will produce no offspring. If both negative and positive weights are present, negative weights are processed before positive ones.

If multiple mating schemes are applied to the same subpopulation, offspring produced by these mating schemes are shuffled randomly. If this is not desired, you can turn off offspring shuffling by setting parameter *shuffleOffspring* to `False`.

1.3.4 Class `PedigreeMating`

This mating scheme evolves a population following an existing pedigree structure. If the `Pedigree` object has `N` ancestral generations and a present generation, it can be used to evolve a population for `N` generations, starting from the topmost ancestral generation. At the *k-th* generation, this mating scheme produces an offspring generation according to subpopulation structure of the `N-k-1` ancestral generation in the pedigree object (e.g. producing the offspring population of generation 0 according to the `N-1` ancestral generation of the pedigree object). For each offspring, this mating scheme copies individual ID and sex from the corresponing individual in the pedigree object. It then locates the parents of each offspring using their IDs in the pedigree object. A list of during mating operators are then used to transmit parental genotype to the offspring. The population being evolved must have an information field `'ind_id'`.

class `PedigreeMating`(*ped, ops, idField="ind_id"*)

Creates a pedigree mating scheme that evolves a population according to `Pedigree` object *ped*. The evolved population should contain individuals with ID (at information field *idField*, default to `'ind_id'`) that match those individual in the topmost ancestral generation who have offspring. After parents of each individuals are determined from their IDs, a list of during-mating operators *ops* are applied to transmit genotypes. The return value of these operators are not checked.

1.3.5 Class `SequentialParentChooser`

This parent chooser chooses a parent from a parental (virtual) subpopulation sequentially. Natural selection is not considered. If the last parent is reached, this parent chooser will restart from the beginning of the (virtual) subpopulation.

class `SequentialParentChooser`(*sexChoice=ANY_SEX*)

Create a parent chooser that chooses a parent from a parental (virtual) subpopulation sequentially. Parameter *choice* can be `ANY_SEX` (default), `MALE_ONLY` and `FEMALE_ONLY`. In the latter two cases, only male or female individuals are selected. A `RuntimeError` will be raised if there is no male or female individual from the population.

1.3.6 Class `SequentialParentsChooser`

This parent chooser chooses two parents (a father and a mother) sequentially from their respective sex groups. Selection is not considered. If all fathers (or mothers) are exhausted, this parent chooser will choose fathers (or mothers) from the beginning of the (virtual) subpopulation again.

class `SequentialParentsChooser`()

Create a parent chooser that chooses two parents sequentially from a parental (virtual) subpopulation.

1.3.7 Class `RandomParentChooser`

This parent chooser chooses a parent randomly from a (virtual) parental subpopulation. Parents are chosen with or without replacement. If parents are chosen with replacement, a parent can be selected multiple times. If individual fitness values are assigned to individuals (stored in an information field *selectionField* (default to `"fitness"`), individuals will be chosen at a probability proportional to his or her fitness value. If parents are chosen without replacement, a parent can be chosen only once. An `RuntimeError` will be raised if all parents are exhausted. Natural selection is disabled in the without-replacement case.

class `RandomParentChooser`(*replacement=True, selectionField="fitness", sexChoice=ANY_SEX*)

Create a random parent chooser that choose parents with or without replacement (parameter *replacement*, default to `True`). If selection is enabled and information field *selectionField* exists in the passed population, the probability that a parent is chosen is proportional to his/her fitness value stored in *selectionField*. This parent chooser by default chooses parent from all individuals (`ANY_SEX`), but it can be made to select only male (`MALE_ONLY`) or female (`FEMALE_ONLY`) individuals by setting parameter *sexChoice*.

1.3.8 Class `RandomParentsChooser`

This parent chooser chooses two parents, a male and a female, randomly from a (virtual) parental subpopulation. Parents are chosen with or without replacement from their respective sex group. If parents are chosen with replacement, a parent can be selected multiple times. If individual fitness values are assigned (stored in information field *selectionField*, default to `"fitness"`, the probability that an individual is chosen is proportional to his/her fitness value among all individuals with the same sex. If parents are chosen without replacement, a parent can be chosen only once. An `RuntimeError` will be raised if all males or females are exhausted. Natural selection is disabled in the without-replacement case.

class `RandomParentsChooser`(*replacement=True, selectionField="fitness"*)

Create a random parents chooser that choose two parents with or without replacement (parameter *replacement*, default to `True`). If selection is enabled and information field *selectionField* exists in the passed population, the probability that a parent is chosen is proportional to his/her fitness value stored in *selectionField*.

1.3.9 Class `PolyParentsChooser`

This parent chooser is similar to random parents chooser but instead of selecting a new pair of parents each time, one of the parents in this parent chooser will mate with several spouses before he/she is replaced. This mimicks multi-spouse mating schemes such as polygyny or polyandry in some populations. Natural selection is supported for both sexes.

class `PolyParentsChooser`(*polySex=MALE, polyNum=1, selectionField="fitness"*)

Create a multi-spouse parents chooser where each father (if *polySex* is MALE) or mother (if *polySex* is FEMALE) has *polyNum* spouses. The parents are chosen with replacement. If individual fitness values are assigned (stored to information field `selectionField`, default to `"fitness"`), the probability that an individual is chosen is proportional to his/her fitness value among all individuals with the same sex.

1.3.10 Class `CombinedParentsChooser`

This parent chooser accepts two parent choosers. It takes one parent from each parent chooser and return them as father and mother. Because two parent choosers do not have to choose parents from the same virtual subpopulation, this parent chooser allows you to choose parents from different subpopulations.

class `CombinedParentsChooser`(*fatherChooser, motherChooser*)

Create a Python parent chooser using two parent choosers *fatherChooser* and *motherChooser*. It takes one parent from each parent chooser and return them as father and mother. If two valid parents are returned, the first valid parent (father) will be used for *fatherChooser*, the second valid parent (mother) will be used for *motherChooser*. Although these two parent choosers are supposed to return a father and a mother respectively, the sex of returned parents are not checked so it is possible to return parents with the same sex using this parents chooser.

1.3.11 Class `PyParentsChooser`

This parent chooser accepts a Python generator function that repeatedly yields one or two parents, which can be references to individual objects or indexes relative to each subpopulation. The parent chooser calls the generator function with parental population and a subpopulation index for each subpopulation and retrieves parents repeatedly using the iterator interface of the generator function.

This parent chooser does not support virtual subpopulation directly. However, because virtual subpopulations are defined in the passed parental population, it is easy to return parents from a particular virtual subpopulation using virtual subpopulation related functions.

class `PyParentsChooser`(*generator*)

Create a Python parent chooser using a Python generator function *parentsGenerator*. This function should accept one or both of parameters *pop* (the parental population) and *subPop* (index of subpopulation) and return the reference or index (relative to subpopulation) of a parent or a pair of parents repeatedly using the iterator interface of the generator function.

1.3.12 Class `OffspringGenerator`

An *offspring generator* generates offspring from parents chosen by a parent chooser. It is responsible for creating a certain number of offspring, determinning their sex, and transmitting genotypes from parents to offspring.

class `OffspringGenerator`(*ops, numOffspring=1, sexMode=RANDOM_SEX*)

Create a basic offspring generator. This offspring generator uses *ops* genotype transmitters to transmit genotypes from parents to offspring.

A number of *during-mating operators* (parameter *ops*) can be used to, among other possible duties such as setting information fields of offspring, transmit genotype from parents to offspring. This general offspring generator does not have any default during-mating operator but all stock mating schemes use an offspring generator with a default operator. For example, a `mendelianOffspringGenerator` is used by `RandomMating` to trasmit genotypes. Note that applicability parameters `begin`, `step`, `end`, `at` and `reps` could be used in these operators but negative population and generation indexes are unsupported.

Parameter *numOffspring* is used to control the number of offspring per mating event, or in another word the number of offspring in each family. It can be a number, a Python function or generator, or a mode parameter followed by some optional arguments. If a number is given, given number of offspring will be generated at each mating event. If a Python function is given, it will be called each time when a mating event happens. When a generator function is specified, it will be called for each subpopulation to provide number of offspring for all mating events during the populating of this subpopulation. Current generation number will be passed to this function or generator function if parameter "gen" is used in this function. In the last case, a tuple (or a list) in one of the following forms can be given:

- `(GEOMETRIC_DISTRIBUTION, p)`
- `(POISSON_DISTRIBUTION, p)`, $p > 0$

- `(BINOMIAL_DISTRIBUTION, p, N)`, $0 < p <= 1$, $N > 0$
- `(UNIFORM_DISTRIBUTION, a, b)`, $0 <= a <= b$.

In this case, the number of offspring will be determined randomly following the specified statistical distributions. Because families with zero offspring are silently ignored, the distribution of the observed number of offspring per mating event (excluding zero) follows zero-truncated versions of these distributions.

Parameter *numOffspring* specifies the number of offspring per mating event but the actual surviving offspring can be less than specified. More spefically, if any during-mating operator returns `False`, an offspring will be discarded so the actually number of offspring of a mating event will be reduced. This is essentially how during-mating selector works.

Parameter *sexMode* is used to control the sex of each offspring. Its default value is usually *RANDOM_SEX* which assign `MALE` or `FEMALE` to each individual randomly, with equal probabilities. If `NO_SEX` is given, offspring sex will not be changed. *sexMode* can also be one of

- `(PROB_OF_MALES, p)` where `p` is the probability of male for each offspring,
- `(NUM_OF_MALES, n)` where `n` is the number of males in a mating event. If `n` is greater than or equal to the number of offspring in this family, all offspring in this family will be `MALE`.
- `(NUM_OF_FEMALES, n)` where `n` is the number of females in a mating event,
- `(SEQUENCE_OF_SEX, s1, s2 ...)` where `s1`, `s2` etc are MALE or FEMALE. The sequence will be used for each mating event. It will be reused if the number of offspring in a mating event is greater than the length of sequence.
- `(GLOBAL_SEQUENCE_OF_SEX, s1, s2, ...)` where `s1`, `s2` etc are MALE or FEMALE. The sequence will be used across mating events. It will be reused if the number of offspring in a subpopulation is greater than the length of sequence.

Finally, parameter *sexMode* accepts a function or a generator function. A function will be called whenever an offspring is produced. A generator will be created at each subpopulation and will be used to produce sex for all offspring in this subpopulation. No parameter is accepted.

1.3.13 Class `ControlledOffspringGenerator`

This offspring generator populates an offspring population and controls allele frequencies at specified loci. At each generation, expected allele frequencies at these loci are passed from a user defined allele frequency *trajectory* function. The offspring population is populated in two steps. At the first step, only families with disease alleles are accepted until until the expected number of disease alleles are met. At the second step, only families with wide type alleles are accepted to populate the rest of the offspring generation. This method is described in detail in "Peng et al, (2007) PLoS Genetics".

class `ControlledOffspringGenerator`(*loci, alleles, freqFunc, ops=[], numOffspring=1, sexMode=RANDOM_SEX*)

Create an offspring generator that selects offspring so that allele frequency at specified loci in the offspring generation reaches specified allele frequency. At the beginning of each generation, expected allele frequency of *alleles* at *loci* is returned from a user-defined trajectory function *freqFunc*. Parameter *loci* can be a list of loci indexes, names, or ALL_AVAIL. If there is no subpopulation, this function should return a list of frequencies for each locus. If there are multiple subpopulations, *freqFunc* can return a list of allele frequencies for all subpopulations or combined frequencies that ignore population structure. In the former case, allele frequencies should be arranged by loc0_sp0, loc1_sp0, ... loc0_sp1, loc1_sp1, ..., and so on. In the latter case, overall expected number of alleles are scattered to each subpopulation in proportion to existing number of alleles in each subpopulation, using a multinomial distribution.

After the expected alleles are calculated, this offspring generator accept and reject families according to their genotype at *loci* until allele frequecies reach their expected values. The rest of the offspring generation is then filled with families without only wild type alleles at these *loci*.

This offspring generator is derived from class *OffspringGenerator*. Please refer to class *OffspringGenerator* for a detailed description of parameters *ops*, *numOffspring* and *sexMode*.

1.4 Pre-defined mating schemes

1.4.1 Class `CloneMating`

A homogeneous mating scheme that uses a sequential parent chooser and a clone offspring generator.

class `CloneMating`(*numOffspring=1, sexMode=None, ops=CloneGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField=None*)

Create a clonal mating scheme that clones parents to offspring using a `CloneGenoTransmitter`. Please refer to class `OffspringGenerator` for parameters *ops* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*. Parameters *sexMode* and *selectionField* are ignored because this mating scheme does not support natural selection, and `CloneGenoTransmitter` copies sex from parents to offspring. Note that `CloneGenoTransmitter` by default also copies all parental information fields to offspring.

1.4.2 Class `RandomSelection`

A homogeneous mating scheme that uses a random single-parent parent chooser with replacement, and a clone offspring generator. This mating scheme is usually used to simulate the basic haploid Wright-Fisher model but it can also be applied to diploid populations.

class `RandomSelection`(*numOffspring=1, sexMode=None, ops=CloneGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Create a mating scheme that select a parent randomly and copy him or her to the offspring population. Please refer to class `RandomParentChooser` for parameter *selectionField*, to class `OffspringGenerator` for parameters *ops* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*. Parameter *sexMode* is ignored because `cloneOffspringGenerator` copies sex from parents to offspring.

1.4.3 Class `RandomMating`

A homogeneous mating scheme that uses a random parents chooser with replacement and a Mendelian offspring generator. This mating scheme is widely used to simulate diploid sexual Wright-Fisher random mating.

class `RandomMating`(*numOffspring=1, sexMode=RANDOM_SEX, ops=MendelianGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Creates a random mating ssheme that selects two parents randomly and transmit genotypes according to Mendelian laws. Please refer to class `RandomParentsChooser` for parameter *selectionField*, to class `OffspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*.

1.4.4 Class `MonogamousMating`

A homogeneous mating scheme that uses a random parents chooser without replacement and a Mendelian offspring generator. It differs from the basic random mating scheme in that each parent can mate only once so there is no half-sibling in the population.

class `MonogamousMating`(*numOffspring=1, sexMode=RANDOM_SEX, ops=MendelianGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField=None*)

Creates a monogamous mating scheme that selects each parent only once. Please refer to class `OffspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*. Parameter *selectionField* is ignored because this mating scheme does not support natural selection.

1.4.5 Class `PolygamousMating`

A homogeneous mating scheme that uses a multi-spouse parents chooser and a Mendelian offspring generator. It differs from the basic random mating scheme in that each parent of sex *polySex* will have *polyNum* spouses.

class `PolygamousMating`(*polySex=MALE, polyNum=1, numOffspring=1, sexMode=RANDOM_SEX, ops=MendelianGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Creates a polygamous mating scheme that each parent mates with multiple spouses. Please refer to class `PolyParentsChooser` for parameters *polySex*, *polyNum* and *selectionField*, to class `OffspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*.

1.4.6 Class `HaplodiploidMating`

A homogeneous mating scheme that uses a random parents chooser with replacement and a haplodiploid offspring generator. It should be used in a haplodiploid population where male individuals only have one set of homologous chromosomes.

class `HaplodiploidMating`(*numOffspring=1.0, sexMode=RANDOM_SEX, ops=HaplodiploidGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Creates a mating scheme in haplodiploid populations. Please refer to class `RandomParentsChooser` for parameter *selectionField*, to class `OffspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*.

1.4.7 Class `SelfMating`

A homogeneous mating scheme that uses a random single-parent parent chooser with or without replacement (parameter *replacement*) and a selfing offspring generator. It is used to mimic self-fertilization in certain plant populations.

class `SelfMating`(*replacement=True, numOffspring=1, sexMode=RANDOM_SEX, ops=SelfingGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Creates a selfing mating scheme where two homologous copies of parental chromosomes are transmitted to offspring according to Mendelian laws. Please refer to class `RandomParentChooser` for parameter *replacement* and *selectionField*, to class `OffspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*.

1.4.8 Class `ControlledRandomMating`

A homogeneous mating scheme that uses a random sexual parents chooser with replacement and a controlled offspring generator using Mendelian genotype transmitter. It falls back to a regular random mating scheme if there is no locus to control or no trajectory is defined.

class `ControlledRandomMating`(*loci=[], alleles=[], freqFunc=None, numOffspring=1, sexMode=RANDOM_SEX, ops=MendelianGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Creates a random mating scheme that controls allele frequency at loci *loci*. At each generation, function *freqFunc* will be called to called to obtain intended frequencies of alleles *alleles* at loci *loci*. The controlled offspring generator will control the acceptance of offspring so that the generation reaches desired allele frequencies at these loci. If *loci* is empty or *freqFunc* is `None`, this mating scheme works identically to a `RandomMating scheme`. Rationals and applications of this mating scheme is described in details in a paper *Peng et al, 2007 (PLoS Genetics)*. Please refer to class `RandomParentsChooser` for parameters *selectionField*, to class `ControlledOffspringGenerator` for parameters *loci*, *alleles*, *freqFunc*, to class `OffspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*.

1.5 Utility Classes

1.5.1 Class `WithArgs`

This class wraps around a user-provided function and provides an attribute `args` so that simuPOP knows which parameters to send to the function. This is only needed if the function can not be defined with allowed parameters.

class `WithArgs`(*func, args*)

Return a callable object that wraps around function `func`. Parameter `args` should be a list of parameter names.

1.5.2 Class `RNG`

This random number generator class wraps around a number of random number generators from GNU Scientific Library. You can obtain and change the RNG used by the current simuPOP module through the `getRNG()` function, or create a separate random number generator and use it in your script.

class `RNG`(*name=None, seed=0*)

Create a RNG object using specified name and seed. If *rng* is not given, environmental variable `GSL_RNG_TYPE` will be used if it is available. Otherwise, generator `mt19937` will be used. If *seed* is not given, `/dev/urandom`, `/dev/random`, or other system random number source will be used to guarantee that random seeds are used even if more than one simuPOP sessions are started simultaneously. Names of supported random number generators are available from `moduleInfo()['availableRNGs']`.

`name``()`

Return the name of the current random number generator.

`randBinomial`(*n, p*)

Generate a random number following a binomial distribution with parameters *n* and *p*.

`randChisq`(*nu*)

Generate a random number following a Chi-squared distribution with *nu* degrees of freedom.

`randExponential`(*mu*)

Generate a random number following a exponential distribution with parameter *mu*.

`randGamma`(*a, b*)

Generate a random number following a gamma distribution with a shape parameters *a* and scale parameter *b*.

`randGeometric`(*p*)

Generate a random number following a geometric distribution with parameter *p*.

`randInt`(*n*)

Return a random number in the range of `[0, 1, 2, ... n-1]`

`randMultinomial`(*N, p*)

Generate a random number following a multinomial distribution with parameters *N* and *p* (a list of probabilities).

`randNormal`(*mu, sigma*)

Generate a random number following a normal distribution with mean *mu* and standard deviation *sigma*.

`randPoisson`(*mu*)

Generate a random number following a Poisson distribution with parameter *mu*.

`randTruncatedBinomial`(*n, p*)

Generate a positive random number following a zero-truncated binomial distribution with parameters *n* and *p*.

`randTruncatedPoisson`(*mu*)

Generate a positive random number following a zero-truncated Poisson distribution with parameter *mu*.

randUniform()
Generate a random number following a rng_uniform [0, 1) distribution.

seed()
Return the seed used to initialize the RNG. This can be used to repeat a previous session.

set(*name=None, seed=0*)
Replace the existing random number generator using RNG*name* with seed *seed*. If *seed* is 0, a random seed will be used. If *name* is empty, use the existing RNG but reset the seed.

1.5.3 Class `WeightedSampler`

A random number generator that returns `0`, `1`, ..., `k-1` with probabilites that are proportional to their weights. For example, a weighted sampler with weights `4`, `3`, `2` and `1` will return numbers `0`, `1`, `2` and `3` with probabilities `0.4`, `0.3`, `0.2` and `0.1`, respectively. If an additional parameter `N` is specified, the weighted sampler will return exact proportions of numbers if `N` numbers are returned. The version without additional parameter is similar to the `sample(prob, replace=FALSE)` function of the R statistical package.

class **WeightedSampler**(*weights=[], N=0*)
Creates a weighted sampler that returns `0`, `1`, ... `k-1` when a list of `k` weights are specified (*weights*). *weights* do not have to add up to 1. If a non-zero *N* is specified, exact proportions of numbers will be returned in *N* returned numbers.

draw()
Returns a random number between `0` and `k-1` with probabilities that are proportional to specified weights.

drawSamples(*n=1*)
Returns a list of *n* random numbers

1.6 Global functions

1.6.1 Function `closeOutput`

closeOutput(*output=""*)
Output files specified by `'>'` are closed immediately after they are written. Those specified by `'>>'` and `'>>>'` are closed by a simulator after `Simulator.evolve()`. However, these files will be kept open if the operators are applied directly to a population using the operators' function form. In this case, function `closeOutput` can be used to close a specific file *output*, and close all unclosed files if *output* is unspecified. An exception will be raised if *output* does not exist or it has already been closed.

1.6.2 Function `describeEvolProcess`

describeEvolProcess(*initOps=[], preOps=[], matingScheme=MatingScheme, postOps=[], finalOps=[], gen=-1, numRep=1*)
This function takes the same parameters as `Simulator.evolve` and output a description of how an evolutionary process will be executed. It is recommended that you call this function if you have any doubt how your simulation will proceed.

1.6.3 Function `loadPopulation`

loadPopulation(*file*)
Load a population from a file saved by `Population::save()`.

1.6.4 Function `loadPedigree`

loadPedigree(*file, idField="ind_id", fatherField="father_id", motherField="mother_id", ploidy=2, loci=[], chromTypes=[], lociPos=[], chromNames=[], alleleNames=[], lociNames=[], subPopNames=[], infoFields=[]*)

Load a pedigree from a file saved by operator `PedigreeTagger` or function `Pedigree.save`. This file contains the ID of each offspring and their parent(s) and optionally sex ('M' or 'F'), affection status ('A' or 'U'), values of information fields and genotype at some loci. IDs of each individual and their parents are loaded to information fields *idField*, *fatherField* and *motherField*. Only numeric IDs are allowed, and individual IDs must be unique across all generations.

Because this file does not contain generation information, generations to which offspring belong are determined by the parent-offspring relationships. Individuals without parents are assumed to be in the top-most ancestral generation. This is the case for individuals in the top-most ancestral generation if the file is saved by function "Pedigree.save()", and for individuals who only appear as another individual's parent, if the file is saved by operator "PedigreeTagger". The order at which offsprng is specified is not important because this function essentially creates a top-most ancestral generation using IDs without parents, and creates the next generation using offspring of these parents, and so on until all generations are recreated. That is to say, if you have a mixture of pedigrees with different generations, they will be lined up from the top most ancestral generation.

If individual sex is not specified, sex of of parents are determined by their parental roles (father or mother) but the sex of individuals in the last generation can not be determined so they will all be males. If additional information fields are given, their names have to be specified using parameter *infoFields*. The rest of the columns are assued to be alleles, arranged *ploidy* consecutive columns for each locus. If paraemter *loci* is not specified, the number of loci is calculated by number of columns divided by *ploidy* (default to 2). All loci are assumed to be on one chromosome unless parameter *loci* is used to specified number of loci on each chromosome. Additional parameters such as *ploidy*, *chromTypes*, *lociPos*, *chromNames*, *alleleNames*, *lociNames* could be used to specified the genotype structured of the loaded pedigree. Please refer to class `Population` for details about these parameters.

1.6.5 Function `moduleInfo`

moduleInfo()

Return a dictionary with information regarding the currently loaded simuPOP module. This dictionary has the following keys:

- `revision`: revision number.
- `version`: simuPOP version string.
- `optimized`: Is this module optimized (`True` or `False`).
- `alleleType`: Allele type of the module (`short`, `long` or `binary`).
- `maxAllele`: the maximum allowed allele state, which is `1` for binary modules, `255` for short modules and `65535` for long modules.
- `compiler`: the compiler that compiles this module.
- `date`: date on which this module is compiled.
- `python`: version of python.
- `platform`: platform of the module.
- `wordsize`: size of word, can be either 32 or 64.
- `alleleBits`: the number of bits used to store an allele
- `maxNumSubPop`: maximum number of subpopulations.
- `maxIndex`: maximum index size (limits population size * total number of marker).
- `debug`: A dictionary with debugging codes as keys and the status of each debugging code (`True` or `False`) as their values.

1.6.6 Function `getRNG`

getRNG()
Return the currently used random number generator

1.6.7 Function `setRNG`

setRNG(*name=None, seed=0*)
Set the type or seed of existing random number generator using RNG*name* with *seed*. If using openMP, it sets the type or seed of random number generator of each thread.

1.6.8 Function `turnOnDebug`

turnOnDebug(*code=""*)
Set debug code *code*. More than one code could be specified using a comma separated string. Name of available codes are available from `moduleInfo()['debug'].keys()`.

1.6.9 Function `turnOffDebug`

turnOffDebug(*code="DBG_ALL"*)
Turn off debug code *code*. More than one code could be specified using a comma separated string. Default to turn off all debug codes.

Chapter 2

Operator References

2.1 Base class for all operators

2.1.1 Class `BaseOperator`

Operators are objects that act on populations. They can be applied to populations directly using their function forms, but they are usually managed and applied by a simulator. In the latter case, operators are passed to the `evolve` function of a simulator, and are applied repeatedly during the evolution of the simulator.

The *BaseOperator* class is the base class for all operators. It defines a common user interface that specifies at which generations, at which stage of a life cycle, to which populations and subpopulations an operator is applied. These are achieved by a common set of parameters such as `begin`, `end`, `step`, `at`, `stage` for all operators. Note that a specific operator does not have to honor all these parameters. For example, a Recombinator can only be applied during mating so it ignores the `stage` parameter.

An operator can be applied to all or part of the generations during the evolution of a simulator. At the beginning of an evolution, a simulator is usually at the beginning of generation `0`. If it evolves `10` generations, it evolves generations `0`, `1`, „„., and `9` (`10` generations) and stops at the begging of generation `10`. A negative generation number `a` has generation number `10 + a`, with -1 referring to the last evolved generation `9`. Note that the starting generation number of a simulator can be changed by its `setGen()` member function.

Output from an operator is usually directed to the standard output (`sys.stdout`). This can be configured using a output specification string, which can be `''` for no output, `'>'` standard terminal output (default), a filename prefixed by one or more `'>'` characters or a Python expression indicated by a leading exclamation mark (`'!expr'`). In the case of `'>filename'` (or equivalently `'filename'`), the output from an operator is written to this file. However, if two operators write to the same file `filename`, or if an operator writes to this file more than once, only the last write operation will succeed. In the case of `'>>filename'`, file `filename` will be opened at the beginning of the evolution and closed at the end. Outputs from multiple operators are appended. `>>>filename` works similar to `>>filename` but `filename`, if it already exists at the beginning of an evolutionary process, will not be cleared. If the output specification is prefixed by an exclamation mark, the string after the mark is considered as a Python expression. When an operator is applied to a population, this expression will be evaluated within the population's local namespace to obtain a population specific output specification. As an advanced feature, a Python function can be assigned to this parameter. Output strings will be sent to this function for processing.

class `BaseOperator`(*output, begin, end, step, at, reps, subPops, infoFields*)

The following parameters can be specified by all operators. However, an operator can ignore some parameters and the exact meaning of a parameter can vary.

output: A string that specifies how output from an operator is written, which can be `''` (no output), `'>'` (standard output), `'filename'` prefixed by one or more '>', or an Python expression prefixed by an exclamation mark

('!expr'). Alternatively, a Python function can be given to handle outputs.

begin: The starting generation at which an operator will be applied. Default to `0`. A negative number is interpreted as a generation counted from the end of an evolution (-1 being the last evolved generation).

end: The last generation at which an operator will be applied. Default to `-1`, namely the last generation.

step: The number of generations between applicable generations. Default to `1`.

at: A list of applicable generations. Parameters `begin`, `end`, and `step` will be ignored if this parameter is specified. A single generation number is also acceptable.

reps: A list of applicable replicates. A common default value `ALL_AVAIL` is interpreted as all replicates in a simulator. Negative indexes such as `-1` (last replicate) is acceptable. `rep=idx` can be used as a shortcut for `rep=[idx]`.

subPops: A list of applicable (virtual) subpopulations, such as `subPops=[sp1, sp2, (sp2, vsp1)]`. `subPops=[sp1]` can be simplied as `subPops=sp1`. Negative indexes are not supported. A common default value (`ALL_AVAIL`) of this parameter reprents all subpopulations of the population being aplied. Suport for this parameter vary from operator to operator and some operators do not support virtual subpopulations at all. Please refer to the reference manual of individual operators for their support for this parameter.

infoFields: A list of information fields that will be used by an operator. You usually do not need to specify this parameter because operators that use information fields usually have default values for this parameter.

apply(*pop*)
Apply an operator to population *pop* directly, without checking its applicability.

clone()
Return a cloned copy of an operator. This function is available to all operators.

2.2 Initialization

2.2.1 Class `InitSex`

This operator initializes sex of individuals, either randomly or use a list of sexes.

class `InitSex`(*maleFreq=0.5, maleProp=-1, sex=[], begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)
Create an operator that initializes individual sex to `MALE` or `FEMALE`. By default, it assigns sex to individuals randomly, with equal probability of having a male or a female. This probabability can be adjusted through parameter *maleFreq* or be made to exact proportions by specifying parameter *maleProp*. Alternatively, a fixed sequence of sexes can be assigned. For example, if `sex=[MALE, FEMALE]`, individuals will be assigned `MALE` and `FEMALE` successively. Parameter *maleFreq* or *maleProp* are ignored if *sex* is given. If a list of (virtual) subpopulation is specified in parameter *subPop*, only individuals in these subpopulations will be initialized. Note that the *sex* sequence, if used, is assigned repeatedly regardless of (virtual) subpopulation boundaries so that you can assign *sex* to all individuals in a population.

2.2.2 Class `InitInfo`

This operator initializes given information fields with a sequence of values, or a user-provided function such as `random.random`.

class `InitInfo`(*values, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)
Create an operator that initialize individual information fields *infoFields* using a sequence of values or a user-defined function. If a list of values are given, it will be used sequentially for all individuals. The values will be reused if its length is less than the number of individuals. The values will be assigned repeatedly regardless of subpopulation boundaries. If a Python function is given, it will be called, without any argument, whenever a value is needed. If a list of (virtual) subpopulation is specified in parameter *subPop*, only individuals in these subpopulations will be initialized.

2.2.3 Class `InitGenotype`

This operator assigns alleles at all or part of loci with given allele frequencies, proportions or values. This operator initializes all chromosomes, including unused genotype locations and customized chromosomes.

class `InitGenotype`(*freq=[], genotype=[], prop=[], haplotypes=[], loci=ALL_AVAIL, ploidy=ALL_AVAIL, begin=0, end=1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

This function creates an initializer that initializes individual genotypes with random alleles or haplotypes with specified frequencies (parameter *freq*) or proportions (parameter *prop*). If parameter *haplotypes* is not specified, *freq* specifies the allele frequencies of alleles `0`, `1`, ... respectively. Alternatively, you can use parameter *prop* to specified the exact proportions of alleles `0`, `1`, ..., although alleles with small proportions might not be assigned at all. Values of parameter *prob* or *prop* should add up to 1. If parameter *haplotypes* is specified, it should contain a list of haplotypes and parameter *prob* or *prop* specifies frequencies or proportions of each haplotype. If *loci*, *ploidy* and/or *subPop* are specified, only specified loci, ploidy, and individuals in these (virtual) subpopulations will be initialized. Parameter *loci* can be a list of loci indexes, names or `ALL_AVAIL`. If the length of a haplotype is not enough to fill all loci, the haplotype will be reused. If a list (or a single) haplotypes are specified without *freq* or *prop*, they are used with equal probability.

In the last case, if a sequence of genotype is specified, it will be uesd repeatedly to initialize all alleles sequentially. This works similar to function `Population.setGenotype()` except that you can limit the initialization to certain *loci* and *ploidy*.

2.3 Expression and Statements

2.3.1 Class `PyOutput`

This operator outputs a given string when it is applied to a population.

class `PyOutput`(*msg="", output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Creates a `PyOutput` operator that outputs a string *msg* to *output* (default to standard terminal output) when it is applied to a population. Please refer to class `BaseOperator` for a detailed description of common operator parameters such as *stage*, *begin* and *output*.

2.3.2 Class `PyEval`

A `PyEval` operator evaluates a Python expression in a population's local namespace when it is applied to this population. The result is written to an output specified by parameter *output*.

class `PyEval`(*expr="", stmts="", exposePop="", output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a `PyEval` operator that evaluates a Python expression *expr* in a population's local namespace when it is applied to this population. If Python statements *stmts* is given (a single or multi-line string), the statement will be executed before *expr*. If *exposePop* is set to an non-empty string, the current population will be exposed in its own local namespace as a variable with this name. This allows the execution of expressions such as `'pop.individual(0).allele(0)'`. The result of *expr* will be sent to an output stream specified by parameter `output`. The exposed population variable will be removed after *expr* is evaluated. Please refer to class `BaseOperator` for other parameters.

Note Although the statements and expressions are evaluated in a population's local namespace, they have access to a global namespace which is the module global namespace. It is therefore possible to refer to any module variable in these expressions. Such mixed use of local and global variables is, however, strongly discouraged.

2.3.3 Class `PyExec`

This operator executes given Python statements in a population's local namespace when it is applied to this population.

class `PyExec`(*stmts=""*, *exposePop=""*, *output=">"*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create a `PyExec` operator that executes statements *stmts* in a population's local namespace when it is applied to this population. If *exposePop* is given, current population will be exposed in its local namespace as a variable named by *exposePop*. Although multiple statements can be executed, it is recommended that you use this operator to execute short statements and use `PyOperator` for more complex once. Note that exposed population variables will be removed after the statements are executed.

2.3.4 Class `InfoEval`

Unlike operator `PyEval` and `PyExec` that work at the population level, in a population's local namespace, operator `InfoEval` works at the individual level, working with individual information fields. When this operator is applied to a population, information fields of eligible individuals are put into the local namespace of the population. A Python expression is then evaluated for each individual. The result is written to an output.

class `InfoEval`(*expr=""*, *stmts=""*, *usePopVars=False*, *exposeInd=""*, *output=">"*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create an operator that evaluate a Python expression *expr* using individual information fields and population variables as variables. If *exposeInd* is not empty, the individual itself will be exposed in the population's local namespace as a variable with name specified by *exposeInd*.

A Python expression (*expr*) is evaluated for each individual. The results are converted to strings and are written to an output specified by parameter *output*. Optionally, a statement (or several statements separated by newline) can be executed before *expr* is evaluated. The evaluation of this statement may change the value of information fields.

This operator is by default applied post-mating. If it stage is set to `DuringMating`, it will be applied to all offspring, regardless of `subPops` settings.

Parameter *usePopVars* is obsolete because population variables are always usable in such expressions.

2.3.5 Class `InfoExec`

Operator `InfoExec` is similar to `InfoEval` in that it works at the individual level, using individual information fields as variables. This is usually used to change the value of information fields. For example, `"b=a*2"` will set the value of information field `b` to `a*a` for all individuals.

class `InfoExec`(*stmts=""*, *usePopVars=False*, *exposeInd=""*, *output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create an operator that executes Python statements *stmts* using individual information fields and population variables as variables. If *exposeInd* is not empty, the individual itself will be exposed in the population's local namespace as a variable with name specified by *exposeInd*.

One or more python statements (*stmts*) are executed for each individual. Information fields of these individuals are then updated from the corresponding variables. For example, `a=1` will set information field *a* of all individuals to `1`, `a=b` will set information field *a* of all individuals to information field `b` or a population variable `b` if `b` is not an information field but a population variable, and `a=ind.sex()` will set information field *a* of all individuals to its sex (needs `exposeInd='ind'`.

This operator is by default applied post-mating. If it stage is set to `DuringMating`, it will be applied to all offspring, regardless of `subPops` settings.

Parameter *usePopVars* is obsolete because population variables will always be usable.

2.4 Demographic models

2.4.1 Class `Migrator`

This operator migrates individuals from (virtual) subpopulations to other subpopulations, according to either pre-specified destination subpopulation stored in an information field, or randomly according to a migration matrix.

In the former case, values in a specified information field (default to *migrate_to*) are considered as destination subpopulation for each individual. If *subPops* is given, only individuals in specified (virtual) subpopulations will be migrated where others will stay in their original subpopulation. Negative values are not allowed in this information field because they do not represent a valid destination subpopulation ID.

In the latter case, a migration matrix is used to randomly assign destination subpoulations to each individual. The elements in this matrix can be probabilities to migrate, proportions of individuals to migrate, or exact number of individuals to migrate.

By default, the migration matrix should have `m` by `m` elements if there are `m` subpopulations. Element `(i, j)` in this matrix represents migration probability, rate or count from subpopulation `i` to `j`. If *subPops* (length `m`) and/or *toSubPops* (length `n`) are given, the matrix should have `m` by `n` elements, corresponding to specified source and destination subpopulations. Subpopulations in *subPops* can be virtual subpopulations, which makes it possible to migrate, for example, males and females at different rates from a subpopulation. If a subpopulation in *toSubPops* does not exist, it will be created. In case that all individuals from a subpopulation are migrated, the empty subpopulation will be kept.

If migration is applied by probability, the row of the migration matrix corresponding to a source subpopulation is intepreted as probabilities to migrate to each destination subpopulation. Each individual's detination subpopulation is assigned randomly according to these probabilities. Note that the probability of staying at the present subpopulation is automatically calculated so the corresponding matrix elements are ignored.

If migration is applied by proportion, the row of the migration matrix corresponding to a source subpopulation is intepreted as proportions to migrate to each destination subpopulation. The number of migrants to each destination subpopulation is determined before random indidividuals are chosen to migrate.

If migration is applied by counts, the row of the migration matrix corresponding to a source subpopulation is intepreted as number of individuals to migrate to each detination subpopulation. The migrants are chosen randomly.

This operator goes through all source (virtual) subpopulations and assign detination subpopulation of each individual to an information field. Unexpected results may happen if individuals migrate from overlapping virtual subpopulations.

class `Migrator`(*rate=[], mode=BY_PROBABILITY, toSubPops=ALL_AVAIL, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields="migrate_to"*)

Create a Migrator that moves individuals from source (virtual) subpopulations *subPops* (default to migrate from all subpopulations) to destination subpopulations *toSubPops* (default to all subpopulations), according to existing values in an information field *infoFields*[0], or randomly according to a migration matrix *rate*. In the latter case, the size of the matrix should match the number of source and destination subpopulations.

Depending on the value of parameter *mode*, elements in the migration matrix (*rate*) are interpreted as either the probabilities to migrate from source to destination subpopulations (*mode* = `BY_PROBABILITY`), proportions of individuals in the source (virtual) subpopulations to the destination subpopulations (*mode* = `BY_PROPORTION`), numbers of migrants in the source (virtual) subpopulations (*mode* = `BY_COUNTS`), or ignored completely (*mode* = `BY_IND_INFO`). In the last case, parameter *subPops* is respected (only individuals in specified (virtual) subpopulations will migrate) but *toSubPops* is ignored.

This operator is by default applied pre-mating (parameter *stage*). Please refer to operator `BaseOperator` for a detailed explanation for all parameters.

2.4.2 Class `SplitSubPops`

Split a given list of subpopulations according to either sizes of the resulting subpopulations, proportion of individuals, or an information field. The resulting subpopulations will have the same name as the original subpopulation.

class `SplitSubPops`(*subPops=ALL_AVAIL, sizes=[], proportions=[], names=[], randomize=True, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, infoFields=[]*)

Split a list of subpopulations *subPops* into finer subpopulations. A single subpopulation is acceptable but virtual subpopulations are not allowed. All subpopulations will be split if *subPops* is not specified.

The subpopulations can be split in three ways:

- If parameter *sizes* is given, each subpopulation will be split into subpopulations with given size. The *sizes* should add up to the size of all orignal subpopulations.
- If parameter *proportions* is given, each subpopulation will be split into subpopulations with corresponding proportion of individuals. *proportions* should add up to `1`.
- If an information field is given (parameter *infoFields*), individuals having the same value at this information field will be grouped into a subpopulation. The number of resulting subpopulations is determined by the number of distinct values at this information field.

If parameter `randomize` is `True` (default), individuals will be randomized before a subpopulation is split. This is designed to remove artificial order of individuals introduced by, for example, some non- random mating schemes. Note that, however, the original individual order is not guaranteed even if this parameter is set to `False`.

Unless the last subpopulation is split, the indexes of existing subpopulations will be changed. If a subpopulation has a name, this name will become the name for all subpopulations separated from this subpopulation. Optionally, you can assign names to the new subpopulations using a list of names specified in parameter *names*. Because the same set of names will be used for all subpopulations, this parameter is not recommended when multiple subpopulations are split.

This operator is by default applied pre-mating (parameter *stage*). Please refer to operator `BaseOperator` for a detailed explanation for all parameters.

Note Unlike operator `Migrator`, this operator does not require an information field such as `migrate_to`.

2.4.3 Class `MergeSubPops`

This operator merges subpopulations *subPops* to a single subpopulation. If `subPops` is ignored, all subpopulations will be merged. Virtual subpopulations are not allowed in *subPops*.

class `MergeSubPops`(*subPops=ALL_AVAIL, name="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, infoFields=[]*)

Create an operator that merges subpopulations *subPops* to a single subpopulation. If *subPops* is not given, all subpopulations will be merged. The merged subpopulation will take the name of the first subpopulation being merged unless a new *name* is given.

This operator is by default applied pre-mating (parameter *stage*). Please refer to operator `BaseOperator` for a detailed explanation for all parameters.

2.4.4 Class `ResizeSubPops`

This operator resizes subpopulations to specified sizes. individuals are added or removed depending on the new subpopulation sizes.

class `ResizeSubPops`(*subPops=ALL_AVAIL, sizes=[], proportions=[], propagate=True, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, infoFields=[]*)

Resize given subpopulations *subPops* to new sizes *size*, or sizes proportional to original sizes (parameter *proportions*). All subpopulations will be resized if *subPops* is not specified. If the new size of a subpopulation is smaller than its original size, extra individuals will be removed. If the new size is larger, new individuals with empty genotype will be inserted, unless parameter *propagate* is set to `True` (default). In this case, existing individuals will be copied sequentially, and repeatedly if needed.

This operator is by default applied pre-mating (parameter *stage*). Please refer to operator `BaseOperator` for a detailed explanation for all parameters.

2.5 Genotype transmitters

2.5.1 Class `GenoTransmitter`

This during mating operator is the base class of all genotype transmitters. It is made available to users because it provides a few member functions that can be used by derived transmitters, and by customized Python during mating operators.

class `GenoTransmitter`(*output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a base genotype transmitter.

`clearChromosome`(*ind, ploidy, chrom*)

Clear (set alleles to zero) chromosome *chrom* on the *ploidy-th* homologous set of chromosomes of individual *ind*. It is equivalent to `ind.setGenotype([0], ploidy, chrom)`.

`copyChromosome`(*parent, parPloidy, offspring, ploidy, chrom*)

Transmit chromosome *chrom* on the *parPloidy* set of homologous chromosomes from *parent* to the *ploidy* set of homologous chromosomes of *offspring*. It is equivalent to `offspring.setGenotype(parent.genotype(parPloidy, chrom), polidy, chrom)`.

`copyChromosomes`(*parent, parPloidy, offspring, ploidy*)

Transmit the *parPloidy* set of homologous chromosomes from *parent* to the *ploidy* set of homologous chromosomes of *offspring*. Customized chromosomes are not copied. It is equivalent to `offspring.setGenotype(parent.genotype(parPloidy), ploidy)`.

2.5.2 Class `CloneGenoTransmitter`

This during mating operator copies parental genotype directly to offspring. This operator works for all mating schemes when one or two parents are involved. If both parents are passed, maternal genotype are copied. In addition to genotypes on all non-customized or specified chromosomes, sex and information fields are by default also coped copied from parent to offspring.

class `CloneGenoTransmitter`(*output="", chroms=ALL_AVAIL, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=ALL_AVAIL*)

Create a clone genotype transmitter (a during-mating operator) that copies genotypes from parents to offspring. If two parents are specified, genotypes are copied maternally. After genotype transmission, offspring sex is copied from parental sex even if sex has been determined by an offspring generator. All or specified information fields (parameter *infoFields*, default to `ALL_AVAIL`) will also be copied from parent to offspring. Parameters *subPops* is ignored. This operator by default copies genotypes on all autosome and sex chromosomes (excluding customized chromosomes), unless a parameter *chroms* is used to specify which chromosomes to copy.

2.5.3 Class `MendelianGenoTransmitter`

This Mendelian offspring generator accepts two parents and pass their genotypes to an offspring following Mendel's laws. Sex chromosomes are handled according to the sex of the offspring, which is usually determined in advance by an offspring generator. Customized chromosomes are not handled.

class `MendelianGenoTransmitter`(*output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a Mendelian genotype transmitter (a during-mating operator) that transmits genotypes from parents to offspring following Mendel's laws. Autosomes and sex chromosomes are handled but customized chromosomes are ignored. Parameters *subPops* and *infoFields* are ignored.

transmitGenotype(*parent, offspring, ploidy*)

Transmit genotype from parent to offspring, and fill the *ploidy* homologous set of chromosomes. This function does not set genotypes of customized chromosomes and handles sex chromosomes properly, according to offspring sex and `ploidy`.

2.5.4 Class `SelfingGenoTransmitter`

A genotype transmitter (during-mating operator) that transmits parental genotype of a parent through self-fertilization. That is to say, the offspring genotype is formed according to Mendel's laws, only that a parent serves as both maternal and paternal parents.

class `SelfingGenoTransmitter`(*output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a self-fertilization genotype transmitter that transmits genotypes of a parent to an offspring through self-fertilization. Cutsomized chromosomes are not handled. Parameters *subPops* and *infoFields* are ignored.

2.5.5 Class `HaplodiploidGenoTransmitter`

A genotype transmitter (during-mating operator) for haplodiploid populations. The female parent is considered as diploid and the male parent is considered as haploid (only the first homologous copy is valid). If the offspring is `FEMALE`, she will get a random copy of two homologous chromosomes of her mother, and get the only paternal copy from her father. If the offspring is `MALE`, he will only get a set of chromosomes from his mother.

class `HaplodiploidGenoTransmitter`(*output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a haplodiploid genotype transmitter (during-mating operator) that transmit parental genotypes from parents to offspring in a haplodiploid population. Parameters *subPops* and *infoFields* are ignored.

2.5.6 Class `MitochondrialGenoTransmitter`

This geno transmitter assumes that the first homologous copy of several (or all) `Customized` chromosomes are copies of mitochondrial chromosomes. It transmits these chromosomes randomly from the female parent to offspring. If this transmitter is applied to populations with more than one homologous copies of chromosomes, it transmits the first homologous copy of chromosomes and clears alleles (set to zero) on other homologous copies.

class `MitochondrialGenoTransmitter`(*output="", chroms=ALL_AVAIL, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Createa a mitochondrial genotype transmitter that treats all Customized chromosomes, or a list of chromosomes specified by *chroms*, as human mitochondrial chromosomes. These chromosomes should have the same length and the same number of loci. This operator transmits these chromosomes randomly from the female parent to offspring of both sexes.

2.5.7 Class `Recombinator`

A genotype transmitter (during-mating operator) that transmits parental chromosomes to offspring, subject to recombination and gene conversion. This can be used to replace `MendelianGenoTransmitter` and `SelfingGenoTransmitter`. It does not work in haplodiploid populations, although a customized genotype transmitter that makes uses this operator could be defined. Please refer to the simuPOP user's guide or online cookbook for details.

Recombination could be applied to all adjacent markers or after specified loci. Recombination rate between two adjacent markers could be specified directly, or calculated using physical distance between them. In the latter case, a recombination intensity is multiplied by physical distance between markers.

Gene conversion is interpreted as double-recombination events. That is to say, if a recombination event happens, it has

a certain probability (can be 1) to become a conversion event, namely triggering another recombination event down the chromosome. The length of the converted chromosome can be controlled in a number of ways.

Note:

simuPOP does not assume any unit to loci positions so recombination intensity could be explained differntly (e.g. cM/Mb, Morgan/Mb) depending on your intepretation of loci positions. For example, if basepair is used for loci position, `intensity=10◦8` indicates `10◦8` per basepair, which is equivalent to `10◦2` per Mb or 1 cM/Mb. If `Mb` is used for physical positions, the same recombination intensity could be achieved by `intensity=0.01`.

class `Recombinator`(*rates=[], intensity=-1, loci=ALL_AVAIL, convMode=NO_CONVERSION, output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a Recombinator (a mendelian genotype transmitter with recombination and gene conversion) that passes genotypes from parents (or a parent in case of self-fertilization) to offspring.

Recombination happens by default between all adjacent markers but can be limited to a given set of *loci*, which can be a list of loci indexes, names or `ALL_AVAIL`. Each locus in this list specifies a recombination point between the locus and the locus immediately **before** it. Loci that are the first locus on each chromosome are ignored.

If a single recombination rate (parameter *rates*) is specified, it will used for all loci (all loci or loci specified by parameter *loci*), regardless of physical distances between adjacent loci.

If a list of recombination rates are specified in *rates*, a parameter *loci* with the same length should also be specified. Different recombination rates can then be used after these loci (between specified loci and their immediate neighbor to the right).

A recombination intensity (*intensity*) can be used to specify recombination rates that are proportional to physical distances between adjacent markers. If the physical distance between two markers is `d`, the recombination rate between them will be `intensity * d`. No unit is assume for loci position and recombination intensity.

Gene conversion is controlled using parameter *convMode*, which can be

- `NoConversion`: no gene conversion (default).
- `(NUM_MARKERS, prob, n)`: With probability *prob*, convert a fixed number (*n*) of markers if a recombination event happens.
- `(GEOMETRIC_DISTRIBUTION, prob, p)`: With probability *prob*, convert a random number of markers if a recombination event happens. The number of markes converted follows a geometric distribution with probability *p*.
- `(TRACT_LENGTH, prob, n)`: With probability *prob*, convert a region of fixed tract length (*n*) if a recombination event happens. The actual number of markers converted depends on loci positions of surrounding loci. The starting position of this tract is the middle of two adjacent markers. For example, if four loci are located at `0, 1, 2, 3` respectively, a conversion event happens between `0` and `1`, with a tract length 2 will start at 0.5 and end at 2.5, covering the second and third loci.
- `(EXPONENTIAL_DISTRIBUTION, prob, p)`: With probability *prob*, convert a region of random tract length if a recombination event happens. The distribution of tract length follows a exponential distribution with probability `p`. The actual number of markers converted depends on loci positions of surrounding loci.

simuPOP uses this probabilistic model of gene conversion because when a recombination event happens, it may become a recombination event if the if the Holliday junction is resolved/repaired successfully, or a conversion event if the junction is not resolved/repaired. The probability, however, is more commonly denoted by the ratio of conversion to recombination events in the literature. This ratio varies greatly from study to study, ranging from 0.1 to 15 (Chen et al, Nature Review Genetics, 2007). This translate to 0.1/0.9õ.1 to 15/16õ.94 of the gene conversion probability.

A `Recombinator` usually does not send any output. However, if an information field is given (parameter *infoFields*), this operator will treat this information field as an unique ID of parents and offspring and output all recombination events in the format of `offspring_id parent_id starting_ploidy loc1 loc2 ...` where `starting_ploidy` indicates which homologous copy genotype replication starts from (`0` or `1`), `loc1`, `loc2` etc are loci after which recombination events happens. If there are multiple chromosomes on the genome, you will see

a lot of (fake) recombination events because of independent segregation of chromosomes. Such a record will be generated for each set of homologous chromosomes so an diploid offspring will have two lines of output. Note that individual IDs need to be set (using a `IdTagger` operator) before this Recombinator is applied.

Note conversion tract length is usually short, and is estimated to be between 337 and 456 bp, with overall range between maybe 50 - 2500 bp. This is usually not enough to convert, for example, two adjacent markers from the HapMap dataset.There is no recombination between sex chromosomes (Chromosomes X and Y), although recombination is possible between pesudoautosomal regions on these chromosomes. If such a feature is required, you will have to simulate the pesudoautosomal regions as separate chromosomes.

transmitGenotype(*parent, offspring, ploidy*)

This function transmits genotypes from a *parent* to the *ploidy-th* homologous set of chromosomes of an *offspring*. It can be used, for example, by a customized genotype transmitter to use sex-specific recombination rates to transmit parental genotypes to offspring.

2.6 Mutation

2.6.1 Class `BaseMutator`

Class `mutator` is the base class of all mutators. It handles all the work of picking an allele at specified loci from certain (virtual) subpopulation with certain probability, and calling a derived mutator to mutate the allele. Alleles can be changed before and after mutation if existing allele numbers do not match those of a mutation model.

class **BaseMutator**(*rates=[], loci=ALL_AVAIL, mapIn=[], mapOut=[], context=0, output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

A mutator mutates alleles from one state to another with given probability. This base mutator does not perform any mutation but it defines common behaviors of all mutators.

By default, a mutator mutates all alleles in all populations of a simulator at all generations. A number of parameters can be used to restrict mutations to certain generations (parameters *begin*, *end*, *step* and *at*), replicate populations (parameter *rep*), (virtual) subpopulations (parameter *subPops*) and loci (parameter *loci*). Parameter *loci* can be a list of loci indexes, names or `ALL_AVAIL`. Please refer to class `BaseOperator` for a detailed explanation of these parameters.

Parameter *rate* or its equivalence specifies the probability that a a mutation event happens. The exact form and meaning of *rate* is mutator-specific. If a single rate is specified, it will be applied to all *loci*. If a list of mutation rates are given, they will be applied to each locus specified in parameter *loci*. Note that not all mutators allow specification of multiple mutation rate, especially when the mutation rate itself is a list or matrix.

Alleles at a locus are non-negative numbers 0, 1, ... up to the maximum allowed allele for the loaded module (1 for binary, 255 for short and 65535 for long modules). Whereas some general mutation models treat alleles as numbers, other models assume specific interpretation of alleles. For example, an `AcgtMutator` assumes alleles `0`, `1`, `2` and `3` as nucleotides `A`, `C`, `G` and `T`. Using a mutator that is incompatible with your simulation will certainly yield erroneous results.

If your simulation assumes different alleles with a mutation model, you can map an allele to the allele used in the model and map the mutated allele back. This is achieved using a *mapIn* list with its `i-th` item being the corresponding allele of real allele `i`, and a *mapOut* list with its *i-th* item being the real allele of allele `i` assumed in the model. For example `mapIn=[0, 0, 1]` and `mapOut=[1, 2]` would allow the use of a `SNPMutator` to mutate between alleles 1 and 2, instead of 0 and 1. Parameters *mapIn* and *mapOut* also accept a user-defined Python function that returns a corresponding allele for a given allele. This allows easier mapping between a large number of alleles and advanced models such as random emission of alleles.

Some mutation models are context dependent. Namely, how an allele mutates will depend on its adjecent alleles. Whereas most simuPOP mutators are context independent, some of them accept a parameter *context* which is the number of alleles to the left and right of the mutated allele. For example *context=1* will make the alleles to the immediate left and right to a mutated allele available to a mutator. These alleles will be mapped in if parameter *mapIn* is defined. How exactly a mutator makes use of these information is mutator dependent.

2.6.2 Class `MatrixMutator`

A matrix mutator mutates alleles `0`, `1`, ..., `n-1` using a `n` by `n` matrix, which specifies the probability at which each allele mutates to another. Conceptually speaking, this mutator goes through all mutable allele and mutate it to another state according to probabilities in the corresponding row of the rate matrix. Only one mutation rate matrix can be specified which will be used for all specified loci. #

class `MatrixMutator`(*rate, loci=ALL_AVAIL, mapIn=[], mapOut=[], output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a mutator that mutates alleles `0`, `1`, ..., `n-1` using a `n` by `n` matrix `rate`. Item `(i,j)` of this matrix specifies the probability at which allele *i* mutates to allele *j*. Diagnal items `(i, i)` are ignored because they are automatically determined by other probabilities. Only one mutation rate matrix can be specified which will be used for all loci in the applied population, or loci specified by parameter *loci*. If alleles other than `0`, `1`, ..., `n-1` exist in the population, they will not be mutated although a warning message will be given if debugging code `DBG_WARNING` is turned on. Please refer to classes `mutator` and `BaseOperator` for detailed explanation of other parameters.

2.6.3 Class `KAlleleMutator`

This mutator implements a *k-allele* mutation model that assumes *k* allelic states (alleles 0, 1, 2, ..., *k-1*) at each locus. When a mutation event happens, it mutates an allele to any other states with equal probability.

class `KAlleleMutator`(*k, rates=[], loci=ALL_AVAIL, mapIn=[], mapOut=[], output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a k-allele mutator that mutates alleles to one of the other `k-1` alleles with equal probability. This mutator by default applies to all loci unless parameter *loci* is specified. A single mutation rate will be used for all loci if a single value of parameter *rates* is given. Otherwise, a list of mutation rates can be specified for each locus in parameter *loci*. If the mutated allele is larger than or equal to `k`, it will not be mutated. A warning message will be displayed if debugging code `DBG_WARNING` is turned on. Please refer to classes `mutator` and `BaseOperator` for descriptions of other parameters.

2.6.4 Class `StepwiseMutator`

A stepwise mutation model treats alleles at a locus as the number of tandem repeats of microsatellite or minisatellite markers. When a mutation event happens, the number of repeats (allele) either increase or decrease. A standard stepwise mutation model increases of decreases an allele by 1 with equal probability. More complex models (generalized stepwise mutation model) are also allowed. Note that an allele cannot be mutated beyond boundaries (0 and maximum allowed allele).

class `StepwiseMutator`(*rates=[], loci=ALL_AVAIL, incProb=0.5, maxAllele=0, mutStep=[], mapIn=[], mapOut=[], output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a stepwise mutation mutator that mutates an allele by increasing or decreasing it. This mutator by default applies to all loci unless parameter *loci* is specified. A single mutation rate will be used for all loci if a single value of parameter *rates* is given. Otherwise, a list of mutation rates can be specified for each locus in parameter *loci*.

When a mutation event happens, this operator increases or decreases an allele by *mutStep* steps. Acceptable input of parameter *mutStep* include

- A number: This is the default mode with default value 1.
- `(GEOMETRIC_DISTRIBUTION, p)`: The number of steps follows a a geometric distribution with parameter *p*.
- A Python function: This user defined function accepts the allele being mutated and return the steps to mutate.

The mutation process is usually neutral in the sense that mutating up and down is equally likely. You can adjust parameter *incProb* to change this behavior.

If you need to use other generalized stepwise mutation models, you can implement them using a `PyMutator`. If performance becomes a concern, I may add them to this operator if provided with a reliable reference.

2.6.5 Class `PyMutator`

This hybrid mutator accepts a Python function that determines how to mutate an allele when an mutation event happens.

class `PyMutator`(*rates=[], loci=ALL_AVAIL, func=None, context=0, mapIn=[], mapOut=[], output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a hybrid mutator that uses a user-provided function to mutate an allele when a mutation event happens. This function (parameter *func*) accepts the allele to be mutated as parameter `allele` and optional array of alleles as parameter `context`, which are *context* alleles the left and right of the mutated allele. Invalid context alleles (e.g. left allele to the first locus of a chromosome) will be marked by -1. The return value of this function will be used to mutate the passed allele. The passed, returned and context alleles might be altered if parameter *mapIn* and *mapOut* are used. This mutator by default applies to all loci unless parameter *loci* is specified. A single mutation rate will be used for all loci if a single value of parameter *rates* is given. Otherwise, a list of mutation rates can be specified for each locus in parameter *loci*. Please refer to classes `mutator` and `BaseOperator` for descriptions of other parameters.

2.6.6 Class `MixedMutator`

This mixed mutator accepts a list of mutators and use one of them to mutate an allele when an mutation event happens.

class `MixedMutator`(*rates=[], loci=ALL_AVAIL, mutators=[], prob=[], mapIn=[], mapOut=[], context=0, output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a mutator that randomly chooses one of the specified *mutators* to mutate an allele when a mutation event happens. The mutators are choosen according to a list of probabilities (*prob*) that should add up to `1`. The passed and returned alleles might be changed if parameters *mapIn* and *mapOut* are used. Most parameters, including *loci*, *mapIn*, *mapOut*, *rep*, and *subPops* of mutators specified in parameter *mutators* are ignored. This mutator by default applies to all loci unless parameter *loci* is specified. Please refer to classes `mutator` and `BaseOperator` for descriptions of other parameters.

2.6.7 Class `ContextMutator`

This context-dependent mutator accepts a list of mutators and use one of them to mutate an allele depending on the context of the mutated allele.

class `ContextMutator`(*rates=[], loci=ALL_AVAIL, mutators=[], contexts=[], mapIn=[], mapOut=[], output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a mutator that choose one of the specified *mutators* to mutate an allele when a mutation event happens. The mutators are choosen according to the context of the mutated allele, which is specified as a list of alleles to the left and right of an allele (*contexts*). For example, `contexts=[(0,0), (0,1), (1,1)]` indicates which mutators should be used to mutate allele `X` in the context of `0X0`, `0X1`, and `1X1`. A context can include more than one alleles at both left and right sides of a mutated allele but all contexts should have the same (even) number of alleles. If an allele does not have full context (e.g. when a locus is the first locus on a chromosome), unavailable alleles will be marked as -1. There should be a mutator for each context but an additional mutator can be specified as the default mutator for unmatched contexts. If parameters *mapIn* is specified, both mutated allele and its context alleles will be mapped. Most parameters, including *loci*, *mapIn*, *mapOut*, *rep*, and *subPops* of mutators specified in parameter *mutators* are ignored. This mutator by default applies to all loci unless parameter *loci* is specified. Please refer to classes `mutator` and `BaseOperator` for descriptions of other parameters.

2.6.8 Class `PointMutator`

A point mutator is different from all other mutators because mutations in this mutator do not happen randomly. Instead, it happens to specific loci and mutate an allele to a specific state, regardless of its original state. This mutator is usually used to introduce a mutant to a population.

class `PointMutator`(*loci, allele, ploidy=0, inds=[], output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=0, infoFields=[]*)

Create a point mutator that mutates alleles at specified *loci* to a given *allele* of individuals *inds*. If there are multiple alleles at a locus (e.g. individuals in a diploid population), only the first allele is mutated unless indexes of alleles are listed in parameter *ploidy*. This operator is by default applied to individuals in the first subpopulation but you can apply it to a different or more than one (virtual) subpopulations using parameter *subPops* ("AllAvail" is also accepted). Please refer to class `BaseOperator` for detailed descriptions of other parameters.

2.6.9 Class `SNPMutator`

A mutator model that assumes two alleles 0 and 1 and accepts mutation rate from 0 to 1, and from 1 to 0 alleles.

class `SNPMutator`(*u=0, v=0, *args, **kwargs*)

Return a `MatrixMutator` with proper mutate matrix for a two-allele mutation model using mutation rate from allele 0 to 1 (parameter `u`) and from 1 to 0 (parameter `v`)

2.6.10 Class `AcgtMutator`

This mutation operator assumes alleles 0, 1, 2, 3 as nucleotides `A`, `C`, `G` and `T` and use a 4 by 4 mutation rate matrix to mutate them. Although a general model needs 12 parameters, less parameters are needed for specific nucleotide mutation models (parameter `model`). The length and meaning of parameter `rate` is model dependent.

class `AcgtMutator`(*rate=[], model='general', *args, **kwargs*)

Create a mutation model that mutates between nucleotides `A`, `C`, `G`, and `T` (alleles are coded in that order as 0, 1, 2 and 3). Currently supported models are Jukes and Cantor 1969 model (`JC69`), Kimura's 2-parameter model (`K80`), Felsenstein 1981 model (`F81`), Hasgawa, Kishino and Yano 1985 model (`HKY85`), Tamura 1992 model (`T92`), Tamura and Nei 1993 model (`TN93`), Generalized time reversible model (`GTR`), and a general model (`general`) with 12 parameters. Please refer to the simuPOP user's guide for detailed information about each model.

2.7 Penetrance

2.7.1 Class `BasePenetrance`

A penetrance model models the probability that an individual has a certain disease provided that he or she has certain genetic (genotype) and environmental (information field) riske factors. A penetrance operator calculates this probability according to provided information and set his or her affection status randomly. For example, an individual will have probability 0.8 to be affected if the penetrance is 0.8. This class is the base class to all penetrance operators and defines a common interface for all penetrance operators.

A penetrance operator can be applied at any stage of an evolutionary cycle. If it is applied before or after mating, it will set affection status of all parents and offspring, respectively. If it is applied during mating, it will set the affection status of each offspring. You can also apply a penetrance operator to an individual using its `applyToIndividual` member function.

By default, a penetrance operator assigns affection status of individuals but does not save the actual penetrance value. However, if an information field is specified, penetrance values will be saved to this field for future analysis.

When a penetrance operator is applied to a population, it is only applied to the current generation. You can, however, use parameter *ancGens* to set affection status for all ancestral generations (`ALL_AVAIL`), or individuals in specified generations if a list of ancestral generations is specified. Note that this parameter is ignored if the operator is applied during mating.

class `BasePenetrance`(*ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a base penetrance operator. This operator assign individual affection status in the present generation (default). If `ALL_AVAIL` or a list of ancestral generations are spcified in parameter *ancGens*, individuals in specified ancestral generations will be processed. A penetrance operator can be applied to specified (virtual) subpopulations (parameter *subPops*) and replicates (parameter *reps*). If an informatio field is given, penetrance value will be stored in this information field of each individual.

`apply`(*pop*)

Set penetrance to all individuals and record penetrance if requested

`applyToIndividual`(*ind, pop=None*)

Apply the penetrance operator to a single individual *ind* and set his or her affection status. A population reference can be passed if the penetrance model depends on population properties such as generation number. This function returns the affection status.

2.7.2 Class `MapPenetrance`

This penetrance operator assigns individual affection status using a user-specified penetrance dictionary.

class `MapPenetrance`(*loci, penetrance, ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a penetrance operator that get penetrance value from a dictionary *penetrance* with genotype at *loci* as keys, and *penetrance* as values. For each individual, genotypes at *loci* are collected one by one (e.g. p0_loc0, p1_loc0, p0_loc1, p1_loc1... for a diploid individual) and are looked up in the dictionary. Parameter *loci* can be a list of loci indexes, names or `ALL_AVAIL`. If a genotype cannot be found, it will be looked up again without phase information (e.g. `(1,0)` will match key `(0,1)`). If the genotype still can not be found, a `ValueError` will be raised. This operator supports sex chromosomes and haplodiploid populations. In these cases, only valid genotypes should be used to generator the dictionary keys.

2.7.3 Class `MaPenetrance`

This operator is called a 'multi-allele' penetrance operator because it groups multiple alleles into two groups: wildtype and non-wildtype alleles. Alleles in each allele group are assumed to have the same effect on individual penetrance. If we denote all wildtype alleles as `A`, and all non-wildtype alleles `a`, this operator assign Individual penetrance according to genotype `AA`, `Aa`, `aa` in the diploid case, and `A` and `a` in the haploid case.

class `MaPenetrance`(*loci, penetrance, wildtype=0, ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Creates a multi-allele penetrance operator that groups multiple alleles into a wildtype group (with alleles *wildtype*, default to `[0]`), and a non-wildtype group. A list of penetrance values is specified through parameter *penetrance*, for genotypes at one or more *loci*. Parameter *loci* can be a list of loci indexes, names or `ALL_AVAIL`. If we denote wildtype alleles using capital letters `A`, `B` ... and non-wildtype alleles using small letters `a`, `b` ..., the penetrance values should be for

- genotypes `A` and `a` for the haploid single-locus case,
- genotypes `AB`, `Ab`, `aB` and `bb` for haploid two=locus cases,
- genotypes `AA`, `Aa` and `aa` for diploid single-locus cases,
- genotypes `AABB`, `AABb`, `AAbb`, `AaBB`, `AaBb`, `Aabb`, `aaBB`, `aaBb`, and `aabb` for diploid two-locus cases,
- and in general 2**n for diploid and 3**n for haploid cases if there are `n` loci.

This operator does not support haplodiploid populations and sex chromosomes.

2.7.4 Class `MlPenetrance`

This penetrance operator is created by a list of penetrance operators. When it is applied to an individual, it applies these penetrance operators to the individual, obtain a list of penetrance values, and compute a combined penetrance value from them and assign affection status accordingly. ADDITIVE, multiplicative, and a heterogeneour multi-locus model are supported. Please refer to Neil Rish (1989) "Linkage Strategies for Genetically Complex Traits" for some analysis of these models.

class `MlPenetrance`(*ops, mode=MULTIPLICATIVE, ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a multiple-locus penetrance operator from a list penetrance operator *ops*. When this operator is applied to an individual (parents when used before mating and offspring when used during mating), it applies these operators to the individual and obtain a list of (usually single-locus) penetrance values. These penetrance values are combined to a single penetrance value using

- *Prod(f_i)*, namely the product of individual penetrance if *mode* = `MULTIPLICATIVE`,
- *sum(f_i)* if *mode* = `ADDITIVE`, and
- *1-Prod(1 - f_i)* if *mode* = `HETEROGENEITY`

0 or 1 will be returned if the combined penetrance value is less than zero or greater than 1.

2.7.5 Class `PyPenetrance`

This penetrance operator assigns penetrance values by calling a user provided function. It accepts a list of loci (parameter `loci`), and a Python function `func` which should be defined with one or more of parameters `geno`, `gen`, `ind`, `pop`, or names of information fields. When this operator is applied to a population, it passes genotypes at specified loci, generation number, a reference to an individual, a reference to the current population (usually used to retrieve population variables) and values at specified information fields to respective parameters of this function. The returned penetrance values will be used to determine the affection status of each individual.

class `PyPenetrance`(*func, loci=[], ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a Python hybrid penetrance operator that passes genotype at specified *loci*, values at specified information fields (if requested), and a generation number to a user-defined function *func*. Parameter *loci* can be a list of loci indexes, names, or `ALL_AVAIL`. The return value will be treated as Individual penetrance.

2.8 Quantitative Trait

2.8.1 Class `BaseQuanTrait`

A quantitative trait in simuPOP is simply an information field. A quantitative trait model simply assigns values to one or more information fields (called trait fields) of each individual according to its genetic (genotype) and environmental (information field) factors. It can be applied at any stage of an evolutionary cycle. If a quantitative trait operator is applied before or after mating, it will set the trait fields of all parents and offspring. If it is applied during mating, it will set the trait fields of each offspring.

When a quantitative trait operator is applied to a population, it is only applied to the current generation. You can, however, use parameter *ancGen=-1* to set the trait field of all ancestral generations, or a generation index to apply to only ancestral generation younger than *ancGen*. Note that this parameter is ignored if the operator is applied during mating.

class `BaseQuanTrait`(*ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a base quantitative trait operator. This operator assigns one or more quantitative traits to trait fields in the present generation (default). If `ALL_AVAIL` or a list of ancestral generations are specified, this operator will

be applied to individuals in these generations as well. A quantitative trait operator can be applied to specified (virtual) subpopulations (parameter *subPops*) and replicates (parameter *reps*).

apply(*pop*)
Set `qtrait` to all individual

2.8.2 Class `PyQuanTrait`

This quantitative trait operator assigns a trait field by calling a user provided function. It accepts a list of loci (parameter *loci*), and a Python function `func` which should be defined with one or more of parameters `geno`, `gen`, `ind`, or names of information fields. When this operator is applied to a population, it passes genotypes at specified loci, generation number, a reference to an individual, and values at specified information fields to respective parameters of this function. The return values will be assigned to specified trait fields.

class `PyQuanTrait`(*func, loci=[], ancGens=ALL_AVAIL, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)
Create a Python hybrid quantitative trait operator that passes genotype at specified *loci*, optional values at specified information fields (if requested), and an optional generation number to a user-defined function *func*. Parameter *loci* can be a list of loci indexes, names, or `ALL_AVAIL`. The return value will be assigned to specified trait fields (*infoField*). If only one trait field is specified, a number or a sequence of one element is acceptable. Otherwise, a sequence of values will be accepted and be assigned to each trait field.

2.9 Natural selection

2.9.1 Class `BaseSelector`

This class is the base class to all selectors, namely operators that perform natural selection. It defines a common interface for all selectors.

A selector can be applied before mating or during mating. If a selector is applied to one or more (virtual) subpopulations of a parental population before mating, it sets individual fitness values to all involved parents to an information field (default to *fitness*). When a mating scheme that supports natural selection is applied to the parental population, it will select parents with probabilities that are proportional to individual fitness stored in an information field (default to *fitness*). Individual fitness is considered **relative** fitness and can be any non-negative number. This simple process has some implications that can lead to advanced usages of natural selection in simuPOP:

- It is up to the mating scheme how to handle individual fitness. Some mating schemes do not support natural selection at all.
- A mating scheme performs natural selection according to fitness values stored in an information field. It does not care how these values are set. For example, fitness values can be inherited from a parent using a tagging operator, or set directly using a Python operator.
- A mating scheme can treat any information field as fitness field. If an specified information field does not exist, or if all individuals have the same fitness values (e.g. 0), the mating scheme selects parents randomly.
- Multiple selectors can be applied to the same parental generation. individual fitness is determined by the last fitness value it is assigned.
- A selection operator can be applied to virtual subpopulations and set fitness values only to part of the individuals.
- individuals with zero fitness in a subpopulation with anyone having a positive fitness value will not be selected to produce offspring. This can sometimes lead to unexpected behaviors. For example, if you only assign fitness value to part of the individuals in a subpopulation, the rest of them will be effectively discarded. If you migrate individuals with valid fitness values to a subpopulation with all individuals having zero fitness, the migrants will be the only mating parents.

- It is possible to assign multiple fitness values to different information fields so that different homogeneous mating schemes can react to different fitness schemes when they are used in a heterogeneous mating scheme.
- You can apply a selector to the offspring generation using the *postOps* parameter of `Simulator.evolve`, these fitness values will be used when the offspring generation becomes parental generation in the next generation.

Alternatively, a selector can be used as a during mating operator. In this case, it caculates fitness value for each offspring which will be treated as **absolute** fitness, namely the probability for each offspring to survive. This process uses the fact that an individual will be discarded when any of the during mating operators returns *False*. It is important to remember that:

- individual fitness needs to be between 0 and 1 in this case.
- This method applies natural selection to offspring instead of parents. These two implementation can be identical or different depending on the mating scheme used.
- Seleting offspring is less efficient than the selecting parents, especially when fitness values are low.
- Parameter *subPops* are applied to the offspring population and is used to judge if an operator should be applied. It thus does not make sense to apply a selector to a virtual subpopulation with affected individuals.

class `BaseSelector`(*output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=ALL_AVAIL*)
Create a base selector object. This operator should not be created directly.

2.9.2 Class `MapSelector`

This selector assigns individual fitness values using a user-specified dictionary. This operator can be applied to populations with arbitrary number of homologous chromosomes.

class `MapSelector`(*loci, fitness, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=ALL_AVAIL*)
Create a selector that assigns individual fitness values using a dictionary *fitness* with genotype at *loci* as keys, and fitness as values. Parameter *loci* can be a list of indexes, loci names or `ALL_AVAIL`. For each individual (parents if this operator is applied before mating, and offspring if this operator is applied during mating), genotypes at *loci* are collected one by one (e.g. p0_loc0, p1_loc0, p0_loc1, p1_loc1... for a diploid individual) and are looked up in the dictionary. If a genotype cannot be found, it will be looked up again without phase information (e.g. `(1,0)` will match key `(0,1)`). If the genotype still can not be found, a `ValueError` will be raised. This operator supports sex chromosomes and haplodiploid populations. In these cases, only valid genotypes should be used to generator the dictionary keys.

2.9.3 Class `MaSelector`

This operator is called a 'multi-allele' selector because it groups multiple alleles into two groups: wildtype and non-wildtype alleles. Alleles in each allele group are assumed to have the same effect on individual fitness. If we denote all wildtype alleles as `A`, and all non-wildtype alleles `a`, this operator assign individual fitness according to genotype `AA`, `Aa`, `aa` in the diploid case, and `A` and `a` in the haploid case.

class `MaSelector`(*loci, fitness, wildtype=0, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=ALL_AVAIL*)
Creates a multi-allele selector that groups multiple alleles into a wildtype group (with alleles *wildtype*, default to `[0]`), and a non-wildtype group. A list of fitness values is specified through parameter *fitness*, for genotypes at one or more *loci*. Parameter *loci* can be a list of indexes, loci names or `ALL_AVAIL`. If we denote wildtype alleles using capital letters `A`, `B` ... and non-wildtype alleles using small letters `a`, `b` ..., the fitness values should be for

- genotypes `A` and `a` for the haploid single-locus case,

- genotypes `AB`, `Ab`, `aB` and `bb` for haploid two=locus cases,
- genotypes `AA`, `Aa` and `aa` for diploid single-locus cases,
- genotypes `AABB`, `AABb`, `AAbb`, `AaBB`, `AaBb`, `Aabb`, `aaBB`, `aaBb`, and `aabb` for diploid two-locus cases,
- and in general 2**n for diploid and 3**n for haploid cases if there are `n` loci.

This operator does not support haplodiploid populations and sex chromosomes.

2.9.4 Class `MlSelector`

This selector is created by a list of selectors. When it is applied to an individual, it applies these selectors to the individual, obtain a list of fitness values, and compute a combined fitness value from them. ADDITIVE, multiplicative, and a heterogeneour multi-locus model are supported.

class `MlSelector`(*ops, mode=MULTIPLICATIVE, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=ALL_AVAIL*)

Create a multiple-locus selector from a list selection operator *selectors*. When this operator is applied to an individual (parents when used before mating and offspring when used during mating), it applies these operators to the individual and obtain a list of (usually single-locus) fitness values. These fitness values are combined to a single fitness value using

- *Prod(f_i)*, namely the product of individual fitness if *mode* = `MULTIPLICATIVE`,
- *1-sum(1 - f_i)* if *mode* = `ADDITIVE`,
- *1-Prod(1 - f_i)* if *mode* = `HETEROGENEITY`, and
- *exp(- sum(1 - f_i))* if *mode* = `EXPONENTIAL`,

zero will be returned if the combined fitness value is less than zero.

2.9.5 Class `PySelector`

This selector assigns fitness values by calling a user provided function. It accepts a list of loci (parameter *loci*) and a Python function `func` which should be defined with one or more of parameters `geno`, `gen`, `ind`, `pop` or names of information fields. Parameter *loci* can be a list of loci indexes, names or `ALL_AVAIL`. When this operator is applied to a population, it passes genotypes at specified loci, generation number, a reference to an individual, a reference to the current population (usually used to retrieve population variable), and values at specified information fields to respective parameters of this function. The returned value will be used to determine the fitness of each individual.

class `PySelector`(*func, loci=[], begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=ALL_AVAIL*)

Create a Python hybrid selector that passes genotype at specified *loci*, values at specified information fields (if requested) and a generation number to a user-defined function *func*. The return value will be treated as individual fitness.

2.10 Tagging operators

2.10.1 Class `IdTagger`

An IdTagger gives a unique ID for each individual it is applies to. These ID can be used to uniquely identify an individual in a multi-generational population and be used to reliably reconstruct a Pedigree.

To ensure uniqueness across populations, a single source of ID is used for this operator. individual IDs are assigned consecutively starting from 1. Value 1 instead of 0 is used because most software applications use 0 as missing values

for parentship. If you would like to reset the sequence or start from a different number, you can call the `reset(startID)` function of any `IdTagger`.

An `IdTagger` is usually used during-mating to assign ID to each offspring. However, if it is applied directly to a population, it will assign unique IDs to all individuals in this population. This property is usually used in the `preOps` parameter of function `Simulator.evolve` to assign initial ID to a population.

class `IdTagger`(*begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields="ind_id"*)

Create an `IdTagger` that assign an unique ID for each individual it is applied to. The IDs are created sequentially and are stored in an information field specified in parameter *infoFields* (default to `ind_id`). This operator is considered a during-mating operator but it can be used to set ID for all individuals of a population when it is directly applied to the population.

`reset`(*startID=1*)

Reset the global individual ID number so that IdTaggers will start from id (default to 1) again.

2.10.2 Class `InheritTagger`

An inheritance tagger passes values of parental information field(s) to the corresponding fields of offspring. If there are two parental values from parents of a sexual mating event, a parameter *mode* is used to specify how to assign offspring information fields.

class `InheritTagger`(*mode=PATERNAL, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields=[]*)

Creates an inheritance tagger that passes values of parental information fields (parameter *infoFields*) to the corresponding fields of offspring. If there is only one parent, values at the specified information fields are copied directly. If there are two parents, parameter *mode* specifies how to pass them to an offspring. More specifically,

- `mode=MATERNAL` Passing the value from mother.
- `mode=PATERNAL` Passing the value from father.
- `mode=MEAN` Passing the average of two values.
- `mode=MAXIMUM` Passing the maximum value of two values.
- `mode=MINIMUM` Passing the minimum value of two values.
- `mode=SUMMATION` Passing the summation of two values.
- `mode=MULTIPLICATION` Passing the multiplication of two values.

An `RuntimeError` will be raised if any of the parents does not exist. This operator does not support parameter *subPops* and does not output any information.

2.10.3 Class `SummaryTagger`

A summary tagger summarize values of one or more parental information field to another information field of an offspring. If mating is sexual, two sets of parental values will be involved.

class `SummaryTagger`(*mode=MEAN, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields=[]*)

Creates a summary tagger that summarize values of one or more parental information field (*infoFields*[:-1]) to an offspring information field (*infoFields*[-1]). A parameter *mode* specifies how to pass summarize parental values. More specifically,

- `mode=MEAN` Passing the average of values.
- `mode=MAXIMUM` Passing the maximum value of values.
- `mode=Minumum` Passing the minimum value of values.

- `mode=SUMMATION` Passing the sum of values.
- `mode=MULTIPLICATION` Passing the multiplication of values.

This operator does not support parameter *subPops* and does not output any information.

2.10.4 Class `ParentsTagger`

This tagging operator records the indexes of parents (relative to the parental generation) of each offspring in specified information fields (default to `father_idx` and `mother_idx`). Only one information field should be specified if an asexsual mating scheme is used so there is one parent for each offspring. Information recorded by this operator is intended to be used to look up parents of each individual in multi-generational Population.

class `ParentsTagger`(*begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields=["father_idx", "mother_idx"]*)

Create a parents tagger that records the indexes of parents of each offspring when it is applied to an offspring during-mating. If two information fields are specified (parameter *infoFields*, with default value `['father_idx', 'mother_idx']`), they are used to record the indexes of each individual's father and mother. Value `-1` will be assigned if any of the parent is missing. If only one information field is given, it will be used to record the index of the first valid parent (father if both parents are valid). This operator ignores parameters *stage*, *output*, and *subPops*.

2.10.5 Class `PedigreeTagger`

This tagging operator records the ID of parents of each offspring in specified information fields (default to `father_id` and `mother_id`). Only one information field should be specified if an asexsual mating scheme is used so there is one parent for each offspring. Information recorded by this operator is intended to be used to record full pedigree information of an evolutionary process.

class `PedigreeTagger`(*idField="ind_id", output="", outputFields=[], outputLoci=[], begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=["father_id", "mother_id"]*)

Create a pedigree tagger that records the ID of parents of each offspring when it is applied to an offspring during-mating. If two information fields are specified (parameter *infoFields*, with default value `['father_id', 'mother_id']`), they are used to record the ID of each individual's father and mother stored in the *idField* (default to `ind_id`) field of the parents. Value `-1` will be assigned if any of the parent is missing. If only one information field is given, it will be used to record the ID of the first valid parent (father if both pedigree are valid).

This operator by default does not send any output. If a valid output stream is given (should be in the form of `'>>filename'` so that output will be concatenated), this operator will output the ID of offspring, IDs of his or her parent(s), sex and affection status of offspring, and values at specified information fields (*outputFields*) and loci (*outputLoci*) in the format of `off_id father_id mother_id M/F A/U fields genotype.` `father_id` or `mother_id` will be ignored if only one parent is involved. This file format can be loaded using function `loadPedigree`.

Because only offspring will be outputed, individuals in the top-most ancestral generation will not be outputed. This is usually not a problem because individuals who have offspring in the next generation will be constructed by function `loadPedigree`, although their information fields and genotype will be missing. If you would like to create a file with complete pedigree information, you can apply this operator before evolution in the *initOps* parameter of functions `Population.evolve` or `Simulator.evolve`. This will output all individuals in the initial population (the top-most ancestral population after evolution) in the same format. Note that sex, affection status and genotype can be changed by other operators so this operator should usually be applied after all other operators are applied.

2.10.6 Class `PyTagger`

A Python tagger takes some information fields from both parents, pass them to a user provided Python function and set the offspring individual fields with the return values.

class `PyTagger`(*func=None, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields=[]*)

Create a hybrid tagger that provides an user provided function *func* with values of specified information fields (determined by parameter names of this function) of parents and assign corresponding information fields of offspring with its return value. If more than one parent are available, maternal values are passed after paternal values. For example, if a function `func(A, B)` is passed, this operator will send two tuples with parental values of information fields `'A'` and `'B'` to this function and assign its return values to fields `'A'` and `'B'` of each offspring. The return value of this function should be a list, although a single value will be accepted if only one information field is specified. This operator ignores parameters *stage*, *output* and *subPops*.

2.11 Statistics Calculation

2.11.1 Class `Stat`

Operator `Stat` calculates various statistics of the population being applied and sets variables in its local namespace. Other operators or functions can retrieve results from or evalulate expressions in this local namespace after `Stat` is applied.

class `Stat`(*popSize=False, numOfMales=False, numOfAffected=False, alleleFreq=[], heteroFreq=[], homoFreq=[], genoFreq=[], haploFreq=[], sumOfInfo=[], meanOfInfo=[], varOfInfo=[], maxOfInfo=[], minOfInfo=[], LD=[], association=[], neutrality=[], structure=[], HWE=[], vars=ALL_AVAIL, suffix="", output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a `Stat` operator that calculates specified statistics of a population when it is applied to this population. This operator is by default applied after mating (parameter *stage*) and can be applied to specified replicates (parameter *rep*) at specified generations (parameter *begin*, *end*, *step*, and *at*). This operator does not produce any output (ignore parameter *output*) after statistics are calculated. Instead, it stores results in the local namespace of the population being applied. Other operators can retrieve these variables or evalulate expression directly in this local namespace. Please refer to operator `BaseOperator` for a detailed explanation of these common operator parameters.

`Stat` supports parameter *subPops*. It usually calculate the same set of statistics for all subpopulations (`subPops=subPopList()`). If a list of (virtual) subpopulations are specified, statistics for only specified subpopulations will be calculated. However, different statistics treat this parameter differently and it is very important to check its reference before you use *subPops* for any statistics.

Calculated statistics are saved as variables in a population's local namespace. These variables can be numbers, lists or dictionaries and can be retrieved using functions `Population.vars()` or `Population.dvars()`. A special default dictionary (`defdict`) is used for dictionaries whose keys are determined dynamically. Accessing elements of such a dictionary with an invalid key will yield value 0 instead of a `KeyError`. If the same variables are calculated for one or more (virtual) subpopulation, the variables are stored in `vars()['subPop'][sp]['var']` where sp is a subpopulation ID (`sp`) or a tuple of virtual subpopulation ID (`(sp, vsp)`). `Population.vars(sp)` and `Population.dvars(sp)` provide shortcuts to these variables.

Operator *Stat* outputs a number of most useful variables for each type of statistic. For example, `alleleFreq` calculates both allele counts and allele frequencies and it by default sets variable `alleleFreq` (`dvars().alleleFreq`) for all or specified subpopulations. If this does not fit your need, you can use parameter *vars* to output additional parameters, or limit the output of existing parameters. More specifically, for this particular statistic, the available variables are `'alleleFreq'`, `'alleleNum'`, `'alleleFreq_sp'` (`'alleleFreq'` in each subpopulation), and `'alleleNum_sp'` (`'alleleNum'` in each subpopulation). You can set `vars=['alleleNum_sp']` to output only subpopulation specific allele count. An optional suffix (parameter *suffix*) can be used to append a suffix to default parameter names. This parameter can be used, for example, to calculate and store the same statistics for different subpopulations (e.g. pairwise `Fst`).

Operator `Stat` supports the following statistics:

popSize: If *popSize=True*, number of individuals in all or specified subpopulations (parameter *subPops*) will be set to the following variables:

- `popSize` (default): Number of individuals in all or specified subpopulations. Because *subPops* does not have to cover all individuals, it may not be the actual population size.
- `popSize_sp`: Size of (virtual) subpopulation `sp`.
- `subPopSize` (default): A list of (virtual) subpopulation sizes. This variable is easier to use than accessing popSize from each (virtual) subpopulation.

numOfMales: If *numOfMales=True*, number of male individuals in all or specified (virtual) subpopulations will be set to the following variables:

- `numOfMales` (default): Total number of male individuals in all or specified (virtual) subpopulations.
- `numOfFemales` (default): Total number of female individuals in all or specified (virtual) subpopulations.
- `propOfMales`: Proportion of male individuals.
- `propOfFemales`: Proportion of female individuals.
- `numOfMales_sp`: Number of male individuals in each (virtual) subpopulation.
- `numOfFemales_sp`: Number of female individuals in each (virtual) subpopulation.
- `propOfMales_sp`: Proportion of male individuals in each (virtual) subpopulation.
- `propOfFemales_sp`: Proportion of female individuals in each (virtual) subpopulation.

numOfAffected: If *numOfAffected=True*, number of affected individuals in all or specified (virtual) subpopulations will be set to the following variables:

- `numOfAffected` (default): Total number of affected individuals in all or specified (virtual) subpopulations.
- `numOfUnaffected` (default): Total number of unaffected individuals in all or specified (virtual) subpopulations.
- `propOfAffected`: Proportion of affected individuals.
- `propOfUnaffected`: Proportion of unaffected individuals.
- `numOfAffected_sp`: Number of affected individuals in each (virtual) subpopulation.
- `numOfUnaffected_sp`: Number of unaffected individuals in each (virtual) subpopulation.
- `propOfAffected_sp`: Proportion of affected individuals in each (virtual) subpopulation.
- `propOfUnaffected_sp`: Proportion of unaffected individuals in each (virtual) subpopulation.

alleleFreq: This parameter accepts a list of loci (loci indexes, names, or `ALL_AVAIL`), at which allele frequencies will be calculated. This statistic outputs the following variables, all of which are dictionary (with loci indexes as keys) of default dictionaries (with alleles as keys). For example, `alleleFreq[loc][a]` returns 0 if allele `a` does not exist.

- `alleleFreq` (default): `alleleFreq[loc][a]` is the frequency of allele `a` at locus for all or specified (virtual) subpopulations.
- `alleleNum` (default): `alleleNum[loc][a]` is the number of allele `a` at locus for all or specified (virtual) subpopulations.
- `alleleFreq_sp`: Allele frequency in each (virtual) subpopulation.
- `alleleNum_sp`: Allele count in each (virtual) subpopulation.

heteroFreq and **homoFreq**: These parameters accept a list of loci (by indexes or names), at which the number and frequency of homozygotes and/or heterozygotes will be calculated. These statistics are only available for diploid populations. The following variables will be outputted:

- `heteroFreq` (default for parameter *heteroFreq*): A dictionary of proportion of heterozygotes in all or specified (virtual) subpopulations, with loci indexes as dictionary keys.
- `homoFreq` (default for parameter *homoFreq*): A dictionary of proportion of homozygotes in all or specified (virtual) subpopulations.

- `heteroNum`: A dictionary of number of heterozygotes in all or specified (virtual) subpopulations.
- `homoNum`: A dictionary of number of homozygotes in all or specified (virtual) subpopulations.
- `heteroFreq_sp`: A dictionary of proportion of heterozygotes in each (virtual) subpopulation.
- `homoFreq_sp`: A dictionary of proportion of homozygotes in each (virtual) subpopulation.
- `heteroNum_sp`: A dictionary of number of heterozygotes in each (virtual) subpopulation.
- `homoNum_sp`: A dictionary of number of homozygotes in each (virtual) subpopulation.

genoFreq: This parameter accept a list of loci (by indexes or names) at which number and frequency of all genotypes are outputed as a dictionary (indexed by loci indexes) of default dictionaries (indexed by tuples of possible indexes). This statistic is available for all population types with genotype defined as ordered alleles at a locus. The length of genotype equals the number of homologous copies of chromosomes (ploidy) of a population. Genotypes for males or females on sex chromosomes or in haplodiploid populations will have different length. Because genotypes are ordered, `(1, 0)` and `(0, 1)` (two possible genotypes in a diploid population) are considered as different genotypes. This statistic outputs the following variables:

- `genoFreq` (default): A dictionary (by loci indexes) of default dictionaries (by genotype) of genotype frequencies. For example, `genoFreq[1][(1, 0)]` is the frequency of genotype (1, 0) at locus 1.
- `genoNum` (default): A dictionary of default dictionaries of genotype counts of all or specified (virtual) subpopulations.
- `genoFreq_sp`: genotype frequency in each specified (virtual) subpopulation.
- `genoFreq_sp`: genotype count in each specified (virtual) subpopulation.

haploFreq: This parameter accepts one or more lists of loci (by index) at which number and frequency of haplotypes are outputted as default dictionaries. `[(1,2)]` can be abbreviated to `(1,2)`. For example, using parameter `haploFreq=(1,2,4)`, all haplotypes at loci `1`, `2` and `4` are counted. This statistic saves results to dictionary (with loci index as keys) of default dictionaries (with haplotypes as keys) such as `haploFreq[(1,2,4)][(1,1,0)]` (frequency of haplotype `(1,1,0)` at loci `(1,2,3)`). This statistic works for all population types. Number of haplotypes for each individual equals to his/her ploidy number. Haplodiploid populations are supported in the sense that the second homologous copy of the haplotype is not counted for male individuals. This statistic outputs the following variables:

- `haploFreq` (default): A dictionary (with tuples of loci indexes as keys) of default dictionaries of haplotype frequencies. For example, `haploFreq[(0, 1)][(1,1)]` records the frequency of haplotype `(1,1)` at loci `(0, 1)` in all or specified (virtual) subpopulations.
- `haploNum` (default): A dictionary of default dictionaries of haplotype counts in all or specified (virtual) subpopulations.
- `haploFreq_sp`: Halptype frequencies in each (virtual) subpopulation.
- `haploNum_sp`: Halptype count in each (virtual) subpopulation.

sumOfinfo, **meanOfInfo**, **varOfInfo**, **maxOfInfo** and **minOfInfo**: Each of these five parameters accepts a list of information fields. For each information field, the sum, mean, variance, maximum or minimal (depending on the specified parameter(s)) of this information field at iddividuals in all or specified (virtual) subpopulations will be calculated. The results will be put into the following population variables:

- `sumOfInfo` (default for *sumOfInfo*): A dictionary of the sum of specified information fields of individuals in all or specified (virtual) subpopulations. This dictionary is indexed by names of information fields.
- `meanOfInfo` (default for *meanOfInfo*): A dictionary of the mean of information fields of all individuals.
- `varOfInfo` (default for *varOfInfo*): A dictionary of the sample variance of information fields of all individuals.
- `maxOfInfo` (default for *maxOfInfo*): A dictionary of the maximum value of information fields of all individuals.

- `minOfInfo` (default for *minOfInfo*): A dictionary of the minimal value of information fields of all individuals.
- `sumOfInfo_sp`: A dictionary of the sum of information fields of individuals in each subpopulation.
- `meanOfInfo_sp`: A dictionary of the mean of information fields of individuals in each subpopulation.
- `varOfInfo_sp`: A dictionary of the sample variance of information fields of individuals in each subpopulation.
- `maxOfInfo_sp`: A dictionary of the maximum value of information fields of individuals in each subpopulation.
- `minOfInfo_sp`: A dictionary of the minimal value of information fields of individuals in each subpopulation.

LD: Parameter `LD` accepts one or a list of loci pairs (e.g. `LD=[[0,1], [2,3]]`) with optional primary alleles at both loci (e.g. `LD=[0,1,0,0]`). For each pair of loci, this operator calculates linkage disequilibrium and optional association statistics between two loci. When primary alleles are specified, signed linkage disequilibrium values are calculated with non-primary alleles are combined. Otherwise, absolute values of diallelic measures are combined to yield positive measure of LD. Association measures are calculated from a `m` by `n` contigency of haplotype counts (`m=n=2` if primary alleles are specified). Please refer to the simuPOP user's guide for detailed information. This statistic sets the following variables:

- `LD` (default) Basic LD measure for haplotypes in all or specified (virtual) subpopulations. Signed if primary alleles are specified.
- `LD_prime` (default) Lewontin's D' measure for haplotypes in all or specified (virtual) subpopulations. Signed if primary alleles are specified.
- `R2` (default) Correlation LD measure for haplotypes in all or specified (virtual) subpopulations.
- `LD_ChiSq` ChiSq statistics for a contigency table with frequencies of haplotypes in all or specified (virtual) subpopulations.
- `LD_ChiSq_p` Single side p-value for the ChiSq statistic. Degrees of freedom is determined by number of alleles at both loci and the specification of primary alleles.
- `CramerV` Normalized ChiSq statistics.
- `LD_sp` Basic LD measure for haplotypes in each (virtual) subpopulation.
- `LD_prime_sp` Lewontin's D' measure for haplotypes in each (virtual) subpopulation.
- `R2_sp` R2 measure for haplotypes in each (virtual) subpopulation.
- `LD_ChiSq_sp` ChiSq statistics for each (virtual) subpopulation.
- `LD_ChiSq_p_sp` p value for the ChiSq statistics for each (virtual) subpopulation.
- `CramerV_sp` Cramer V statistics for each (virtual) subpopulation.

association: Parameter `association` accepts a list of loci, which can be a list of indexes, names, or `ALL_AVAIL`. At each locus, one or more statistical tests will be performed to test association between this locus and individual affection status. Currently, simuPOP provides the following tests:

- An allele-based Chi-square test using alleles counts. This test can be applied to loci with more than two alleles, and to haploid populations.
- A genotype-based Chi-square test using genotype counts. This test can be applied to loci with more than two alleles (more than 3 genotypes) in diploid populations. `aA` and `Aa` are considered to be the same genotype.
- A genotype-based Cochran-Armitage trend test. This test can only be applied to diallelic loci in diploid populations. A codominant model is assumed.

This statistic sets the following variables:

- `Allele_ChiSq` A dictionary of allele-based Chi-Square statistics for each locus, using cases and controls in all or specified (virtual) subpopulations.

- `Allele_ChiSq_p` (default) A dictionary of *p-values* of the corresponding Chi-square statistics.
- `Geno_ChiSq` A dictionary of genotype-based Chi-Square statistics for each locus, using cases and controls in all or specified (virtual) subpopulations.
- `Geno_ChiSq_p` A dictionary of *p-values* of the corresponding genotype-based Chi-square test.
- `Armitage_p` A dictionary of *p-values* of the Cochran-Armitage tests, using cases and controls in all or specified (virtual) subpopulations.
- `Allele_ChiSq_sp` A dictionary of allele-based Chi-Square statistics for each locus, using cases and controls from each subpopulation.
- `Allele_ChiSq_p_sp` A dictionary of p-values of allele-based Chi-square tests, using cases and controls from each (virtual) subpopulation.
- `Geno_ChiSq_sp` A dictionary of genotype-based Chi-Square tests for each locus, using cases and controls from each subpopulation.
- `Geno_ChiSq_p_sp` A dictionary of p-values of genotype-based Chi-Square tests, using cases and controls from each subpopulation.
- `Armitage_p_sp` A dictionary of *p-values* of the Cochran- Armitage tests, using cases and controls from each subpopulation.

neutrality: This parameter performs neutrality tests (detection of natural selection) on specified loci, which can be a list of loci indexes, names or `ALL_AVAIL`. It currently only outputs *Pi*, which is the average number of pairwise difference between loci. This statistic outputs the following variables:

- `Pi` Mean pairwise difference between all sequences from all or specified (virtual) subpopulations.
- `Pi_sp` Mean paiewise difference between all sequences in each (virtual) subpopulation.

structure: Parameter `structure` accepts a list of loci at which statistics that measure population structure are calculated. *structure* accepts a list of loci indexes, names or `ALL_AVAIL`. This parameter currently supports the following statistics:

- Weir and Cockerham's Fst (1984). This is the most widely used estimator of Wright's fixation index and can be used to measure Population differentiation. However, this method is designed to estimate Fst from samples of larger populations and might not be appropriate for the calculation of Fst of large populations.
- Nei's Gst (1973). The Gst estimator is another estimator for Wright's fixation index but it is extended for multi-allele (more than two alleles) and multi-loci cases. This statistics should be used if you would like to obtain a *true* Fst value of a large Population.
- `F_st` (default) The WC84 *Fst* statistic estimated for all specified loci.
- `F_is` The WC84 *Fis* statistic estimated for all specified loci.
- `F_it` The WC84 *Fit* statistic estimated for all specified loci.
- `f_st` A dictionary of locus level WC84 *Fst* values.
- `f_is` A dictionary of locus level WC84 *Fis* values.
- `f_it` A dictionary of locus level WC84 *Fit* values.
- `G_st` Nei's Gst statistic estimated for all specified loci.
- `g_st` A dictionary of Nei's Gst statistic estimated for each locus.

HWE: Parameter `HWE` accepts a list of loci at which exact two-side tests for Hardy-Weinberg equilibrium will be performed. This statistic is only available for diallelic loci in diploid populations. *HWE* can be a list of loci indexes, names or `ALL_AVAIL`. This statistic outputs the following variables:

- `HWE` (default) A dictionary of p-values of HWE tests using genotypes in all or specified (virtual) subpopulations.
- `HWE_sp` A dictionary of p-values of HWS tests using genotypes in each (virtual) subpopulation.

2.12 Conditional operators

2.12.1 Class `IfElse`

This operator uses a condition, which can be a fixed condition, an expression or a user-defined function, to determine which operators to be applied when this operator is applied. A list of if-operators will be applied when the condition is `True`. Otherwise a list of else-operators will be applied.

class `IfElse`(*cond, ifOps=[], elseOps=[], output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a conditional operator that will apply operators *ifOps* if condition *cond* is met and *elseOps* otherwise. If a Python expression (a string) is given to parameter *cond*, the expression will be evalulated in each population's local namespace when this operator is applied. When a Python function is specified, it accepts parameter `pop` when it is applied to a population, and one or more parameters `pop`, `off`, `dad` or `mom` when it is applied during mating. The return value of this function should be `True` or `False`. Otherwise, parameter *cond* will be treated as a fixed condition (converted to `True` or `False`) upon which one set of operators is always applied. The applicability of *ifOps* and *elseOps* are controlled by parameters *begin*, *end*, *step*, *at* and *rep* of both the `IfElse` operator and individual operators but *ifOps* and *elseOps* opeartors does not support negative indexes for replicate and generation numbers.

2.12.2 Class `TerminateIf`

This operator evaluates an expression in a population's local namespace and terminate the evolution of this population, or the whole simulator, if the return value of this expression is `True`. Termination caused by an operator will stop the execution of all operators after it. The generation at which the population is terminated will be counted in the *evolved generations* (return value from `Simulator::evolve`) if termination happens after mating.

class `TerminateIf`(*condition="", stopAll=False, message="", output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a terminator with an expression *condition*, which will be evalulated in a population's local namespace when the operator is applied to this population. If the return value of *condition* is `True`, the evolution of the population will be terminated. If *stopAll* is set to `True`, the evolution of all replicates of the simulator will be terminated. If this operator is allowed to write to an *output* (default to ""), the generation number, proceeded with an optional *message*.

2.12.3 Class `DiscardIf`

This operator discards individuals according to either an expression that evaluates according to individual information field, or a Python function that accepts individual and its information fields.

class `DiscardIf`(*cond, exposeInd="", output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create an operator that discard individuals according to an expression or the return value of a Python function (parameter *cond*). This operator can be applied to a population before or after mating, or to offspring during mating. If an expression is passed to *cond*, it will be evalulated with each individual's information fields (see operator `InfoEval` for details). If *exposeInd* is non-empty, individuals will be available for evaluation in the expression as an variable with name spacied by *exposeInd*. If the expression is evaluated to be `True`, individuals (if applied before or after mating) or offspring (if applied during mating) will be removed or discard. If a function is passed to *cond*, it should accept paramters *ind* and *pop* or names of information fields when it is applied to a population (pre or post mating), or parameters *off*, *dad*, *mom*, *pop* (parental population), or names of information fields if the operator is applied during mating. Individuals will be discarded if this function returns `True`. A constant expression (e.g. `True`) is also acceptable). Because this operator supports parameter *subPops*, only individuals belonging to specified (virtual) subpopulations will be screened.

2.13 The Python operator

2.13.1 Class `PyOperator`

An operator that calls a user-defined function when it is applied to a population (pre- or post-mating) or offsprings (during-mating). The function can have have parameters `pop` when the operator is applied pre- or post-mating, `pop, off, dad, mom` when the operator is applied during-mating. An optional parameter can be passed if parameter *param* is given. In the during-mating case, parameters `pop`, `dad` and `mom` can be ignored if *offspringOnly* is set to `True`.

class `PyOperator`(*func, param=None, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a pure-Python operator that calls a user-defined function when it is applied. If this operator is applied before or after mating, your function should have form `func(pop)` or `func(pop, param)` where `pop` is the population to which the operator is applied, `param` is the value specified in parameter *param*. `param` will be ignored if your function only accepts one parameter. Althernatively, the function should have form `func(ind)` with optional parameters `pop` and `param`. In this case, the function will be called for all individuals, or individuals in subpopulations *subPops*. Individuals for which the function returns `False` will be removed from the population. This operator can therefore perform similar functions as operator `DiscardIf`.

If this operator is applied during mating, your function should accept parameters `pop`, `off` (or `ind`), `dad`, `mom` and `param` where `pop` is the parental population, and `off` or `ind`, `dad`, and `mom` are offspring and their parents for each mating event, and `param` is an optional parameter. If *subPops* are provided, only offspring in specified (virtual) subpopulations are acceptable.

This operator does not support parameters *output*, and *infoFields*. If certain output is needed, it should be handled in the user defined function *func*. Because the status of files used by other operators through parameter *output* is undetermined during evolution, they should not be open or closed in this Python operator.

2.14 Miscellaneous operators

2.14.1 Class `NoneOp`

This operator does nothing when it is applied to a population. It is usually used as a placeholder when an operator is needed syntactically.

class `NoneOp`(*output=">", begin=0, end=0, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a `NoneOp`.

2.14.2 Class `Dumper`

This operator dumps the content of a population in a human readable format. Because this output format is not structured and can not be imported back to simuPOP, this operator is usually used to dump a small population to a terminal for demonstration and debugging purposes.

class `Dumper`(*genotype=True, structure=True, ancGens=UNSPECIFIED, width=1, max=100, loci=[], output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a operator that dumps the genotype structure (if *structure* is `True`) and genotype (if *genotype* is `True`) to an *output* (default to standard terminal output). Because a population can be large, this operator will only output the first 100 (parameter *max*) individuals of the present generation (parameter *ancGens*). All loci will be outputed unless parameter *loci* are used to specify a subset of loci. If a list of (virtual) subpopulations are specified, this operator will only output individuals in these outputs. Please refer to class `BaseOperator` for a detailed explanation for common parameters such as *output* and *stage*.

2.14.3 Class `SavePopulation`

An operator that save populations to specified files.

class `SavePopulation`(*output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create an operator that saves a population to *output* when it is applied to the population. This operator supports all output specifications (", `'filename'`, `'filename'` prefixed by one or more '>' characters, and `'!expr'`) but output from different operators will always replace existing files (effectively ignore '>' specification). Parameter *subPops* is ignored. Please refer to class `BaseOperator` for a detailed description about common operator parameters such as *stage* and *begin*.

2.14.4 Class `Pause`

This operator pauses the evolution of a simulator at given generations or at a key stroke. When a simulator is stopped, you can go to a Python shell to examine the status of an evolutionary process, resume or stop the evolution.

class `Pause`(*stopOnKeyStroke=False, prompt=True, output=">", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create an operator that pause the evolution of a population when it is applied to this population. If *stopOnKeyStroke* is `False` (default), it will always pause a population when it is applied, if this parameter is set to `True`, the operator will pause a population if *any* key has been pressed. If a specific character is set, the operator will stop when this key has been pressed. This allows, for example, the use of several pause operators to pause different populations.

After a population has been paused, a message will be displayed (unless *prompt* is set to `False`) and tells you how to proceed. You can press `'s'` to stop the evolution of this population, `'S'` to stop the evolution of all populations, or `'p'` to enter a Python shell. The current population will be available in this Python shell as `"pop_X_Y"` when `X` is generation number and `Y` is replicate number. The evolution will continue after you exit this interactive Python shell.

Note Ctrl-C will be intercepted even if a specific character is specified in parameter *stopOnKeyStroke*.

2.14.5 Class `TicToc`

This operator, when called, output the difference between current and the last called clock time. This can be used to estimate execution time of each generation. Similar information can also be obtained from `turnOnDebug("DBG_PROFILE")`, but this operator has the advantage of measuring the duration between several generations by setting `step` parameter. As an advanced feature that mainly used for performance testing, this operator accepts a parameter *stopAfter* (seconds), and will stop the evolution of a population if the overall time exceeds *stopAfter*. Note that elapsed time is only checked when this operator is applied to a population so it might not be able to stop the evolution process right after *stopAfter* seconds. This operator can also be applied during mating. Note that to avoid excessive time checking, this operator does not always check system time accurately.

class `TicToc`(*output=">", stopAfter=0, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a `TicToc` operator that outputs the elapsed since the last time it was applied, and the overall time since the first time this operator is applied.

2.15 Function form of operators

2.15.1 Function `acgtMutate`

`acgtMutate`(*pop, *args, **kwargs*)

Function form of operator `AcgtMutator`

2.15.2 Function `contextMutate`

contextMutate(*pop, *args, **kwargs*)
Function form of operator `ContextMutator`

2.15.3 Function `discardIf`

discardIf(*pop, *args, **kwargs*)
Apply operator `DiscardIf` to population *pop* to remove individuals according to an expression or a Python function.

2.15.4 Function `dump`

dump(*pop, *args, **kwargs*)
Apply operator `Dumper` to population *pop*.

2.15.5 Function `infoEval`

infoEval(*pop, *args, **kwargs*)
Evaluate *expr* for each individual, using information fields as variables. Please refer to operator `InfoEval` for details.

2.15.6 Function `infoExec`

infoExec(*pop, *args, **kwargs*)
Execute *stmts* for each individual, using information fields as variables. Please refer to operator `InfoExec` for details.

2.15.7 Function `initGenotype`

initGenotype(*pop, *args, **kwargs*)
Apply operator `InitGenotype` to population *pop*.

2.15.8 Function `initInfo`

initInfo(*pop, *args, **kwargs*)
Apply operator `InitInfo` to population *pop*.

2.15.9 Function `initSex`

initSex(*pop, *args, **kwargs*)
Apply operator `InitSex` to population *pop*.

2.15.10 Function `kAlleleMutate`

kAlleleMutate(*pop, *args, **kwargs*)
Function form of operator `KAlleleMutator`

2.15.11 Function `maPenetrance`

maPenetrance(*pop, loci, penetrance, wildtype=0, ancGens=True, *args, **kwargs*)

Apply opertor `MaPenetrance` to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

2.15.12 Function `mapPenetrance`

mapPenetrance(*pop, loci, penetrance, ancGens=True, *args, **kwargs*)

Apply opertor `MapPenetrance` to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

2.15.13 Function `matrixMutate`

matrixMutate(*pop, *args, **kwargs*)

Function form of operator `MatrixMutator`

2.15.14 Function `mergeSubPops`

mergeSubPops(*pop, *args, **kwargs*)

Merge subpopulations *subPops* of population *pop* into a single subpopulation. Please refer to the operator form of this funciton (`MergeSubPops`) for details

2.15.15 Function `migrate`

migrate(*pop, *args, **kwargs*)

Function form of operator `Migrator`.

2.15.16 Function `mixedMutate`

mixedMutate(*pop, *args, **kwargs*)

Function form of operator `MixedMutator`

2.15.17 Function `mlPenetrance`

mlPenetrance(*pop, ops, mode, ancGens=True, *args, **kwargs*)

Apply opertor `MapPenetrance` to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

2.15.18 Function `pointMutate`

pointMutate(*pop, *args, **kwargs*)

Function form of operator `PointMutator`

2.15.19 Function `pyEval`

pyEval(*pop, *args, **kwargs*)

Evaluate statements *stmts* (optional) and expression *expr* in population *pop*'s local namespace and return the result of *expr*. If *exposePop* is given, population *pop* will be exposed in its local namespace as a variable with a name specified by *exposePop*. Unlike its operator counterpart, this function returns the result of *expr* rather than writting it to an output.

2.15.20 Function `pyExec`

pyExec(*pop, *args, **kwargs*)

Execute *stmts* in population *pop*'s local namespace.

2.15.21 Function `pyMutate`

pyMutate(*pop, *args, **kwargs*)

Function form of operator `PyMutator`

2.15.22 Function `pyPenetrance`

pyPenetrance(*pop, func, loci=[], ancGens=True, *args, **kwargs*)

Apply opertor `MapPenetrance` to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

2.15.23 Function `pyQuanTrait`

pyQuanTrait(*pop, func, loci=[], ancGens=True, *args, **kwargs*)

Apply opertor `PyQuanTrait` to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

2.15.24 Function `resizeSubPops`

resizeSubPops(*pop, *args, **kwargs*)

Resize subpopulations *subPops* of population *pop* into new sizes *size*. Individuals will be added or removed accordingly. Please refer to the operator form of this funciton (`ResizeSubPops`) for details

2.15.25 Function `snpMutate`

snpMutate(*pop, *args, **kwargs*)

Function form of operator `SNPMutator`

2.15.26 Function `splitSubPops`

splitSubPops(*pop, *args, **kwargs*)

Split subpopulations (*subPops*) of population *pop* according to either *sizes* or *proportions* of the resulting subpopulations, or an information field. Please refer to the operator form of this function (`splitSubPop`) for details.

2.15.27 Function `stat`

stat(*pop, *args, **kwargs*)

Apply operator `Stat` with specified parameters to population *pop*. Resulting statistics could be accessed from the local namespace of `pop` using functions `pop.vars()` or `pop.dvars()`

2.15.28 Function `stepwiseMutate`

stepwiseMutate(*pop, *args, **kwargs*)

Function form of operator `StepwiseMutator`

2.15.29 Function `tagID`

tagID(*pop, reset=False, *args, **kwargs*)

Apply operator `IdTagger` to population *pop* to assign a unique ID to all individuals in the population. Individuals ID will starts from a system wide index. You can reset this start ID using parameter `reset` which can be `True` (reset to 1) or a non-negative number (start from this number).

Chapter 3

Utility Modules

3.1 Module `simuOpt`

Module `simuOpt` provides a function `simuOpt.setOptions` to control which simuPOP module to load, and how it is loaded, and a class `simuOpt.Params` that helps users manage simulation parameters.

When simuPOP is loaded, it checkes a few environmental variables (`SIMUOPTIMIZED`, `SIMUALLELETYPE`, and `SIMUDEBUG`) to determine which simuPOP module to load, and how to load it. More options can be set using the `simuOpt.setOptions` function. For example, you can suppress the banner message when simuPOP is loaded and require a minimal version of simuPOP for your script. simuPOP recognize the following commandline arguments

`--optimized`: Load the optimized version of a simuPOP module.

`--gui=None|batch|interactive|True|wxPython|Tkinter`: Whether or not use a graphical toolkit and which one to use. `--gui=batch` is usually used to run a script in batch mode (do not start a parameter input dialog and use all default values unless a parameter is specified from command line or a configuraiton file. If `--gui=interactive`, an interactive shell will be used to solicit input from users. Otherwise, simuPOP will try to use a graphical parameter input dialog, and falls to an interactive mode when no graphical Toolkit is available. Please refer to parameter `gui` for `simuOpt.setOptions` for details.

class `params.Params` provides a powerful way to handle commandline arguments. Briefly speaking, a `Params` object can be created from a list of parameter specification dictionaries. The parameters are then become attributes of this object. A number of functions are provided to determine values of these parameters using commandline arguments, a configuration file, or a parameter input dialog (using `Tkinter` or `wxPython`). Values of these parameters can be accessed as attributes, or extracted as a list or a dictionary. Note that the `Params.getParam` function automatically handles the following commandline arguments.

`-h` or `--help`: Print usage message.

`--config=configFile`: Read parameters from a configuration file *configFile*.

3.1.1 Function `setOptions`

`setOptions`(*alleleType=None, optimized=None, gui=None, quiet=None, debug=None, version=None, revision=None, numThreads=None*)

Set options before simuPOP is loaded to control which simuPOP module to load, and how the module should be loaded.

alleleType: Use the standard, binary or long allele version of the simuPOP module if `alleleType` is set to 'short', 'binary', or 'long' respectively. If this parameter is not set, this function will try to get its value from environmental variable `SIMUALLELETYPE`. The standard (short) module will be used if the environmental

variable is not defined.

optimized: Load the optimized version of a module if this parameter is set to `True` and the standard version if it is set to `False`. If this parameter is not set (`None`), the optimized version will be used if environmental variable `SIMUOPTIMIZED` is defined. The standard version will be used otherwise.

gui: Whether or not use graphical user interfaces, which graphical toolkit to use and how to process parameters in non-GUI mode. If this parameter is `None` (default), this function will check environmental variable `SIMUGUI` or commandline option `--gui` for a value, and assume `True` if such an option is unavailable. If `gui=True`, simuPOP will use `wxPython`-based dialogs if `wxPython` is available, and use `Tkinter`-based dialogs if `Tkinter` is available and use an interactive shell if no graphical toolkit is available. `gui='Tkinter'` or `'wxPython'` can be used to specify the graphical toolkit to use. If `gui='interactive'`, a simuPOP script prompt users to input values of parameters. If `gui='batch'`, default values of unspecified parameters will be used. In any case, commandline arguments and a configuration file specified by parameter --config will be processed. This option is usually left to `None` so that the same script can be run in both GUI and batch mode using commandline option `--gui`.

quiet: If set to `True`, suppress the banner message when a simuPOP module is loaded.

debug: A list of debug code (as string) that will be turned on when simuPOP is loaded. If this parameter is not set, a list of comma separated debug code specified in environmental variable `SIMUDEBUG`, if available, will be used. Note that setting `debug=[]` will remove any debug code that might have been by variable `SIMUDEBUG`.

version: A version string (e.g. 1.0.0) indicating the required version number for the simuPOP module to be loaded. simuPOP will fail to load if the installed version is older than the required version.

revision: Obsolete with the introduction of parameter version.

numThreads: Number of Threads that will be used to execute a simuPOP script. The values can be a positive number (number of threads) or 0 (all available cores of the computer, or whatever number set by environmental variable `OMP_NUM_THREADS`). If this parameter is not set, the number of threads will be set to 1, or a value set by environmental variable `OMP_NUM_THREADS`.

3.1.2 Class `Params`

class Params provides a uniform interface for simuPOP scripts to handle parameters. It allows users to get parameters from command line options, a configuration file, a parameter input dialog (*tkInter* or *wxPython*) or from interative input. This class provides parameter validation, conversion and and some utility functions to print, save and restore parameters.

A Params object accepts a parameter specification list that consists of dictionaries with pre-defined keys. Each item defines an option in terms of command line option, entry name in a configuration file, label in a parameter input dialog, acceptable types, validation rules and a default value. The following keys are currently supported:

name: Long command line option name. For example `'version'` checks the presence of argument `--version`. For example, `'mu'` matches command line option `--mu=0.001` or `--mu 0.001`. **This item defines the name of an option and cannot be ignored**. An options that does not expect a value is identified as a single BooleanType in allowedTypes or a default value `False` when no allowedTypes is defined. Such a value should have default value `False` and the presence of this argument in the command line (e.g. `--verbose`) change it to `True`.

label: The label of the input field in a parameter input dialog. It will also be used as the prompt for this option during interactive parameter input. **Options without a label will not be displayed in the parameter input dialog and will not be saved to a configuration file**. A typical example of such an option is `--version`.

default: Default value for this parameter. It is used as the default value in the parameter input dialog, and as the option value when a user presses `Enter` directly during interactive parameter input. **A default value is required for all options**.

description: A long description of this parameter. This description will be put into the usage information, and as parameter tooltip in the parameter input dialog. This string will be reformatted when it is written to a usage

string (remove newlines and extra spaces and re-indent), with the exception that **lines with 'l' as the first non-space/tab character will be outputed as is without the leading 'l' symbol**.

allowedTypes: A list of acceptable types of this option. class `Params` will try to convert user input to these types. For example, if `allowedTypes` is `list` or `tuple` and the user's input is a scalar, the input will be converted to a list automatically. An option will not be accepted if such conversion fails. If this item is not specified, the type of the default value will be used. If only one type is acceptable, a single value can be used as input (ignore []).

validator: An expression or a function to validate the parameter. If an expression (a string) is used, it will be evaluated using current values of parameters as inputs. If a function is specified, it will be called with the value of the parameter. The option will not be accepted if the expression is evalulated as "False" or if the function returns `False`. This module defines a large number of such validation functions but user defined functions are also acceptable.

type: Type of the input parameter. Which can be a datatype (e.g. `bool`), a list of acceptable types (e.g. `(int, long)`), or one of `'chooseOneOf', values`, `'chooseFrom', values`, `'filename'` (a valid filename), `'dirname'` (a valid directory name), `'integer'` (`int` or `long`), `'integers'` (list of integers), `'number'` (`int`, `long` or `float`), `'numbers'` (list of numbers), `'string'` (the same as `str`), `'strings'` (list of strings). These types will advise class `simuOpt.Params` how to accept parameters, and how to convert user input to appropriate types. For example, a file browser will be used to browse for a valid filename in a parameter input dialog (if `--gui=True`) for a parameter of type `'filename'`, and input `'52'` will be automatically converted to `(52,)` for a parameter of type `'integers'`.

separator: This item specifies a separator (group header) in the parameter input dialog. All other fields are ignored.

arg, longarg, useDefault, chooseFrom, chooseOneOf, allowedTypes: These parameters are deprecated because of the introduction of the 'name', 'gui_type' keys and the 'batch' mode.

Not all keys need to be specified in each option description. Missing values are handled using some internal rules. For example, items without a `label` will not be displayed on the parameter dialog. This will effectively *hide* a parameter although users who know this parameter can set it using command line options.

The `Params.Params` class defines a number of functions to collect, validate, and manipulate parameters using this parameter specification list.

As a shortcut to create a Params object with a number of attributes, a Params object can be created with additional `key=value` pairs that could be assessed as attributes. This is used to create a `Params` object in which *parameters* are assigned directly.

class `Params`(*options=[], doc='', details='', **kwargs*)

Create a `Params` oject using a list of parameter specification dictionaries *options*. Additional *doc* and *details* can be specified which will be displayed as script summary (on the top of a parameter input dialog) and script introduction (the first part of a help message), respectively. Additional attributes could be assigned to a `Params` object as keyword arguments. Note that it is customary to use module document (the first string object in a Python script) as *details*, using parameter `details=__doc__`.

`addOption`(*name='', default=None, **kwargs*)

Append an entry to the parameter specification list. Dictionary entries should be specified as keyword arguments such as `name='option'`. More specifically, you can specify parameters `name` (required), `label`, `default` (required), `description`, `validator`, `type`, and `separator`. This option will have a name specified by `name` and an initial default value specified by `default`.

`asDict`()

Return parameters as a dictionary.

`asList`()

Return parameters as a list.

`getParam`(*gui=None, nCol=None, configFile=None, args=None, checkArgs=True*)

Get parameters from commandline option, configuration file, a parameter input dialog and from interactive user input.

gui: Whether or not use a dialog and which graphical toolkit to use. Global gui setting is used by default but you can also set this parameter to `True`, `False`, `Tkinter` or `wxPython` to override the global setting.

nCol: Number of columns in the parameter input dialog. This is usual determine automatically depending on the number of options.

configFile: Configuration file from which to load values of parameters. If unspecified, it will be determined from command line option `--config`.

args: Command line arguments are obtained from `sys.argv` unless a list of options are provided in this argument.

checkArgs: This function by default checks if all commandline arguments have been processed, you can set `chekArgs` to `False` if some of the arguments are intended to be processed separately.

guiGetParam(*nCol=None, gui=None*)

Get parameter from a `Tkinter` or `wxPython` dialog. The parameter will try to arrange parameters optimaly but you can also set the number of columns using parameter *nCol*. If both GUI toolkits are available, `wxPython` will be used unless *gui* is set to `Tkinter`. If none of the toolkits are available, this function will raise an `ImportError`.

If `Params.valueValidFile` or `Params.valueValidDir` is used to validate a parameter, double click the text input box of this parameter will open a file or directory browse dialog.

loadConfig(*file, params=[]*)

Load configuration from a file. If a list of parameters are specified in *params*, only these parameters will be processed.

processArgs(*args=None, params=[]*)

Try to get parameters from a list of arguments *args* (default to `sys.argv`). If `-h` or `--help` is in *args*, this function prints out a usage message and returns `False`. If a list of parameters are specified in *params*, only these parameters will be processed.

saveConfig(*file, params=[]*)

Write a configuration file to *file*. This file can be later read with command line option `-c` or `--config`. All parameters with a `label` entry are saved unless a list of parameters are specified in *params*. In addition to parameter definitions, command lines options to specify the same set of parameters are saved to the configuration file.

termGetParam(*params=[]*)

Get parameters from interactive user input. By default, all parameters are processed unless one of the following conditions is met:

1. Parameter without a label
2. Parameter with `useDefault` set to `True` (deprecated)
3. Parameter that have been determined from command line options or a configuration file
4. Parameter that have been determined by a previous call of this function.

If a list of parameters are given in *params*, these parameters are processed regardless the mentioned conditions.

usage(*usage='usage: %prog [-opt [arg] | –opt [=arg]] ...'*)

Reutn the usage message from the option description list. `'%prog'` in parameter *usage* will be replaced by `os.path.basename(sys.argv[0])`.

3.1.3 Function `param`

param(*name='', default=None, **kwargs*)

A simple wrapper that allows the specification of a parameter using a function instead of a dictionary. Please refer to class `simuOpt.Params` for allowed keyword arguments and their meanings.

3.1.4 Function `valueNot`

valueNot(*t*)

Return a function that returns true if passed option does not equal t, or does not passes validator t

3.1.5 Function `valueOr`

valueOr(*t1, t2*)

Return a function that returns true if passed option passes validator t1 or t2

3.1.6 Function `valueAnd`

valueAnd(*t1, t2*)

Return a function that returns true if passed option passes validator t1 and t2

3.1.7 Function `valueOneOf`

valueOneOf(**args*)

Return a function that returns true if passed option is one of the parameters, or one of the values in the only parameter

3.1.8 Function `valueTrueFalse`

valueTrueFalse()

Return a function that returns true if passed option is True or False

3.1.9 Function `valueBetween`

valueBetween(*a, b*)

Return a function that returns true if passed option is between value a and b (a and b included)

3.1.10 Function `valueGT`

valueGT(*a*)

Return a function that returns true if passed option is greater than a

3.1.11 Function `valueGE`

valueGE(*a*)

Return a function that returns true if passed option is greater than or equal to a

3.1.12 Function `valueLT`

valueLT(*a*)

Return a function that returns true if passed option is less than a

3.1.13 Function `valueLE`

valueLE(*a*)

Return a function that returns true if passed option is less than or equal to a

3.1.14 Function `valueEqual`

valueEqual(*a*)

Return a function that returns true if passed option equals a

3.1.15 Function `valueNotEqual`

valueNotEqual(*a*)

Return a function that returns true if passed option does not equal a

3.1.16 Function `valueIsNum`

valueIsNum()

Return a function that returns true if passed option is a number (int, long or float)

3.1.17 Function `valueIsInteger`

valueIsInteger()

Return a function that returns true if passed option is an integer (int, long)

3.1.18 Function `valueIsList`

valueIsList(*size=None*)

Return a function that returns true if passed option is a sequence. If a `size` is given, the sequence must have the specified size (e.g. `size=3`), or within the range of sizes (e.g. `size=[1, 5]`). A `None` can be used as unspecified lower or upper bound.

3.1.19 Function `valueListOf`

valueListOf(*t, size=None*)

Return a function that returns true if passed option val is a list of type `t` if `t` is a type, if `v` is one of `t` if `t` is a list, or if `v` passes test `t` if `t` is a validator (a function). If a `size` is given, the sequence must have the specified size (e.g. `size=3`), or within the range of sizes (e.g. `size=[1, 5]`). A `None` can be used as unspecified lower or upper bound.

3.1.20 Function `valueSumTo`

valueSumTo(*a, eps=1e-07*)

Return a function that returns true if passed option sum up to a. An eps value can be specified to allowed for numerical error.

3.1.21 Function `valueValidDir`

valueValidDir()
Return a function that returns true if passed option val if a valid directory

3.1.22 Function `valueValidFile`

valueValidFile()
Return a function that returns true if passed option val if a valid file

3.2 Module `simuPOP.utils`

This module provides some commonly used operators and format conversion utilities.

3.2.1 Class `Trajectory`

A `Trajectory` object contains frequencies of one or more loci in one or more subpopulations over several generations. It is usually returned by member functions of class `TrajectorySimulator` or equivalent global functions `simulateForwardTrajectory` and `simulateBackwardTrajectory`.

The `Trajectory` object provides several member functions to facilitate the use of Trajectory-simulation techiniques. For example, `Trajectory.func()` returns a trajectory function that can be provided directly to a `ControlledOffspringGenerator`; `Trajectory.mutators()` provides a list of `PointMutator` that insert mutants at the right generations to initialize a trajectory.

For more information about Trajectory simulation techniques and related controlled random mating scheme, please refer to the simuPOP user's guide, and Peng et al (PLoS Genetics 3(3), 2007).

class **Trajectory**(*endGen, nLoci*)
Create a `Trajectory` object of alleles at *nLoci* loci with ending generation *endGen*. *endGen* is the generation when expected allele frequencies are reached after mating. Therefore, a trajectory for 1000 generations should have `endGen=999`.

freq(*gen, subPop*)
Return frequencies of all loci in subpopulation *subPop* at generation *gen* of the simulated Trajectory. Allele frequencies are assumed to be zero if *gen* is out of range of the simulated Trajectory.

func()
Return a Python function that returns allele frequencies for each locus at specified loci. If there are multiple subpopulations, allele frequencies are arranged in the order of `loc0_sp0`, `loc1_sp0`, ..., `loc0_sp1`, `loc1_sp1`, ... and so on. The returned function can be supplied directly to the `freqFunc` parameter of a controlled random mating scheme (`ControlledRandomMating`) or a homogeneous mating scheme that uses a controlled offspring generator (`ControlledOffspringGenerator`).

mutants()
Return a list of mutants in the form of (loc, gen, subPop)

mutators(*loci, inds=0, allele=1, *args, **kwargs*)
Return a list of `PointMutator` operators that introduce mutants at the beginning of simulated trajectories. These mutators should be added to the `preOps` parameter of `Simulator.evolve` function to introduce a mutant at the beginning of a generation with zero allele frequency before mating, and a positive allele frequency after mating. A parameter `loci` is needed to specify actual loci indexes in the real forward simulation. Other than default parameters `inds=0` and `allele=1`, additional parameters could be passed to point mutator as keyward parameters.

plot(*filename=None, **kwargs*)

Plot simulated Trajectory using `R` through a Python module `rpy`. The function will return silently if module `plotter` cannot be imported.

This function will use different colors to plot trajectories at different loci. The trajectories are plotted from generation 0 to *endGen* even if the trajectories are short. The y-axis ranges from 0 to 1 and is labeled `Allele frequency`. If a valid *filename* is given, the figure will be saved to *filename* in a format specified by file extension. Currently supported formats/extensions are eps, jpg, bmp, tif, png and pdf. The availability of formats may be limited by your version of R.

This function makes use of the derived keyword parameter feature of module `plotter`. Allowed prefixes are `par`, `plot`, `lines` and `dev_print`. Allowed repeating suffix are `loc` and `sp`. For example, you could use parameter `plot_ylim` to reset the default value of `ylim` in R function `plot`.

3.2.2 Class `TrajectorySimulator`

A Trajectory Simulator takes basic demographic and genetic (natural selection) information of an evolutionary process of a diploid population and allow the simulation of Trajectory of allele frequencies of one or more loci. Trajectories could be simulated in two ways: forward-time and backward-time. In a forward-time simulation, the simulation starts from certain allele frequency and simulate the frequency at the next generation using given demographic and genetic information. The simulation continues until an ending generation is reached. A Trajectory is successfully simulated if the allele frequency at the ending generation falls into a specified range. In a backward-time simulation, the simulation starts from the ending generation with a desired allele frequency and simulate the allele frequency at previous generations one by one until the allele gets lost (allele frequency equals zero).

The result of a trajectory simulation is a trajectory object which can be used to direct the simulation of a special random mating process that controls the evolution of one or more disease alleles so that allele frequencies are consistent across replicate simulations. For more information about Trajectory simulation techniques and related controlled random mating scheme, please refer to the simuPOP user's guide, and Peng et al (PLoS Genetics 3(3), 2007).

class **TrajectorySimulator**(*N, nLoci=1, fitness=None, logger=None*)

Create a trajectory Simulator using provided demographic and genetic (natural selection) parameters. Member functions *simuForward* and *simuBackward* can then be used to simulate trajectories within certain range of generations. This class accepts the following parameters

N: Parameter *N* accepts either a constant number for population size (e.g. N=1000), a list of subpopulation sizes (e.g. N=[1000, 2000]), or a demographic function that returns population or subpopulation sizes at each generation. During the evolution, multiple subpopulations can be merged into one, and one population can be split into several subpopulations. The number of subpopulation is determined by the return value of the demographic function. Note that *N* should be considered as the population size at the end of specified generation.

nLoci: Number of unlinked loci for which trajectories of allele frequencies are simulated. We assume a diploid population with diallelic loci. The Trajectory represents frequencies of a

fitness: Parameter fitness can be `None` (no selection), a list of fitness values for genotype with 0, 1, and 2 disease alleles (*AA*, *Aa*, and *aa*) at one or more loci; or a function that returns fitness values at each generation. When multiple loci are involved (*nLoci*), *fitness* can be a list of 3 (the same fitness values for all loci), a list of 3*nLoci (different fitness values for each locus) or a list of 3**nLoci (fitness value for each combination of genotype). The fitness function should accept generation number and a subpopulation index. The latter parameter allows, and is the only way to specify different fitness in each subpopulation.

logger: A logging object (see Python module `logging`) that can be used to output intermediate results with debug information.

simuBackward(*endGen, endFreq, minMutAge=None, maxMutAge=None, maxAttempts=1000*)

Simulate trajectories of multiple disease susceptibility loci using a forward time approach. This function accepts allele frequencies of alleles of multiple unlinked loci (*endFreq*) at the end of generation *endGen*.

Depending on the number of loci and subpopulations, parameter *beginFreq* can be a number (same frequency for all loci in all subpopulations), or a list of frequencies for each locus (same frequency in all subpopulations), or a list of frequencies for each locus in each subpopulation in the order of `loc0_sp0`, `loc1_sp0`, ..., `loc0_sp1`, `loc1_sp1`, ... and so on.

This simulator will simulate a trajectory generation by generation and restart if the disease allele got fixed (instead of lost), or if the length simulated Trajectory does not fall into *minMutAge* and *maxMutAge* (ignored if `None` is given). This simulator will return `None` if no valid Trajectory is found after `maxAttempts` attemps.

simuForward(*beginGen, endGen, beginFreq, endFreq, maxAttempts=10000*)

Simulate trajectories of multiple disease susceptibility loci using a forward time approach. This function accepts allele frequencies of alleles of multiple unlinked loci at the beginning generation (`freq`) at generation `beginGen`, and expected *range* of allele frequencies of these alleles (`endFreq`) at the end of generation `endGen`. Depending on the number of loci and subpopulations, these parameters accept the following inputs:

beginGen: Starting generation. The initial frequecies are considered as frequencies at the *beginning* of this generation.

endGen: Ending generation. The ending frequencies are considerd as frequencies at the *end* of this generation.

beginFreq: The initial allele frequency of involved loci in all subpopulations. It can be a number (same frequency for all loci in all subpopulations), or a list of frequencies for each locus (same frequency in all subpopulations), or a list of frequencies for each locus in each subpopulation in the order of `loc0_sp0`, `loc1_sp0`, ..., `loc0_sp1`, `loc1_sp1`, ... and so on.

endFreq: The range of acceptable allele frequencies at the ending generation. The ranges can be specified for all loci in all subpopulations, for all loci (allele frequency in the whole population is considered), or for all loci in all subpopulations, in the order of `loc0_sp0`, `loc1_sp0`, ... `loc0_sp1`, ... and so on.

This simulator will simulate a trajectory generation by generation and restart if the resulting frequencies do not fall into specified range of frequencies. This simulator will return `None` if no valid Trajectory is found after `maxAttempts` attemps.

3.2.3 Function `simulateForwardTrajectory`

simulateForwardTrajectory(*N, beginGen, endGen, beginFreq, endFreq, nLoci=1, fitness=None, maxAttempts=10000, logger=None*)

Given a demographic model (*N*) and the fitness of genotype at one or more loci (*fitness*), this function simulates a trajectory of one or more unlinked loci (*nLoci*) from allele frequency *freq* at generation *beginGen* forward in time, until it reaches generation *endGen*. A `Trajectory` object will be returned if the allele frequency falls into specified ranges (*endFreq*). `None` will be returned if no valid Trajectory is simulated after `maxAttempts` attempts. Please refer to class `Trajectory`, `TrajectorySimulator` and their member functions for more details about allowed input for these parameters. If a *logger* object is given, it will send detailed debug information at `DEBUG` level and ending allele frequencies at the `INFO` level. The latter can be used to adjust your fitness model and/or ending allele frequency if a trajectory is difficult to obtain because of parameter mismatch.

3.2.4 Function `simulateBackwardTrajectory`

simulateBackwardTrajectory(*N, endGen, endFreq, nLoci=1, fitness=None, minMutAge=None, maxMutAge=None, maxAttempts=1000, logger=None*)

Given a demographic model (*N*) and the fitness of genotype at one or more loci (*fitness*), this function simulates a trajectory of one or more unlinked loci (*nLoci*) from allele frequency *freq* at generation *endGen* backward in time, until all alleles get lost. A `Trajectory` object will be returned if the length of simulated Trajectory with `minMutAge` and `maxMutAge` (if specified). `None` will be returned if no valid Trajectory is simulated after `maxAttempts` attempts. Please refer to class `Trajectory`, `TrajectorySimulator` and their member functions for more details about allowed input for these parameters. If a *logger* object is given, it will send detailed debug information at

DEBUG level and ending generation and frequency at the INFO level. The latter can be used to adjust your fitness model and/or ending allele frequency if a trajectory is difficult to obtain because of parameter mismatch.

3.2.5 Function migrIslandRates

migrIslandRates(*r, n*)

migration rate matrix

```
x m/(n-1) m/(n-1) ....
m/(n-1) x .............
.....
.... m/(n-1) m/(n-1) x
```

where x = 1-m

3.2.6 Function migrHierarchicalIslandRates

migrHierarchicalIslandRates(*r1, r2, n*)

Return the migration rate matrix for a hierarchical island model where there are different migration rate within and across groups of islands.

r1: Within group migration rates. It can be a number or a list of numbers for each group of the islands.

r2: Across group migration rates which is the probability that someone will migrate to a subpopulation outside of his group. A list of r2 could be specified for each group of the islands.

n: Number of islands in each group. E.g. n=[5, 4] specifies two groups of islands with 5 and 4 islands each.

For individuals in an island, the probability that it remains in the same island is 1-r1-r2 (r1, r2 might vary by island groups), that it migrates to another island in the same group is r1 and migrates to another island outside of the group is r2. migrate rate to a specific island depends on the size of group.

3.2.7 Function migrSteppingStoneRates

migrSteppingStoneRates(*r, n, circular=False*)

migration rate matrix, circular stepping stone model (X=1-m)

```
X   m/2              m/2
m/2 X   m/2          0
0   m/2 x   m/2 ......0
...
m/2 0 ....       m/2  X
```

or non-circular

```
X   m/2              m/2
m/2 X   m/2          0
0   m/2 X   m/2 ......0
...
...              m   X
```

This function returns [[1]] when there is only one subpopulation.

3.2.8 Class ProgressBar

The ProgressBar class defines a progress bar. This class will use a text-based progress bar that outputs progressing dots (.) with intermediate numbers (e.g. 5 for 50%) under a non-GUI mode (gui=False). In the GUI mode, a Tkinter or wxPython progress dialog will be used (gui=Tkinter or gui=wxPython). The default mode is determined by the global gui mode of simuPOP (see also simuOpt.setOptions).

This class is usually used as follows:

```
progress = ProgressBar("Start simulation", 500)
for i in range(500):
    # i+1 can be ignored if the progress bar is updated by 1 step
    progress.update(i+1)
# if you would like to make sure the done message is displayed.
progress.done()
```

class ProgressBar(*message, totalCount, progressChar='.', block=2, done=' Done.\n', gui=None*)

Create a progress bar with message, which will be the title of a progress dialog or a message for textbased progress bar. Parameter totalCount specifies total expected steps. If a text-based progress bar is used, you could specified progress character and intervals at which progresses will be displayed using parameters progressChar and block. A ending message will also be displayed in text mode.

done()

Finish progressbar, print 'done' message if in text-mode.

update(*count=None*)

Update the progreebar with count steps done. The dialog or textbar may not be updated if it is updated by full percent(s). If count is None, the progressbar increases by one step (not percent).

3.2.9 Function viewVars

viewVars(*var, gui=None*)

list a variable in tree format, either in text format or in a: wxPython window.

var: A dictionary variable to be viewed. Dictionary wrapper objects returned by Population.dvars() and Simulator.dvars() are also acceptable.

gui: If gui is False or 'Tkinter', a text presentation (use the pprint module) of the variable will be printed to the screen. If gui is 'wxPython' and wxPython is available, a wxPython windows will be used. The default mode is determined by the global gui mode (see also simuOpt.setOptions).

3.2.10 Function saveCSV

saveCSV(*pop, filename='', infoFields=[], loci=True, header=True, subPops=ALL_AVAIL, genoFormatter=None, infoFormatter=None, sexFormatter={1: 'M', 2: 'F'}, affectionFormatter={False: 'U', True: 'A'}, sep=', ', **kwargs*)

Save a simuPOP population pop in csv format. Columns of this file is arranged in the order of information fields (infoFields), sex (if sexFormatter is not None), affection status (if affectionFormatter is not None), and genotype (if genoFormatter is not None). This function only output individuals in the present generation of population pop. This function accepts the following parameters:

pop: A simuPOP population object.

filename: Output filename. Leading '>' characters are ignored. However, if the first character of this filename is '!', the rest of the name will be evalulated in the population's local namespace. If filename is empty, the content will be written to the standard output.

infoFileds: Information fields to be outputted. Default to none.

loci: If a list of loci is given, only genotype at these loci will be written. Default to `ALL_AVAIL`, meaning all available loci. You can set this parameter to `[]` if you do not want to output any genotype.

header: Whether or not a header should be written. These headers will include information fields, sex (if `sexFormatter` is not `None`), affection status (if `affectionFormatter` is not `None`) and loci names. If genotype at a locus needs more than one column, `_1`, `_2` etc will be appended to loci names. Alternatively, a complete header (a string) or a list of column names could be specified directly.

subPops: A list of (virtual) subpopulations. If specified, only individuals from these subpopulations will be outputed.

infoFormatter: A format string that is used to format all information fields. If unspecified, `str(value)` will be used for each information field.

genoFormatter: How to output genotype at specified loci. Acceptable values include `None` (output allele names), a dictionary with genotype as keys, (e.g. `genoFormatter={(0,0):1, (0,1):2, (1,0):2, (1,1):3}`, or a function with genotype (as a tuple of integers) as inputs. The dictionary value or the return value of this function can be a single or a list of number or strings.

sexFormatter: How to output individual sex. Acceptable values include `None` (no output) or a dictionary with keys `MALE` and `FEMALE`.

affectionFormatter: How to output individual affection status. Acceptable values include `None` (no output) or a dictionary with keys `True` and `False`.

Parameters `genoCode`, `sexCode`, and `affectionCode` from version 1.0.0 have been renamed to `genoFormatter`, `sexFormatter` and `affectionFormatter` but can still be used.

3.3 Module `simuPOP.plotter`

This module defines several utility functions and Python operators that make use of the Python rpy module (`http://rpy.sourceforge.net`) to plot expressions and information fields of evolving populations using a popular statistical analysis language R (`http://www.r-project.org`). Note that rpy2, the successor of rpy, is currently not supported.

Each operator calls a sequence of R functions to draw and save figures. A special parameter passing mechanism is used so that you can specify arbitrary parameters to these functions. For example, you can use parameter `par_mfrow=[2,2]` to pass `mfrow=[2,2]` to function `par`, and use `lty_rep=[1,2]` to pass `lty=1` and `lty=2` to specify different line types for different replicates. The help message of each class will describe which and in what sequence these R functions are called to help you figure out which parameters are allowed.

3.3.1 Function `newDevice`

newDevice()

Create a new graphics window and return its device number in R. This function essentially calls `getOption('device')()` in R.

3.3.2 Function `saveFigure`

saveFigure(*file=None, **kwargs*)

Save current figure into `file`. File format and graphics device are determined by file extension. Supported file formats include `pdf`, `png`, `bmp`, `jpg (jpeg)`, `tif (tiff)`, and `eps`, which correspond to R devices `pdf`, `png`, `bmp`, `jpeg`, `tiff` and `postscript`. A postscript device will be used if there is no file extension or the file extension is not recognizable. Additional keyword parameters will be passed to the underlying `dev.print` function.

3.3.3 Class `VarPlotter`

This class defines a Python operator that uses R to plot the current and historical values of a Python expression (`expr`), which are evaluated (against each population's local namespace) and saved during evolution. The return value of the expression can be a number or a sequence, but should have the same type and length across all replicates and generations. Histories of each value (or each item in the returned sequence) of each replicate form a line, with generation numbers as its x-axis. Number of lines will be the number of replicates multiplied by dimension of the expression. Although complete histories are usually saved, you can use parameter `win` to save histories only within the last `win` generations.

A figure will be draw at the end of the last replicate (except for the first generation where no line could be drawn) unless the current generation is less than `update` generations away from the last generation at which a figure has been drawn. Lines for multiple replicates or dimensions could be plotted in the same figure (by default), or be seperated to subplots by replicates (`byRep`), by each dimention of the results (`byDim`), or by both. These figure could be saved to files in various formats if parameter `saveAs` is specified. File format is determined by file extension. After the evolution, the graphic device could be left open (`leaveOpen`).

Besides parameters mentioned above, arbitrary keyword parameters could be specified and be passed to the underlying R drawing functions `plot` and `lines`. These parameters could be used to specify line type (`lty`), color (`col`), title (`main`), limit of x and y axes (`xlim` and `ylim`) and many other options (see R manual for details). As a special case, multiple values can be passed to each replicate and/or dimension if the name of a parameter ends with `_rep`, `_dim`, or `_repdim` For example, `lty_rep=range(1, 5)` will pass parameters `lty=1`, ... `lty=4` to four replicates. You can also pass parameters to specific R functions such as `par`, `plot`, `lines`, `legend`, `dev_print` by prefixing parameter names with a function name. For example, `dev_print_width=300` will pass `width=300` to function `dev.print()` when you save your figures using this function. In addition, if the value of a parameter is a string starting with `!`, the evaluated result of the remaining string will be used as parameter value. Further customization of your figures could be achieved by writing your own hook functions that will be called before and after a figure is drawn, and after each `plot` call.

This opertor calls R functions `par`, `plot`, `lines`, `legend`, and `dev.print`. Functions `plot` and `lines` are the default destination for keyword arguments and the ones that accept list parameters to customize lines by replicate and/or dimension.

class `VarPlotter`(*expr, win=0, update=1, byRep=False, byDim=False, saveAs='', leaveOpen=False, legend=[], preHook=None, postHook=None, plotHook=None, begin=0, end=-1, step=1, at=[], reps=True, **kwargs*)

expr: expression that will be evaluated at each replicate's local namespace when the operator is applied. Its value can be a number or a list (or tuple) but the type and length of the return value should be consistent for all replicates and at all generations.

win: Window of generations. If given, only values from generation -win to -1 will be plotted.

update: Update the figure after specified generations. For example, you can evalulate an expression and save its values at every 10 generations (parameter `step=10`) but only draw a figure after every 50 generations (parameter `update=50`.

byRep: Separate values at different replicates to different subplots.

byDim: Separate items from sequence results of `expr` to different subplots. If both `byRep` and `byDim` are `True`, the subplots will be arranged by variable and then replicates.

saveAs: Save figures in files saveAs_gen.ext (e.g. `figure_10.eps` if `saveAs='figure.eps'`). If ext is given, a corresponding device will be used. Otherwise, a default postscript driver will be used. Currently supported formats include `.pdf`, `.png`, `.bmp`, `.jpg`, and `.tif`. The default filename could be overridden by derived argument `dev_print_file`.

leaveOpen: Whether or not leave the plot open when plotting is done. Default to `False` functions. If this option is set to `True`, you will have to close the graphic device explicitly using function `r.dev_off()`. Note that leaving the device open allows further manipuation of the figures outside of this operator.

legend: labels of the lines. This operator will look for keyword parameters such as `col`, `lty`, `lwd`, and `pch` and call the `legend` function to draw a legend. If figure has multiple lines for both replicates and dimensions, legends should be given to each dimension, and then each replicate.

preHook: A function that, if given, will be called before the figure is draw. The `r` object from the `plotter` module will be passed to this function.

postHook: A function that, if given, will be called after the figure is drawn. The `r` object from the `plotter` module will be passed to this function.

plotHook: A function that, if given, will be called after each `plot` function. The `r` object from the `plotter` module, generation list, data being plotted, replicate number (if applicable) and dimension index (if applicable) will be passed as keyword arguments `r`, `gen`, `data`, `rep` (optional) and `dim` (optional).

kwargs: Additional keyword arguments that will be interpreted and sent to underlying R functions. These arguments could have prefixes (destination function names) `plot_`, `lines_`, `par_`, `legend_` and `dev_print_`, and suffixes (list parameters) `_rep`, `_dim`, and `_repdim`. Arguments without prefixes are sent to functions `plot` and `lines`. String values with a leading `!` will be replaced by its evaluated result against the current population.

3.3.4 Class `ScatterPlotter`

This class defines a Python operator that uses R to plot individuals in a Population, using values at two information fields as their x- and y-axis.

Arbitrary keyword parameters could be specified and be passed to the underlying R drawing functions `plot` and `points`. These parameters could be used to specify point type (`pch`), color (`col`), title (`main`), limit of x and y axes (`xlim` and `ylim`) and many other options (see R manual for details). You can also pass parameters to specific R functions such as `par`, `plot`, `points`, `legend`, `pdf` by prefixing parameter names with a function name. For example, `par_mar=[1]*4` will pass `par=[1]*4` to function `par()` which is called before a figure is drawn. (Note that the function to save a figure is `dev.print` so parameters such as `dev_print_width` should be used.) Further customization of your figures could be achieved by writing your own hook functions that will be called before and after a figure is drawn.

The power of this operator lies in its ability to differentiate individuals from different (virtual) subpopulations. If you specify IDs of (virtual) subpopulations (VSPs) in parameter `subPops`, only individuals from these VSPs will be displayed. Points from these subpopulations will be drawn with different shapes and colors. You can also customize these points using list parameters with suffix `_sp`. For example, if you have defined two VSPs by sex and set `subPops=[(0, 0), (0, 1)]`, `col_sp=['blue', 'red']` will color male individuals with blue and female individuals with red. In addition, if the value of a parameter is a string starting with `!`, the evaluated result of the remaining string will be used as parameter value.

This opertor calls R functions `par`, `plot`, `points`, `legend`, and `dev.print`. Functions `plot` and `points` are the default destination for keyword arguments and the ones that accept list parameters to customize lines by (virtual) subpopulation.

class `ScatterPlotter`(*infoFields=[], saveAs='', leaveOpen=False, legend=[], preHook=None, postHook=None, begin=0, end=-1, step=1, at=[], reps=True, subPops=[], **kwargs*)

infoFields: Two information fields whose values will be the x- and y-axis of each point (individual) in the plot.

subPops: A list of subpopulations and virtual subpopulations. Only individuals from these subpopulations will be plotted. Default to subpopulation indexes.

saveAs: Save figures in files saveAs_gen_rep.ext (e.g. `figure_10_0.eps` if `saveAs='figure.eps'`). If ext is given, a corresponding device will be used. Otherwise, a default postscript driver will be used. Currently supported formats include `.pdf`, `.png`, `.bmp`, `.jpg`, and `.tif`. The default filename could be overriden by derived argument `dev_print_file`.

leaveOpen: Whether or not leave the plot open when plotting is done. Default to `False` functions. If this option is set to `True`, you will have to close the graphic device explicitly using function `r.dev_off()`. Note that leaving the device open allows further manipuation of the figures outside of this operator.

legend: labels of the points. It must match the specified subpopulations.

preHook: A function that, if given, will be called before the figure is draw. The `r` object from the `plotter` module will be passed to this function.

postHook: A function that, if given, will be called after the figure is drawn. The `r` object from the `plotter` module will be passed to this function.

kwargs: Additional keyword arguments that will be interpreted and sent to underlying R functions. These arguments could have prefixes (destination function names) `plot_`, `points_`, `par_`, `legend_` and `dev_print_`, and suffixes (list parameters) `_sp`. Arguments without prefixes are sent to functions `plot` and `points`. String values with a leading `!` will be replaced by its evaluated result against the current population.

3.3.5 Class `InfoPlotter`

This operator uses a R function such as `hist` and `qqplot` to plot properties of one or more information fields of individuals in one or more (virtual) subpopulations. Separate subplots are used for different information fields and subpopulations.

This operator essentially gets values of information fields and sends them to a R function such as `hist`. The resulting figures could be customized by additional keyword parameters and various hook functions. For example, a `qqline` function could be called in a `plotHook` function to add a QQ line to a `qqnorm` plot. The `plotHook` can be used to draw the whole (sub)plot if no R function is specified for parameter `func`.

Besides regular keyword parameters, keyword parameters ending in `_sp`, `_fld` or `_spfld` are expected to have multiple values which will be used for differnt subpopulations, information fields, and their combinations. You can also specify which function the keyword should be sent by prefixing a function name to the parameter name. For example, `pch_fld=[1, 2]` will use different symbols for different information fields, and `par_mar=[1]*4` will send parameter `mar=[1]*4` to function `par`. In addition, if the value of a parameter is a string starting with `!`, the evaluated result of the remaining string will be used as parameter value.

This opertor calls R functions `par`, `dev.print`, and a user-specified function. Additional keyword arguments without function prefix will be sent to this function.

class `InfoPlotter`(*func=None, infoFields=[], saveAs='', leaveOpen=False, preHook=None, postHook=None, plotHook=None, begin=0, end=-1, step=1, at=[], reps=True, subPops=[], **kwargs*)

func: Name of the R function that will be called to draw figures from values of given information fields. No R function will be called if it is not specified. In this case, a `plotHook` can be used to plot passed values.

infoFields: Information fields whose values will be sent to the specified plotting function.

subPops: A list of subpopulations and virtual subpopulations. Each subpopulation will be plotted in a separate subplot.

saveAs: Save figures in files saveAs_gen_rep.ext (e.g. `figure_10_0.eps` if `saveAs='figure.eps'`). If ext is given, a corresponding device will be used. Otherwise, a default postscript driver will be used. Currently supported formats include `.pdf`, `.png`, `.bmp`, `.jpg`, and `.tif`. The default filename could be overriden by derived argument `dev_print_file`.

leaveOpen: Whether or not leave the plot open when plotting is done. Default to `False` functions. If this option is set to `True`, you will have to close the graphic device explicitly using function `r.dev_off()`. Note that leaving the device open allows further manipuation of the figures outside of this operator.

preHook: A function that, if given, will be called before the figure is draw. The `r` object from the `plotter` module will be passed to this function.

postHook: A function that, if given, will be called after the figure is drawn. The `r` object from the `plotter` module will be passed to this function.

plotHook: A function that, if given, will be called after each specified plot function. The `r` object from the `plotter` module, data being plotted, name of the information field and index of subpopulation (in parameter `subPops`, if applicable) will be passed with keywords `r`, `data`, `field` and `subPop` (optional) respectively.

kwargs: Additional keyword arguments that will be interpreted and sent to underlying R functions. These arguments could have prefixes (destination function names) `par_`, `dev_print_` and the function you specify (parameter `func`), and suffixes (list parameters) `_sp`, `_fld`, and `_spfld`. Arguments without prefixes are sent

to the user specified function. String values with a leading `!` will be replaced by its evaluated result against the current population.

3.3.6 Class `HistPlotter`

An `InfoPlotter` that uses function `hist`.

class `HistPlotter`(**args, **kwargs*)

Returns an `InfoPlotter` that uses R function `hist` to draw histogram of individual information fields of specified (virtual) subpopulations. Please see `InfoPlotter` for details.

3.3.7 Class `QQPlotter`

An `InfoPlotter` that uses function `qqnorm`.

class `QQPlotter`(**args, **kwargs*)

Returns an `InfoPlotter` that uses R function `qqnorm` to draw qq plot of individual information fields of specified (virtual) subpopulations. Please see `InfoPlotter` for details.

3.3.8 Class `BoxPlotter`

This operator draws boxplots of one or more information fields of individuals in one or more (virtual) subpopulations of a population. Although a `InfoPlotter` with `func=boxplot` could be used to plot boxplots for each information field and/or subpopulation, this class allows multiple whiskers to share one plot. How the whiskers are oraganized is controlled by parameters `byField` and `bySubPop`.

This operator essentially gets values of information fields and sends them to boxplots. Individual ownerships (subpopulation or field) are also passed so that multiple whiskers could be drawn in the same plot. The resulting figures could be customized by additional keyword parameters and various hook functions.

Besides regular keyword parameters, keyword parameters ending in `_sp`, `_fld` or `_spfld` are expected to have multiple values which will be used for differnt subpopulations, information fields, and their combinations. You can also specify which function the keyword should be sent by prefixing a function name to the parameter name. For example, `pch_fld=[1, 2]` will use different symbols for different information fields, and `par_mar=[1]*4` will send parameter `mar=[1]*4` to function `par`. In addition, if the value of a parameter is a string starting with `!`, the evaluated result of the remaining string will be used as parameter value.

This opertor calls R functions `par`, `boxplot` and `dev.print`. Keyword parameters without function prefix will be passed to `boxplot`.

class `BoxPlotter`(*infoFields=[], byField=False, bySubPop=False, saveAs='', leaveOpen=False, preHook=None, postHook=None, plotHook=None, begin=0, end=-1, step=1, at=[], reps=True, subPops=[], **kwargs*)

infoFields: Information fields whose values will be sent to R function `boxplot`.

subPops: A list of subpopulations and virtual subpopulations. Separate whiskers will be drawn for individuals in these subpopulations.

byField: If multiple information fields are specified, separate the whiskers different subplots if this parameter is `True`.

bySubPop: If multiple (virtual) subpopulations are specified, separate the whiskers to different subplots if this parameter is `True`.

saveAs: Save figures in files saveAs_gen_rep.ext (e.g. `figure_10_0.eps` if `saveAs='figure.eps'`). If ext is given, a corresponding device will be used. Otherwise, a default postscript driver will be used. Currently supported formats include `.pdf`, `.png`, `.bmp`, `.jpg`, and `.tif`. The default filename could be overriden by derived argument `dev_print_file`.

leaveOpen: Whether or not leave the plot open when plotting is done. Default to `False` functions. If this option is set to `True`, you will have to close the graphic device explicitly using function `r.dev_off()`. Note that leaving the device open allows further manipuation of the figures outside of this operator.

preHook: A function that, if given, will be called before the figure is draw. The `r` object from the `plotter` module will be passed to this function.

postHook: A function that, if given, will be called after the figure is drawn. The `r` object from the `plotter` module will be passed to this function.

plotHook: A function that, if given, will be called after each specified plot function. The `r` object from the `plotter` module, current field and subpopulation will be passed with keywords `r`, `field` and `subPop` if applicable.

kwargs: Additional keyword arguments that will be interpreted and sent to underlying R functions. These arguments could have prefixes (destination function names) `plot_`, `boxplot_`, `par_`, and `dev_print_`, and suffixes (list parameters) `_sp`, `_fld` and `_spfld`. Arguments without prefixes are sent to function `boxplot`. String values with a leading `!` will be replaced by its evaluated result against the current population.

3.4 Module `simuPOP.sampling`

This module provides classes and functions that could be used to draw samples from a simuPOP population. These functions accept a list of parameters such as `subPops` ((virtual) subpopulations from which samples will be drawn) and `numOfSamples` (number of samples to draw) and return a list of populations. Both independent individuals and dependent individuals (Pedigrees) are supported.

Independent individuals could be drawn from any Population. pedigree information is not necessary and is usually ignored. Unique IDs are not needed either although such IDs could help you identify samples in the parent Population.

Pedigrees could be drawn from multi-generational populations or age-structured populations. All individuals are required to have a unique ID (usually tracked by operator `IdTagger` and are stored in information field `ind_id`). Parents of individuals are usually tracked by operator `PedigreeTagger` and are stored in information fields `father_id` and `mother_id`. If parental information is tracked using operator `ParentsTagger` and information fields `father_idx` and `mother_idx`, a function `sampling.indexToID` can be used to convert index based pedigree to ID based Pedigree. Note that `ParentsTagger` can not be used to track Pedigrees in age-structured populations because they require parents of each individual resides in a parental generation.

All sampling functions support virtual subpopulations through parameter `subPops`, although sample size specification might vary. This feature allows you to draw samples with specified properties. For example, you could select only female individuals for cases of a female-only disease, or select individuals within certain age-range. If you specify a list of (virtual) subpopulations, you are usually allowed to draw certain number of individuals from each subpopulation.

3.4.1 Class `BaseSampler`

A sampler extracts individuals from a simuPOP population and return them as separate populations. This base class defines the common interface of all sampling classes, including how samples prepared and returned.

class `BaseSampler`(*subPops=ALL_AVAIL*)

Create a sampler with parameter `subPops`, which will be used to prepare population for sampling. `subPops` should be a list of (virtual) subpopulations from which samples are drawn. The default value is ALL_AVAIL, which means all available subpopulations of a Population.

`drawSample`(*pop*)

Draw and return a sample.

`drawSamples`(*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

prepareSample(*pop, rearrange*)
Prepare passed population object for sampling according to parameter `subPops`. If samples are drawn from the whole population, a Population will be trimmed if only selected (virtual) subpopulations are used. If samples are drawn separately from specified subpopulations, Population `pop` will be rearranged (if `rearrange==True`) so that each subpoulation corresponds to one element in parameter `subPops`.

3.4.2 Class `RandomSampler`

A sampler that draws individuals randomly.

class **RandomSampler**(*sizes, subPops=ALL_AVAIL*)
Creates a random sampler with specified number of individuals.

drawSample(*input_pop*)
Draw a random sample from passed population.

drawSamples(*pop, numOfSamples*)
Draw multiple samples and return a list of populations.

prepareSample(*pop, rearrange*)
Prepare passed population object for sampling according to parameter `subPops`. If samples are drawn from the whole population, a Population will be trimmed if only selected (virtual) subpopulations are used. If samples are drawn separately from specified subpopulations, Population `pop` will be rearranged (if `rearrange==True`) so that each subpoulation corresponds to one element in parameter `subPops`.

3.4.3 Function `drawRandomSample`

drawRandomSample(*pop, sizes, subPops=ALL_AVAIL*)
Draw `sizes` random individuals from a population. If a single `sizes` is given, individuals are drawn randomly from the whole population or from specified (virtual) subpopulations (parameter `subPops`). Otherwise, a list of numbers should be used to specify number of samples from each subpopulation, which can be all subpopulations if `subPops=ALL_AVAIL` (default), or from each of the specified (virtual) subpopulations. This function returns a population with all extracted individuals.

3.4.4 Function `drawRandomSamples`

drawRandomSamples(*pop, sizes, numOfSamples=1, subPops=ALL_AVAIL*)
Draw `numOfSamples` random samples from a population and return a list of populations. Please refer to function `drawRandomSample` for more details about parameters `sizes` and `subPops`.

3.4.5 Class `CaseControlSampler`

A sampler that draws affected and unaffected individuals randomly.

class **CaseControlSampler**(*cases, controls, subPops=ALL_AVAIL*)
Ceates a case-control sampler with specified number of cases and controls.

drawSample(*input_pop*)
Draw a case control sample

drawSamples(*pop, numOfSamples*)
Draw multiple samples and return a list of populations.

prepareSample(*input_pop*)
Find out indexes all affected and unaffected individuales.

3.4.6 Function `drawCaseControlSample`

drawCaseControlSample(*pop, cases, controls, subPops=ALL_AVAIL*)

Draw a case-control samples from a population with `cases` affected and `controls` unaffected individuals. If single `cases` and `controls` are given, individuals are drawn randomly from the whole Population or from specified (virtual) subpopulations (parameter `subPops`). Otherwise, a list of numbers should be used to specify number of cases and controls from each subpopulation, which can be all subpopulations if `subPops=ALL_AVAIL` (default), or from each of the specified (virtual) subpopulations. This function returns a population with all extracted individuals.

3.4.7 Function `drawCaseControlSamples`

drawCaseControlSamples(*pop, cases, controls, numOfSamples=1, subPops=ALL_AVAIL*)

Draw `numOfSamples` case-control samples from a population with `cases` affected and `controls` unaffected individuals and return a list of populations. Please refer to function `drawCaseControlSample` for a detailed descriptions of parameters.

3.4.8 Class `PedigreeSampler`

The base class of all pedigree based sampler.

class **PedigreeSampler**(*families, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Creates a pedigree sampler with parameters

families: number of families. This can be a number or a list of numbers. In the latter case, specified families are drawn from each subpopulation.

subPops: A list of (virtual) subpopulations from which samples are drawn. The default value is ALL_AVAIL, which means all available subpopulations of a population.

drawSample(*input_pop*)

Randomly select Pedigrees

drawSamples(*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

family(*id*)

Get the family of individual with id.

prepareSample(*pop, loci=[], infoFields=[], ancGens=True*)

Prepare self.pedigree, some pedigree sampler might need additional loci and information fields for this sampler.

3.4.9 Class `AffectedSibpairSampler`

A sampler that draws a nuclear family with two affected offspring.

class **AffectedSibpairSampler**(*families, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Initialize an affected sibpair sampler.

drawSample(*input_pop*)

Randomly select Pedigrees

drawSamples(*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

family(*id*)
Return id, its spouse and their children

prepareSample(*input_pop*)
Find the father or all affected sibpair families

3.4.10 Function drawAffectedSibpairSample

drawAffectedSibpairSample(*pop, families, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)
Draw affected sibpair samples from a population. If a single `families` is given, affected sibpairs and their parents are drawn randomly from the whole population or from specified (virtual) subpopulations (parameter `subPops`). Otherwise, a list of numbers should be used to specify number of families from each subpopulation, which can be all subpopulations if `subPops=ALL_AVAIL` (default), or from each of the specified (virtual) subpopulations. This function returns a population that contains extracted individuals.

3.4.11 Function drawAffectedSibpairSamples

drawAffectedSibpairSamples(*pop, families, numOfSamples=1, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)
Draw `numOfSamples` affected sibpair samplesa from population `pop` and return a list of populations. Please refer to function `drawAffectedSibpairSample` for a description of other parameters.

3.4.12 Class NuclearFamilySampler

A sampler that draws nuclear families with specified number of affected parents and offspring.

class **NuclearFamilySampler**(*families, numOffspring, affectedParents=0, affectedOffspring=0, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)
Creates a nuclear family sampler with parameters

families: number of families. This can be a number or a list of numbers. In the latter case, specified families are drawn from each subpopulation.

numOffspring: number of offspring. This can be a fixed number or a range [min, max].

affectedParents: number of affected parents. This can be a fixed number or a range [min, max].

affectedOffspring: number of affected offspring. This can be a fixed number of a range [min, max].

subPops: A list of (virtual) subpopulations from which samples are drawn. The default value is ALL_AVAIL, which means all available subpopulations of a population.

drawSample(*input_pop*)
Randomly select Pedigrees

drawSamples(*pop, numOfSamples*)
Draw multiple samples and return a list of populations.

family(*id*)
Return id, its spouse and their children

3.4.13 Function drawNuclearFamilySample

drawNuclearFamilySample(*pop, families, numOffspring, affectedParents=0, affectedOffspring=0, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)
Draw nuclear families from a population. Number of offspring, number of affected parents and number of

affected offspring should be specified using parameters `numOffspring`, `affectedParents` and `affectedOffspring`, which can all be a single number, or a range `[a, b]` (`b` is incldued). If a single `families` is given, Pedigrees are drawn randomly from the whole population or from specified (virtual) subpopulations (parameter `subPops`). Otherwise, a list of numbers should be used to specify numbers of families from each subpopulation, which can be all subpopulations if `subPops=ALL_AVAIL` (default), or from each of the specified (virtual) subpopulations. This function returns a population that contains extracted individuals.

3.4.14 Function `drawNuclearFamilySamples`

drawNuclearFamilySamples(*pop, families, numOffspring, affectedParents=0, affectedOffspring=0, numOfSamples=1, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Draw `numOfSamples` affected sibpair samplesa from population `pop` and return a list of populations. Please refer to function `drawNuclearFamilySample` for a description of other parameters.

3.4.15 Class `ThreeGenFamilySampler`

A sampler that draws three-generation families with specified pedigree size and number of affected individuals.

class ThreeGenFamilySampler(*families, numOffspring, pedSize, numOfAffected=0, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

families: number of families. This can be a number or a list of numbers. In the latter case, specified families are drawn from each subpopulation.

numOffspring: number of offspring. This can be a fixed number or a range [min, max].

pedSize: number of individuals in the Pedigree. This can be a fixed number or a range [min, max].

numAfffected: number of affected individuals in the Pedigree. This can be a fixed number or a range [min, max]

subPops: A list of (virtual) subpopulations from which samples are drawn. The default value is ALL_AVAIL, which means all available subpopulations of a population.

drawSample(*input_pop*)

Randomly select Pedigrees

drawSamples(*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

family(*id*)

Return id, its spouse, their children, children's spouse and grandchildren

3.4.16 Function `drawThreeGenFamilySample`

drawThreeGenFamilySample(*pop, families, numOffspring, pedSize, numOfAffected=0, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Draw three-generation families from a population. Such families consist of grant parents, their children, spouse of these children, and grand children. Number of offspring, total number of individuals, and total number of affected individuals in a pedigree should be specified using parameters `numOffspring`, `pedSize` and `numOfAffected`, which can all be a single number, or a range `[a, b]` (`b` is incldued). If a single `families` is given, Pedigrees are drawn randomly from the whole Population or from specified (virtual) subpopulations (parameter `subPops`). Otherwise, a list of numbers should be used to specify numbers of families from each subpopulation, which can be all subpopulations if `subPops=ALL_AVAIL` (default), or from each of the specified (virtual) subpopulations. This function returns a population that contains extracted individuals.

3.4.17 Function `drawThreeGenFamilySamples`

drawThreeGenFamilySamples(*pop, families, numOffspring, pedSize, numOfAffected=0, numOfSamples=1, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Draw `numOfSamples` three-generation pedigree samples from population `pop` and return a list of populations. Please refer to function `drawThreeGenFamilySample` for a description of other parameters.

3.4.18 Class `CombinedSampler`

A combined sampler accepts a list of sampler objects, draw samples and combine the returned sample into a single population. An id field is required to use this sampler, which will be used to remove extra copies of individuals who have been drawn by different samplers.

class **CombinedSampler**(*samplers=[], idField='ind_id'*)

samplers: A list of samplers

drawSamples(*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

prepareSample(*pop, rearrange*)

Prepare passed population object for sampling according to parameter `subPops`. If samples are drawn from the whole population, a Population will be trimmed if only selected (virtual) subpopulations are used. If samples are drawn separately from specified subpopulations, Population `pop` will be rearranged (if `rearrange==True`) so that each subpoulation corresponds to one element in parameter `subPops`.

3.4.19 Function `drawCombinedSample`

drawCombinedSample(*pop, samplers, idField='ind_id'*)

Draw different types of samples using a list of `samplers`. A Population consists of all individuals from these samples will be returned. An `idField` that stores an unique ID for all individuals is needed to remove duplicated individuals who are drawn multiple numOfSamples from these samplers.

3.4.20 Function `drawCombinedSamples`

drawCombinedSamples(*pop, samplers, numOfSamples=1, idField='ind_id'*)

Draw combined samples `numOfSamples` numOfSamples and return a list of populations. Please refer to function `drawCombinedSample` for details about parameters `samplers` and `idField`.

3.5 Module `simuPOP.gsl`

This module exposes the following GSL (GUN Scientific Library) functions used by simuPOP to the user interface. Although more functions may be added from time to time, this module is not intended to become a complete wrapper for GSL. Please refer to the GSL reference manual (`http://www.gnu.org/software/gsl/manual/html_node/`) for details about these functions. Note that random number generation functions are wrapped into the simuPOP.RNG class.

- `gsl_cdf_gaussian_P(x, sigma)`
- `gsl_cdf_gaussian_Q(x, sigma)`
- `gsl_cdf_gaussian_Pinv(P, sigma)`
- `gsl_cdf_gaussian_Qinv(Q, sigma)`

- gsl_cdf_ugaussian_P(x)
- gsl_cdf_ugaussian_Q(x)
- gsl_cdf_ugaussian_Pinv(P)
- gsl_cdf_ugaussian_Qinv(Q)
- gsl_cdf_exponential_P(x, mu)
- gsl_cdf_exponential_Q(x, mu)
- gsl_cdf_exponential_Pinv(P, mu)
- gsl_cdf_exponential_Qinv(Q, mu)
- gsl_cdf_chisq_P(x, nu)
- gsl_cdf_chisq_Q(x, nu)
- gsl_cdf_chisq_Pinv(P, nu)
- gsl_cdf_chisq_Qinv(Q, nu)
- gsl_cdf_gamma_P(x, a, b)
- gsl_cdf_gamma_Q(x, a, b)
- gsl_cdf_gamma_Pinv(P, a, b)
- gsl_cdf_gamma_Qinv(Q, a, b)
- gsl_ran_gamma_pdf(x, a, b)
- gsl_cdf_beta_P(x, a, b)
- gsl_cdf_beta_Q(x, a, b)
- gsl_cdf_beta_Pinv(P, a, b)
- gsl_cdf_beta_Qinv(Q, a, b)
- gsl_ran_beta_pdf(x, a, b)
- gsl_cdf_binomial_P(k, p, n)
- gsl_cdf_binomial_Q(k, p, n)
- gsl_ran_binomial_pdf(k, p, n)
- gsl_cdf_poisson_P(k, mu)
- gsl_cdf_poisson_Q(k, mu)
- gsl_ran_poisson_pdf(k, mu)

3.6 Module simuPOP.sandbox

simuPOP sandbox. This module contains classes (operators and mating schemes) and functions that are either experimental (and have a chance to be formally added to simuPOP), or designed for special purposes. Because of the temporary nature of these classes, it is recommended that modules that use these classes include Python version of them so that they can be used when the sandbox version is no longer available.

3.6.1 Function `revertFixedSites`

revertFixedSites(*pop*)
Apply operator `RevertFixedSites` to `pop`

3.6.2 Class `RevertFixedSites`

This operator looks into a population in mutational space and revert a mutant to wildtype allele if it is fixed in the population. If a valid output is specifieid, fixed alleles will be outputed with a leading generation number.

class **RevertFixedSites**(*output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)
Create an operator to revert alleles at fixed loci from value 1 to 0. Parameter *subPops* is ignored.

3.6.3 Class `MutSpaceMutator`

This is an infite site mutation model in mutational space. The alleles in the population is assumed to be locations of mutants. A mutation rate is given that mutate alleles in 'regions'. If number of mutants for an individual exceed the number of loci, 10 loci will be added to everyone in the population.

class **MutSpaceMutator**(*rate, ranges, model=1, output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)
This operator accepts a list of ranges which is the 'real range' of each chromosome. Mutation happens with muation rate *rate* and mutants will be recorded to the population (instead of alleles). By default, this mutator assumes an finite-allele model where all mutations are allowed and if a mutant (allele 1) is mutated, it will be mutated to allele 0 (back mutation). Alternatively (*model* = 2), an infinite-sites mutation model can be used where mutations can happen only at a new locus. Mutations happen at a locus with existing mutants will be moved to a random locus without existing mutant. A warning message will be printed if there is no vacant locus available. If a valid *output* is given, mutants will be outputted in the format of "gen mutant ind type" where type is 0 for forward (0->1), 1 for backward (1->0), 2 for relocated mutations, and 3 for ignored mutation because no vacent locus is available. The second mode has the advantage that all mutants in the simulated population can be traced to a single mutation event. If the regions are reasonably wide and mutation rates are low, these two mutation models should yield similar results.

apply(*pop*)
Apply an operator to population *pop* directly, without checking its applicability.

3.6.4 Class `MutSpaceSelector`

This selector assumes that alleles are mutant locations in the mutational space and assign fitness values to them according to a random distribution. The overall individual fitness is determined by either an additive, an multiplicative or an exponential model.

class **MutSpaceSelector**(*selDist, mode=EXPONENTIAL, output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=ALL_AVAIL*)
Create a selector that assigns individual fitness values according to random fitness effects. *selDist* can be

- `(CONSTANT, s, h)` where s will be used for all mutants. The fitness value for genotypes AA, Aa and aa will be (1, 1-hs, 1-s). If h is unspecified, a default value h=0.5 (additive model) will be used.
- `(GAMMA_DISTRIBUTION, theta, k, h` where s follows a gamma distribution with scale parameter theta and shape parameter k. Fitness values for genotypes AA, Aa and aa will be 1, 1-hs and 1-s. A default value h=0.5 will be used if h is unspecified.
- a Python function, which will be called when selection coefficient of a new mutant is needed. This function should return a single value s (with default value h=0.5) or a sequence of (h, s). Mutant location will be passed to this function if it accepts a parameter `loc`. This allows the definition of site-specific selection

coefficients. Individual fitness will be combined in ADDITIVE, MULTIPLICATIVE or EXPONENTIAL mode. (See MlSelector for details). If an output is given, mutants and their fitness values will be written to the output, in the form of 'mutant s h'.

3.6.5 Class MutSpaceRecombinator

This during mating operator recombine chromosomes, which records mutant locations, using a fixed recombination rate (per base pair).

class MutSpaceRecombinator(*rate, ranges, output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a Recombinator (a mendelian genotype transmitter with recombination and gene conversion) that passes genotypes from parents (or a parent in case of self-fertilization) to offspring. A recombination *rate* in the unit of base pair is needed.