simuPOP Reference Manual

Release 0.9.1 (Rev: 2407)

Bo Peng

December 2004

Last modified
February 2, 2009

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 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. Statistics of populations can be calculated and visualized dynamically which makes simuPOP an ideal tool to demonstrate population genetics models; generate datasets under various evolutionary settings, and more importantly, study complex evolutionary processes and evaluate gene mapping methods.

simuPOP is written in C++ and is provided as Python modules. It provides a large number of building blocks (populations, mating schemes, various genetic forces in the form of operators, simulators and gene mapping methods) to construct a simulation. This provides a R/Splus or Matlab-like environment where users can interactively create, manipulate and evolve populations, monitor and visualize population statistics and apply gene mapping methods. 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.

This reference manual lists all variables, functions, classes and utility modules of simuPOP. Please report any error to the simuPOP mailing list `simupop-list@lists.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 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 `individual`, `population`, and `simulator` 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. Although alleles at different loci usually have different names, simuPOP uses the same names for alleles across loci for simplicity.
- Names of loci and chromosomes.
- Names of information fields attached to each individual.

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

class `GenoStruTrait`()

A `GenoStruTrait` object is created with the creation of a `population` so it cannot be initialized directly.

`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.

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.

chromName(*chrom*)
Return the name of a chromosome *chrom*. Default to `chrom#` where # is the 1-based index of the chromosome.

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

chromType(*chrom*)
Return the type of a chromosome *chrom* (`Customized`, `Autosome`, `ChromosomeX`, or `ChromosomeY`).

chromTypes()
Return the type of all chromosomes (`Customized`, `Autosome`, `ChromosomeX` or `ChromosomeY`).

numChrom()
Return the number of chromosomes.

absLocusIndex(*chrom, locus*)
Return the absolute index of locus *locus* on chromosome *chrom*. An `IndexError` will be raised if *chrom* or *locus* is out of range. c.f. `chromLocusPair`.

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

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

lociDist(*loc1, loc2*)
Return the distance between loci *loc1* and *loc2* on the same chromosome. A negative value will be returned if *loc1* is after *loc2*.

lociNames()
Return the names of all loci specified by the *lociNames* parameter of the `population` function.

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.

locusName(*loc*)
Return the name of locus *loc* specified by the *lociNames* parameter of the `population` function. Default to `locX-Y` where `X` and `Y` are 1-based chromosome and locus indexes (`loc1-1`, `loc1-2`, ... etc)

locusPos(*loc*)
Return the position of locus *loc* specified by the *lociPos* parameter of the `population` function. An `IndexError` will be raised if the absolute index *loc* is greater than or equal to the total number of loci.

numLoci(*chrom*)
Return the number of loci on chromosome *chrom*, equivalent to `numLoci()[`*chrom*`]`.

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

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

alleleName(*allele*)
Return the name of allele *allele* specified by the *alleleNames* parameter of the `population` function. If the name of an allele is not specified, its index (`'0'`, `'1'`, `'2'`, etc) is returned. An `IndexError` will be raised if *allele* is larger than the maximum allowed allele state of this module (`MaxAllele()`).

alleleNames()
Return a list of allele names given by the *alleleNames* parameter of the `population` function. This list does not have to cover all possible allele states of a population so `alleleNames()[`*allele*`]` might fail (use `alleleNames(`*allele*`)` instead).

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.

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, are 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*`)`.

allele(*idx*)
Return the current allele at a locus, using its absolute index *idx*.

allele(*idx, p*)
Return the current allele at locus *idx* on the *p*-th set of homologous chromosomes.

allele(*idx, p, chrom*)
Return the current allele at locus *idx* on chromosome *chrom* of the *p*-th set of homologous chromosomes.

setAllele(*allele, idx*)
Set allele *allele* to a locus, using its absolute index *idx*.

setAllele(*allele, idx, p*)
Set allele *allele* to locus *idx* on the *p*-th homologous set of chromosomes.

setAllele(*allele, idx, p, chrom*)
Set allele *allele* to locus *idx* on chromosome *chrom* of the *p*-th homologous set of chromosomes.

genotype()
Return an editable array (a `carray` of length `totNumLoci()*ploidy()`) that represents all alleles of an individual.

genotype(*p*)
Return an editable array (a `carray` of length `totNumLoci()`) that represents all alleles on the *p*-th homologous set of chromosomes.

genotype(*p, chrom*)
Return an editable array (a `carrary` of legnth `numLoci(`*chrom*`)`) that represents all alleles on chromosome *chrom* of the *p*-th homologous set of chromosomes.

setGenotype(*geno*)
Fill the genotype of an individual using a list of alleles *geno*. `geno` will be reused if its length is less than `totNumLoci()*ploidy()`.

setGenotype(*geno, p*)
Fill the genotype of the *p*-th homologous set of chromosomes using a list of alleles *geno*. `geno` will be reused if its length is less than `totNumLoci()`.

setGenotype(*geno, p, chrom*)
Fill the genotype of chromosome *chrom* on the *p*-th homologous set of chromosomes using a list of alleles *geno*. `geno` will be reused if its length is less than `mumLoci(`*chrom*`)`.

setSex(*sex*)
Set individual sex to `Male` or `Female`.

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

sexChar()
Return the sex of an individual, `M` for male or `F` for `female`.

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

affectedChar()
Return `A` if this individual is affected, or `U` otherwise.

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

info(*idx*)
Return the value of an information field *idx* (an index).

info(*name*)
Return the value of an information field *name*.

intInfo(*idx*)
Return the value of an information field *idx* (an index) as an integer number.

intInfo(*name*)
Return the value of an information field *name* as an integer number.

setInfo(*value, idx*)
Set the value of an information field *idx* (an index) to *value*.

setInfo(*value, name*)
Set the value of an information field *name* to *value*.

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. Default to `[1]`, meaning one chromosome with a single locus.

chromTypes: A list that specifies the type of each chromosome, which can be `Autosome`, `ChromosomeX`, `ChromosomeY`, 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). A nested list that specifies positions of loci on each chromosome is also acceptable.

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 `chrom1`, `chrom2`, ... etc.

alleleNames: A list of allele names for all markers. For example, `alleleNames=('A','C','T','G')` names allele `0` – `3A`, `C`, `T`, and `G` respectively.

lociNames: A list or a matrix (separated by chromosomes) of names for each locus. Default to `"locX-Y"` where `X` and `Y` are 1-based chromosome and locus indexes, respectively. 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.

Note simuPOP does not yet support locus-specific allele names.

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

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

subPopBegin(*subPop*)
Return the index of the first individual in subpopulation *subPop*. An `IndexError` will be raised if *subPop* is out of range.

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`.

setSubPopName(*name, subPop*)
Assign a name *name* to subpopulation *subPop*. *does* not have to be unique.

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.

subPopName(*subPop*)
Return the name of a subpopulation *subPop*, and `'unnamed'` if no name is assigned to *subPop*. If *subPop* is a virtual subpopulation (specified by a `(sp, vsp)` pair), a combined name such as `subPop1 - Male` is returned.

subPopNames()
Return the names of all subpopulations (excluding virtual subpopulations). `'unnamed'` will be returned for unnamed subpopulations.

popSize()
Return the total number of individuals in all subpopulations.

subPopSize(*subPop*)
Return the size of a subpopulation (`subPopSize(sp)`) or a virtual subpopulation (`subPopSize([sp, vsp])`).

subPopSizes()
Return the sizes of all subpopulations in a list. Virtual subpopulations are not considered.

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

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.

individual(*idx*)
Return a refernce to individual *ind* in the population.

individual(*idx, subPop*)
Return a refernce to individual *ind* in subpopulation *subPop*.

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

individuals(*subPop*)
Return an iterator that can be used to iterate through all individuals in a subpopulation (`subPop=spID`) or a virtual subpopulation (`subPop=[spID, vspID]`).

genotype()
Return an editable array of the genotype of all individuals in this population.

genotype(*subPop*)
Return an editable array of the genotype of all individuals in subpopulation *subPop*. Virtual subpopulation is unsupported.

setGenotype(*geno*)
Fill the genotype of all individuals of a population using a list of alleles *geno*. *geno* will be reused if its length is less than `popSize()*totNumLoci()*ploidy()`.

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

ancestor(*idx, gen*)
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`).

ancestor(*ind, subPop, gen*)
Return a reference to individual `idx` of subpopulation *subPop* in ancestral generation `gen`.

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

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*, advancing the greatness level of all existing ancestral generations by one. If *ancestralDepth* is positive and there are already *ancestralDepth* ancestral generations (see also:

`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.

`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.

`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.

`addChrom`(*lociPos, lociNames=[], chromName="", 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. If they are not specified, simuPOP will try to assign default names, and raise a `ValueError` if the default names have been used.

`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.

`addLoci`(*chrom, pos, names=[]*)
Insert loci *names* at positions *pos* on chromosome *chrom*. These parameters should be lists of the same length, although *names* may be ignored, in which case random names will be given. 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. 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.

`extract`(*field=None, loci=None, infoFields=None, ancGen=-1*)
Extract subsets of individuals, loci and/or information fields from the current population and create a new one. If information field *field* is not `None`, individuals with negative values at this information field will be removed, and others are put into subpopulations specified by this field. The extracted population will keep the original subpopulation names if two populations have the same number of subpopulations. If *loci* is not `None`, only genotypes at *loci* are extracted. If *infoFields* is not `None`, only these information fields will be extracted. If *ancGen* is not `-1` (default, meaing all ancestral generations), only *ancGen* ancestral generations will be kept. As an advanced feature, *field* can be information field of a pedigree object *ped*. This allows extraction of individuals according to pedigrees identified in a pedigree object. This pedigree should have the same number of individuals in all generations.

`mergeSubPops`(*subPops=[]*)
Merge subpopulations *subPops*. If *subPops* is empty (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.

removeIndividuals (*inds*)
Remove individual(s) *inds* (absolute indexes) from the current population. A subpopulation will be kept even if all individuals from it are removed. This function only affects the current generation.

removeLoci (*loci=[], keep=[]*)
Remove *loci* (absolute indexes) 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 subpopulation(s) *subPop* and all their individuals. Indexes of subpopulations after removed subpopulations will be shifted.

resize (*size, propagate=False*)
Resize population by giving new subpopulation sizes *size*. 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.

setSubPopByIndInfo (*field*)
Rearrange individuals to their new subpopulations according to their integer values at information field *field* (value returned by `individual::indInfo(field)`). Individuals with negative values at this *field* will be removed. Existing subpopulation names are unchanged but new subpopulations will not assign a name (`'unnamed'`).

splitSubPop (*subPop, sizes*)
Split subpopulation *subPop* into subpopulations of given *sizes*, which should add up to the size of subpopulation *subPop*. Alternatively, *sizes* can be a list of proportions (add up to `1`) from which the sizes of new subpopulations are determined. 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 split subpopulations.

addInfoField (*field, init=0*)
Add an information field *field* to a population and initialize its values to *init*.

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.

indInfo (*idx*)
Return the information field `idx` (an index) of all individuals as a list.

indInfo (*name*)
Return the information field `name` of all individuals as a list.

indInfo (*idx, subPop*)
Return the information field `idx` (an index) of all individuals in (virtual) subpopulation *subPop* as a list.

indInfo (*name, subPop*)
Return the information field `name` of all individuals in (virtual) subpopulation *subPop* as a list.

setIndInfo (*values, idx*)
Set information field `idx` (an index) of the current population to *values*. *values* will be reused if its length is smaller than `popSize()`.

setIndInfo (*values, name*)
Set information field `name` of the current population to *values*. *values* will be reused if its length is smaller than `popSize()`.

setIndInfo (*values, idx, subPop*)
Set information field `idx` (an index) of a subpopulation (`subPop=sp`) or a virtual subpopulation (`subPop=[sp, vsp]`) to *values*. *values* will be reused if its length is smaller than `subPopSize(subPop)`.

setIndInfo (*values, name, subPop*)
Set information field `name` of a subpopulation (`subPop=sp`) or a virtual subpopulation (`subPop=[sp, vsp]`) to *values*. *values* will be reused if its length is smaller than `subPopSize(subPop)`.

setInfoFields (*fields, init=0*)
Set information fields *fields* to a population and initialize them with value *init*. All existing information fields will be removed.

updateInfoFieldsFrom (*fields, pop, fromFields=[], ancGen=-1*)
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 *ancGen* is not `-1`, only the most recent *ancGen* generations are updated.

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

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

vars ()
Return variables of a population as a Python dictionary.

vars (*subPop*)
Return a dictionary `vars()["subPop"][subPop]`. *subPop* can be a number (`subPop=spID`), or a pair of numbers (`subPop=(spID, vspID)`). A `ValueError` will be raised if key `'subPop'` does not exist in `vars()`, or if key *subPop* does not exist in `vars()["subPop"]`.

dvars ()
Return a wrapper of Python dictionary returned by `vars()` so that dictionary keys can be accessed as attributes. For example `pop.dvars().alleleFreq` is equivalent to `pop.vars()["alleleFreq"]`.

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

1.1.4 Class `simulator`

A simuPOP simulator is responsible for evolving one or more replicates of a *population* forward in time, subject to various *operators*. Populations in a simulator are created as identical copies of a population and will become different after evolution. A *mating scheme* needs to be specified, which will be used to generate offspring generations during evolution. A number of functions are provided to access simulator properties, access populations and their variables, copy, save and load a simulator.

The most important member function of a simulator is `evolve`, which evolves populations forward in time, subject to various *operators*. For convenience, member functions are provided to set virtual splitter, add information field and set ancestral depth to all populations in a simulator.

class simulator (*pop, matingScheme, rep=1*)
Create a simulator with *rep* replicates of population *pop*. Population *pop* will be copied *rep* times (default to `1`), while keeping the passed population intact. A mating scheme *matingScheme* will be used to evolve these populations.

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.

save(*filename*)
Save a simulator to file `filename`, which can be loaded by a global function `LoadSimulator`.

gen()
Return the current generation number, which is the initial generation number (`0`, or some value set by `setGen(gen)`) plus the total number of generations evolved.

setGen(*gen*)
Set the current generation number of a simulator to *gen*.

evolve(*ops, preOps=[], postOps=[], gen=-1, dryrun=False*)
Evolve all populations *gen* generations, subject to operators *opspreOps* and *postOps*. Operators *preOps* are applied to all populations (subject to applicability restrictions of the operators, imposed by the *rep* parameter of these operators) before evolution. They are usually used to initialize populations. Operators *postOps* are applied to all populations after the evolution.

Operators *ops* are applied during the life cycle of each generation. Depending on the stage of these operators, they can be applied before-, during-, and/or post-mating. These operators can be applied at all or some of the generations, depending the *begin*, *end*, *step*, and *at* parameters of these operators. Populations in a simulator are evolved one by one. At each generation, the applicability of these operators are determined. Pre-mating operators are applied to a population first. A mating scheme is then used to populate an offspring generation, using applicable during-mating operators. After an offspring generation is successfully generated and becomes the current generation, applicable post-mating operators are applied to it. Because the order at which operators are applied can be important, and the stage(s) at which operators are applied are not always clear, a parameter *dryRun* can be used. If set to `True`, this function will print out the order at which all operators are applied, without actually evolving the populations.

Parameter *gen* can be set to a positive number, which is the number of generations to evolve. If *gen* is negative (default), the evolution will continue indefinitely, until all replicates are stopped by a special kind of operators called *terminators*. At the end of the evolution, the generations that each replicates have evolved are returned. If not all replicates are stopped at the same generation, the negative replicate numbers are calculated according to *active* replicates, meaning replicate -1 will refer to the last active replicate even if the last replicate has stopped. In addition, *postOps* are applied to all replicates, including those that stopped before other replicates.

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). Modifying the returned object is discouraged because it will change the population within the simulator. If an independent copy of the population is needed, use `simu.population(rep).clone()`.

add(*pop*)
Add a population *pop* to the end of an existing simulator. This creates an cloned copy of *pop* in the simulator so the evolution of the simulator will not change *pop*.

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

setMatingScheme(*matingScheme*)
Set a new mating scheme *matingScheme* to a simulator.

vars(*rep*)
Return the local namespace of the *rep-th* population, equivalent to `x.population(rep).vars()`.

vars(*rep, subPop*)
Return a dictionary of subpopulation variables in the local namespace of the *rep-th* population, equivalent to `x.population(rep).vars(subPop)`.

dvars(*rep*)
Return a wrapper of Python dictionary returned by `vars(rep)` so that dictionary keys can be accessed as attributes. For example `simu.dvars(1).alleleFreq` is equivalent to `simu.vars(1)["alleleFreq"]`.

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

1.1.5 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.

A pedigree class can be created from a population, or loaded from a disk file, which is usually saved by an operator during a previous evolutionary process. Depending on how a pedigree is saved, sex and affection status information may be missing.

class pedigree(*pop, loci=[], infoFields=[], ancGen=-1, fatherField="father_idx", motherField="mother_idx"*)
Create a pedigree object from a population, using a subset of loci (parameter *loci*, default to no loci), information fields (parameter *infoFields*, default to no information field except for *parentFields*), and ancestral generations (parameter *ancGen*, default to all ancestral generations). By default, information field `father_idx` and `mother_idx` are used to locate parents. If individuals in a pedigree has only one parent, the information field that stores parental indexes should be specified in parameter *fatherField* or *motherField*. The other field should be set to an empty string.

clone()
Create a cloned copy of a pedigree.

numParents()
Return the number of parents each individual has. This function returns the number of information fields used to store parental indexes, even if one of the fields are unused.

father(*idx, subPop*)
Return the index of the father of individual *idx* in subpopulation *subPop* in the parental generation. Return `-1` if this individual has no father (`fatherField` is empty or the value of information field is negative).

mother(*idx, subPop*)
Return the index of the mother of individual *idx* in subpopulation *subPop* in the parental generation. Return `-1` if this individual has no mother (`motherField` is empty or the value of information field is negative).

locateRelatives(*relType=[], relFields=[], ancGen=-1*)
This function locates relatives (of type *relType*) of each individual and store their indexes in specified information fields *relFields*. The length of *relFields* determines how many relatives an individual can have.

Parameter *relType* specifies what type of relative to locate. It can be `Self`, `Spouse` (having at least one common offspring), `Offspring`, `FullSibling` (having common father and mother), or `Sibling` (having at least one common parent). Optionally, you can specify the sex of relatives you would like

to locate, in the form of `relType=(type, sexChoice)`. sexChoice can be `AnySex` (default), `MaleOnly`, `FemaleOnly`, `SameSex` or `OppositeSex`.

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

traceRelatives (*pathGen, pathFields, pathSex=[], resultFields=[]*)

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 three pieces of information specified by three parameters. Parameter *pathGen* specifies starting, intermediate and ending generations. *pathFields* specifies which information fields to look for at each step, and *pathSex* specifies sex choices at each generation, which should be a list of `AnySex`, `MaleOnly`, `FemaleOnly`, `SameSex` and `OppsiteSex`. The default value for this paramter is `AnySex` at all steps. The length of *pathGen* should be one more than *pathFields*, and *pathSex* if *pathSex* is given.

For example, if `pathGen=[0, 1, 1, 0]`, `pathFields = [['father_idx', 'mother_idx'], ['sib1', 'sib2'], ['off1', 'off2']]`, and `pathSex = [AnySex, MaleOnly, FemaleOnly]`, this function will locate `father_idx` and `mother_idx` for each individual at generation 0, find all individuals referred by `father_idx` and `mother_idx` at generation 1, 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 at the present geneartion. The results are father or mother's brother's daughters. Their indexes will be saved in each individuals information fields *resultFields*. Note that this function will locate and set relatives for individuals only at the starting generation specified at `pathGen[0]`.

1.2 Virtual subpopulation splitters

1.2.1 Class `vspSplitter`

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.

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

class vspSplitter()

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.

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`()
Create a sex splitter that defines male and female VSPs.

`name`(*vsp*)
Return `"Male"` if *vsp=0* and `"Female"` otherwise.

`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`()
Create a splitter that defined two VSPs by affection status.

`name`(*vsp*)
Return `"Unaffected"` if *vsp=0* and `"Affected"` if *vsp=1*.

`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=[]*)
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*. Of course, only one of the parameters *values* and *cutoff* should be defined, values in *cutoff* should be distinct, and in an increasing order.

`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.

`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=[]*)

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`.

name(*vsp*)

Return the name of VSP *vsp*, which is `"Prop p"` where `p=propotions[vsp]`.

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*)

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.

name(*vsp*)

Return the name of VSP *vsp*, which is `"Range [a, b]"` where `[a, b]` is range `ranges[vsp]`.

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*)

Create a splitter that defines VSPs by individual genotype at *loci* (a locus index or a list of loci indexes). 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.

name(*vsp*)

Return name of VSP *vsp*, which is `"Genotype loc1,loc2:genotype"` as defined by parameters *loci* and *alleles*.

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. This splitter is usually used to define different types of VSPs to a population.

class `combinedSplitter` (*splitters=[]*)

Create a combined splitter using a list of *splitters*. For example, `combinedSplitter([sexSplitter(), affectionSplitter()])` defines a combined splitter with four VSPs.

`name` (*vsp*)

Return the name of a VSP *vsp*, which is the name a VSP defined by one of the combined splitters.

`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 Basic Mating Schemes

1.3.1 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=[], subPop=[], 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. The function should take two parameters, a generation number and a list of subpopulation sizes before mating, and return a list of subpopulation sizes for the offspring generation. A single number can be returned if there is only one subpopulation. If latter form is used, the specified function will be called at each generation to determine the size of the offspring generation. Parameters *subPop* and *weight* are ignored in this case.

If this mating shcme is used within a heterogeneous mating scheme. Parameters *subPop* and *weight* are used to determine which (virtual) subpopulation 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.

`clone` ()

Deep copy of a homogeneous mating scheme.

1.3.2 Class `heteroMating`

A heterogeneous mating scheme that applies a list of 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 `homoMating` for a detailed explanation of this parameter.

Each mating scheme defined in *matingSchemes* can be applied to one or more (virtual) subpopulation. If parameter *subPop* is not specified, a mating scheme will be applied to all subpopulations. If a (virtual) subpopulation is specified, a mating scheme will be applied to a specific (virtual) subpopulation. A special case is when *subPop* is given as `(-1, vsp)`. In this case, the mating scheme will be applied to virtual subpopulation `vsp` in all 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`.

clone()
Deep copy of a heterogeneous mating scheme

1.3.3 Class `pedigreeMating`

A pedigree mating scheme that evolves a population following a pedigree object.

class pedigreeMating(*ped, generator, setSex=False, setAffection=False, copyFields=[]*)
Creates a mating scheme that evolve a population following a pedigree object *ped*. Considering this pedigree as a population with `N` ancestral generations, the starting population is the greatest ancestral generation of *ped*. The mating scheme creates an offspring generation that match the size of generation `N-1` and chooses parents according to the parents of individuals at this generation. Depending on the *gen* parameter of the simulator, the process continues generation by generation for `N` generations if `gen` >= N), or `gen` generations if `gen` < `N`. During the evolution, an offspring generator *generator* is used to produce one offspring at a time, regardless of the *numOffspring* setting of this offspring generator. If individuals in pedigree *ped* has only one parent, the offspring generator should be compatible.

By default, the pedigree mating scheme does not set offspring sex and affection status using sex and affection status of corresponding individuals in the pedigree. However, if such information is valid in the pedigree object *ped*, you can set parameters *setSex* and/or *setAffection* to `True` to set sex and/of affection status to offspring during the evolutionary process. Similarly, you can specify some information fields in *copyFields* to copy some information fields from pedigree to the evolving population. Note that these information will be copied also to the starting population (from the greatest ancestral generation in *ped*).

clone()
Deep copy of a Python mating scheme

1.4 Parent choosers and offspring generators

1.4.1 Class `sequentialParentChooser`

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

class sequentialParentChooser()
Create a parent chooser that chooses a parent from a parental (virtual) subpopulation sequentially.

clone()
Deep copy of a sequential parent chooser.

1.4.2 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.

clone()
Deep copy of a sequential parents chooser.

1.4.3 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 natural selection is enabled, the probability that an individual is chosen is proportional to his/her fitness value stored in an information field *selectionField* (default to `"fitness"`). If parents are chosen without replacement, a parent can be chosen only once. An `RuntimeError` will be raised if all parents are exhausted. Selection is disabled in the without-replacement case.

class randomParentChooser(*replacement=True, selectionField="fitness"*)
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*.

clone()
Deep copy of a random parent chooser.

1.4.4 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 natural selection is enabled, the probability that an individual is chosen is proportional to his/her fitness value among all individuals with the same sex. Selection will be disabled if specified information field *selectionField* (default to `"fitness"`) does not exist.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. 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*.

clone()
Deep copy of a random parents chooser.

1.4.5 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 natural selection is enabled, the probability that an individual is chosen is proportional to his/her fitness value among all individuals with the same sex. Selection will be disabled if specified information field *selectionField* (default to `"fitness"`) does not exist.

`clone`()

Deep copy of a parent chooser.

1.4.6 Class `alphaParentsChooser`

This parent chooser mimicks some animal populations where only certain individuals (usually males) can mate. Alpha individuals can be chosen either randomly (with natural selection) or according to an information field. After the alpha individuals are selected, the parent chooser works identical to a random mating scheme, except that one of the parents are chosen from these alpha individuals.

class `alphaParentsChooser` (*alphaSex=Male, alphaNum=0, alphaField=string, selectionField="fitness"*)

Create a parent chooser that chooses father (if *alphaSex* is `Male`) or mother (if *alphaSex* is `Female`) from a selected group of alpha individuals. If *alphaNum* is given, alpha individuals are chosen randomly or according to individual fitness if natural selection is enabled. If *alphaField* is given, individuals with non-zero values at this information field are considered as alpha individuals. After alpha individuals are selected, *alphaSex* parent will be chosen from the alpha individuals randomly or according to individual fitness. The other parents are chosen randomly.

`clone`()

Deep copy of an alpha parents chooser.

1.4.7 Class `infoParentsChooser`

This parent chooser chooses an individual randomly, and then his/her spouse his/her spouse from a given set of information fields, which stores indexes of individuals in the same generation. An information field will be ignored if its value is negative, or if sex is incompatible.

Depending on what indexes are stored in these information fields, this parent chooser can be used to implement different types of mating schemes where selection of spouse is limited. For example, a consanguineous mating scheme can be implemeneted using this mating scheme if certain type of relatives are located for each individual, and are used for mating.

This parent chooser uses `randomParentChooser` to choose one parent and randomly choose another one from the information fields. Natural selection is supported during the selection of the first parent. Because of potentially uneven distribution of valid information fields, the overall process may not be as random as expected.

class `infoParentsChooser` (*infoFields=[], func=None, param=None, selectionField="fitness"*)

Create a information parent chooser a parent randomly (with replacement, and with selection if natural selection is enabled), and then his/her spouse from indexes stored in *infoFields*. If a Python function *func* is specified, it will be called before parents are chosen. This function accepts the parental population and an optional parameter *param* and is usually used to locate qualified spouse for each parent. The return value of this function is ignored.

clone()
Deep copy of a infomation parent chooser.

1.4.8 Class pyParentsChooser

This parents chooser accept a Python generator function that repeatedly yields an index (relative to each subpopulation) of a parent, or indexes of two parents as a Python list of tuple. The parent chooser calls the generator function with parental population and a subpopulation index for each subpopulation and retrieves indexes of parents repeatedly using the iterator interface of the generator function.

This parent chooser does not support virtual subpopulation directly. A `ValueError` will be raised if this parent chooser is applied to a virtual subpopulation. 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(*parentsGenerator*)
Create a Python parent chooser using a Python generator function *parentsGenerator*. This function should accept a population object (the parental population) and a subpopulation number and return the index of a parent or a pair of parents repeatedly using the iterator interface of the generator function.

clone()
Deep copy of a python parent chooser.

1.4.9 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=RandomSex*)
Create a basic offspring generator. This offspring generator uses *ops* genotype transmitters to transmit genotypes from parents to offspring. It expects *numParents* from an upstream parents chooser and raises an `RuntimeError` if incorrect number of parents are passed. If both one and two parents can be handled, `0` should be specified for this parameter.

A number of *genotype transmitters* can be used to transmit genotype from parents to offspring. Additional during-mating operators can be passed from the `evolve()` function of a *simulator*, but the *ops* operators will be applied before them. An exception is that if one of the passed operators is set to form offspring genotype (a flag `setOffGenotype`), operators in *ops* with the same flag will not be applied. For example, a `recombinator` will override a `mendelianGenoTransmitter` used in `randomMating` if it is used in the `ops` parameter of the `evolve` function. This general offspring generator does not use any genotype transmitter. A number of derived offspring generators are available with a default transmitter. For example, a `mendelianOffspringGenerator` uses a `mendelianGenoTransmitter` to transmit genotypes.

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 function, 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. Current generation number will be passed to this function, and its return value will be considered the number of offspring. In the last case, a tuple (or a list) in one of the following forms: `(GeometricDistribution, p)`, `(PoissonDistribution, p)`, `(BinomialDistribution, p, N)`, or `(UniformDistribution, a, b)` can be given. The number of offspring will be determined randomly following these statistical distributions. Please refer to the simuPOP user's guide for a detailed description of these distribution and their parameters.

Parameter *sexMode* is used to control the sex of each offspring. Its default value is usually *RandomSex* which assign `Male` or `Female` to each individual randomly, with equal probabilities. If `NoSex` is given, all individuals

will be `Male`. *sexMode* can also be one of `(ProbOfMale, p)`, `(NumOfMale, n)`, and `(NumOfFemale, n)`. The first case specifies the probability of male for each offspring. The next two cases specifies the number of male or female individuals in each family, respectively. If `n` is greater than or equal to the number of offspring in this family, all offspring in this family will be `Male` or `Female`.

clone()
Make a deep copy of this offspring generator.

1.4.10 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) *Forward-time simulations of populations with complex human diseases*, PLoS Genetics".

class controlledOffspringGenerator(*loci, alleles, freqFunc, ops=[], numOffspring=1, sexMode=RandomSex*)
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*. If there are multiple subpopulations, *freqFunc* can return a list of allele frequencies for each subpopulation, or a list of allele frequencies in the whole population. 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 multi-nomial 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*.

clone()
Deep copy of a controlled random mating scheme.

mendelianOffspringGenerator(*ops=[], *args, **kwargs*)
An offspring generator that uses `mendelianGenoTransmitter()` as a default genotype transmitter. Additional during mating operators can be specified using the *ops* parameter. Other parameters are passed directly to `offspringGenerator`.

Chapter 2

Operator References (under revision)

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.

class `baseOperator` (*output, stage, begin, end, step, at, rep, 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'`).

stage: Stage(s) of a life cycle at which an operator will be applied. It can be `PreMating`, `DuringMating`, `PostMating` or any of their combined stages `PrePostMating`, `PreDuringMatingDuringPostMating` and `PreDuringPostMating`. Note that all operators have their default stage parameter and some of them ignore this parameter because they can only be applied at certain stage(s) of a life cycle.

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.

rep: A list of applicable replicates. An empty list (default) 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)]`. An empty list (default) is interpreted as all subpopulations. `subPops=[sp1]` can be simplied as `subPops=sp1`. Negative indexes are not supported. Suport for this parameter vary from operator to operator. Some operators do not support virtual subpopulations and some operators do not support this parameter 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` (Function `InitSex`)

This operator initialize sex of individuals, either randomly or use a list of sexes. For convenience, the function of this operator is included in other *initializers* such as `initByFreq` and `initByValue` so that you do not have to intiailize sexes separately from genotype.

class initSex(*maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create an operator that initialize individual sex to `Male` or `Female`. By default, it assign sex to individuals randomly, with equal probability of having a male or a female. This probabability can be adjusted through parameter *maleFreq*. 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* is ignored if *sex* is given. If a list of (virtual) subpopulation is specified in parameter *subPop*, only individuals in these subpopulations will be initialized.

apply(*pop*)
Apply this operator to population *pop*

clone()
Deep copy of an `initSex` operator.

2.2.2 Class `initByFreq` (Function `InitByFreq`)

This operator assigns alleles at all or part of loci with given allele frequencies. Alternatively, an individual can be initialized and be copied to all individuals in the same (virtual) subpopulations.

class `initByFreq`(*alleleFreq=[], loci=[], ploidy=[], identicalInds=False, initSex=True, maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

This function creates an initializer that initializes individual genotypes randomly, using allele frequencies specified in parameter *alleleFreq*. Elements in *alleleFreq* specifies the allele frequencies of allele `0`, `1`, ... respectively. These frequencies should add up to `1`. If *loci*, *ploidy* and/or *subPop* are specified, only specified loci, ploidy, and individuals in these (virtual) subpopulations will be initialized. If *identicalInds* is `True`, the first individual in each (virtual) subpopulation will be initialized randomly, and be copied to all other individuals in this (virtual) subpopulation. If a list of frequencies are given, they will be used for each (virtual) subpopulation. If *initSex* is `True` (default), `initSex(maleFreq, sex)` will be applied. This operator initializes all chromosomes, including unused genotype locations and customized chromosomes.

`apply`(*pop*)

Apply this operator to population *pop*

`clone`()

Deep copy of the operator `initByFreq`

2.2.3 Class `initByValue` (Function `InitByValue`)

This operator initialize individuals by given values.

class `initByValue`(*value=[], loci=[], ploidy=[], proportions=[], initSex=True, maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

This function creates an initializer that initializes individual genotypes with given genotype *value*. If *loci*, *ploidy* and/or *subPop* are specified, only specified loci, ploidy, and individuals in these (virtual) subpopulations will be initialized. *value* can be used to initialize given *loci*, all loci, and all homologous copies of these loci. If *proportions* (a list of positive numbers that add up to `1`) is given, *value* should be a list of values that will be assigned randomly according to their respective proportion. If a list of values are given without *proportions*, they will be used for each (virtual) subpopulations. If *initSex* is `True` (default), `initSex(maleFreq, sex)` will be applied. This operator initializes all chromosomes, including unused genotype locations and customized chromosomes.

`apply`(*pop*)

Apply this operator to population *pop*

`clone`()

Deep copy of the operator `initByValue`

2.3 Standard genotype transmitters

2.3.1 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`(*begin=0, end=-1, step=1, at=[], rep=[], subPops=[], 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*.

clone ()
Deep copy of a base genotype transmitter.

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*.

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.

initialize (*pop*)
Initialize a base genotype operator for a population. This function should be called before any other functions are used to transmit genotype.

2.3.2 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.

class cloneGenoTransmitter (*begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a clone genotype transmitter.

clone ()
Deep copy of a clone genotype transmitter.

2.3.3 Class `mendelianGenoTransmitter`

Mendelian offspring generator accepts two parents and pass their genotype to a number of offspring following Mendelian's law. Basically, one of the paternal chromosomes is chosen randomly to form the paternal copy of the offspring, and one of the maternal chromosome is chosen randomly to form the maternal copy of the offspring. The number of offspring produced is controled by parameters `numOffspring`, `numOffspringFunc`, `maxNumOffspring` and `mode`. Recombination will not happen unless a during-mating operator recombinator is used.

class mendelianGenoTransmitter (*begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a Mendelian genotype transmitter.

clone ()
Deep copy of a Mendelian genotype transmitter.

initialize (*pop*)
Initialize a base genotype operator for a population. This function should be called before function `transmitGenotype` is used to transmit genotype.

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`.

2.3.4 Class `selfingGenoTransmitter`

Selfing offspring generator works similarly as a mendelian offspring generator but a single parent produces both

the paternal and maternal copy of the offspring chromosomes. This offspring generator accepts a dipload parent. A random copy of the parental chromosomes is chosen randomly to form the parental copy of the offspring chromosome, and is chosen randomly again to form the maternal copy of the offspring chromosome.

class selfingGenoTransmitter(*begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a self-fertilization genotype transmitter.

clone()
Deep copy of a selfing genotype transmitter.

2.3.5 Class haplodiploidGenoTransmitter

Haplodiploid offspring generator mimics sex-determination in honey bees. Given a female (queen) parent and a male parent, the female is considered as diploid with two set of chromosomes, and the male is condiered as haploid. Actually, the first set of male chromosomes are used. During mating, female produce eggs, subject to potential recombination and gene conversion, while male sperm is identical to the parental chromosome.

Female offspring has two sets of chromosomes, one from mother and one from father. Male offspring has one set of chromosomes from his mother.

class haplodiploidGenoTransmitter(*begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a haplodiploid genotype transmitter.

clone()
Deep copy of a haplodiploid transmitter.

2.3.6 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.

class mitochondrialGenoTransmitter(*chroms=[], begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Createa a mitochondrial genotype transmitter that treats all Customized chromosomes, or a list of chromosomes specified by *chroms*, as human mitochondrial chromosomes. It transmits these chromosomes randomly from the female parent to offspring of both sexes.

clone()
Deep copy of a mitochondrial genotype transmitter.

2.4 Expression and Statements

2.4.1 Class pyOutput

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

class pyOutput(*msg="", output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], 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*.

apply(*pop*)
Simply output some info

clone()
Deep copy of a *pyOutput* operator.

2.4.2 Class `pyEval` (Function `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=string, output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Crete 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.

apply(*pop*)
Apply the `pyEval` operator to population *pop*.

clone()
Deep copy of a `pyEval` operator

evaluate(*pop*)
Evaluate the expression and optional statements in the local namespace of population *pop* and return its result as a string.

2.4.3 Class `pyExec` (Function `PyExec`)

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

class pyExec(*stmts="", exposePop=string, output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], 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 variable will be removed after the statements are executed.

clone()
Deep copy of a `pyExec` operator

2.4.4 Class `infoEval` (Function `infoEval`)

Unlike operator pyEval and pyExec that work at the population level, in its local namespace, infoEval works at the individual level, working with individual information fields. is statement can change the value of existing information fields. Optionally, variables in population's local namespace can be used in the statement, but this should be used with caution.

class infoEval(*expr="", stmts="", usePopVars=False, exposePop=False, name="", output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Evaluate Python statements with variables being an individual's information fields The expression and statements will be executed for each individual, in a Python namespace (dictionary) where individual information fields are made available as variables. Population dictionary can be made avaialbe with option usePopVars. Changes to these variables will change the corresponding information fields of individuals.

Please note that, 1. If population variables are used, and there are name conflicts between information fields and variables, population variables will be overridden by information fields, without any warning. 2. Information fields are float numbers. An exceptions will raise if an information field can not be converted to a float number. 3. This operator can be used in all stages. When it is used during-mating, it will act on each offspring.

expr: The expression to be evaluated. The result will be sent to `output`.

stmts: The statement that will be executed before the expression

subPop: A shortcut to `subPops=[subPop]`

subPops: Subpopulations this operator will apply to. Default to all.

usePopVars: If `True`, import variables from expose the current population as a variable named `pop`

exposePop: If `True`, expose the current population as a variable named `pop`

name: Used to let pure Python operator to identify themselves

output: Default to >. I.e., output to standard output. Note that because the expression will be executed for each individual, the output can be large.

apply(*pop*)
Apply the `infoEval` operator

clone()
Deep copy of a `infoEval` operator

name()
Return the name of an expression The name of a `infoEval` operator is given by an optional parameter `name`. It can be used to identify this `infoEval` operator in debug output, or in the dryrun mode of simulator::evolve.

2.4.5 Class `infoExec` (Function `infoExec`)

Execute a Python statement for each individual, using information fields This operator takes a list of statements and executes them. No value will be returned or outputted.

class infoExec(*stmts="", usePopVars=False, exposePop=False, name="", output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Fields, optionally with variable in population's local namespace Please refer to class `infoEval` for parameter descriptions.

clone()
Deep copy of a `infoExec` operator

2.5 Migration

2.5.1 Class `migrator`

This operator migrates individuals from (virtual) subpopulations to other subpopulations, according to either prespecified 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. An `RuntimeError` will be raised if an individual is assigned to migrate more than once. This might happen if you are migrating from two overlapping virtual subpopulations.

class `migrator`(*rate=[], mode=ByProbability, toSubPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], 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* = `ByProbability`), proportions of individuals in the source (virtual) subpopulations to the destination subpopulations (*mode* = `ByProportion`), numbers of migrants in the source (virtual) subpopulations (*mode* = `ByCounts`), or ignored completely (*mode* = `ByIndInfo`). In the last case, parameter *subPops* is respected (only individuals in specified (virtual) subpopulations will migrate) but *toSubPops* is ignored.

`apply`(*pop*)

Apply the migrator to populaiton *pop*.

`clone`()

Deep copy of a migrator

`rate`()

Return migration rate

2.5.2 Class `splitSubPop` (Function `SplitSubPop`)

Split a subpopulation

class splitSubPop(*which=0, sizes=[], proportions=[], randomize=True, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["migrate_to"]*)

Split a subpopulation Split a subpopulation by sizes or proportions. Individuals are randomly (by default) assigned to the resulting subpopulations. Because mating schemes may introduce certain order to individuals, randomization ensures that split subpopulations have roughly even distribution of genotypes.

which: Which subpopulation to split. If there is no subpopulation structure, use `0` as the first (and only) subpopulation.

sizes: New subpopulation sizes. The sizes should be added up to the original subpopulation (subpopulation `which`) size.

proportions: Proportions of new subpopulations. Should be added up to `1`.

randomize: Whether or not randomize individuals before population split. Default to True.

apply(*pop*)

Apply a `splitSubPop` operator

clone()

Deep copy of a `splitSubPop` operator

2.5.3 Class `mergeSubPops` (Function `MergeSubPops`)

Merge subpopulations This operator merges subpopulations `subPops` to a single subpopulation. If `subPops` is ignored, all subpopulations will be merged.

class mergeSubPops(*subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Merge subpopulations

subPops: Subpopulations to be merged. Default to all.

apply(*pop*)

Apply a `mergeSubPops` operator

clone()

Deep copy of a `mergeSubPops` operator

2.5.4 Class `resizeSubPops` (Function `ResizeSubPops`)

Resize subpopulations This operator resize subpopulations `subPops` to a another size. If `subPops` is ignored, all subpopulations will be resized. If the new size is smaller than the original one, the remaining individuals are discarded. If the new size if greater, individuals will be copied again if propagate is true, and be empty otherwise.

class resizeSubPops(*newSizes=[], propagate=True, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Resize subpopulations

newSizes: Of the specified (or all) subpopulations.

subPop: Subpopulations to be resized. Default to all.

propagate: If true (default) and the new size if greater than the original size, individuals will be copied over.

apply(*pop*)

Apply a `resizeSubPops` operator

clone()

Deep copy of a `resizeSubPops` operator

2.6 Mutation

2.6.1 Class `mutator`

Base class of all mutators. The base class of all functional mutators. It is not supposed to be called directly.

Every mutator can specify `rate` (equal rate or different rates for different loci) and a vector of applicable loci (default to all but should have the same length as `rate` if `rate` has length greater than one).

Maximum allele can be specified as well but more parameters, if needed, should be implemented by individual mutator classes.

There are numbers of possible allelic states. Most theoretical studies assume an infinite number of allelic states to avoid any homoplasy. If it facilitates any analysis, this is however extremely unrealistic.

class `mutator` (*rate=[], loci=[], maxAllele=0, output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create a mutator, do not call this constructor directly All mutators have the following common parameters. However, the actual meaning of these parameters may vary according to different models. The only differences between the following mutators are the way they actually mutate an allele, and corresponding input parameters. The number of mutation events at each locus is recorded and can be accessed from the `mutationCount` or `mutationCounts` functions.

rate: Can be a number (uniform rate) or an array of mutation rates (the same length as `loci`)

loci: A vector of locus indexes. Will be ignored only when single rate is specified. Default to all loci.

maxAllele: Maximum allowed allele. Interpreted by each sub mutator class. Default to `pop.maxAllele()`.

`apply` (*pop*)
Apply a mutator

`clone` ()
Deep copy of a `mutator`

`maxAllele` ()
Return maximum allowable allele number

`mutate` (*allele*)
Describe how to mutate a single allele

`mutationCount` (*locus*)
Return mutation count at `locus`

`mutationCounts` ()
Return mutation counts

`rate` ()
Return the mutation rate

`setMaxAllele` (*maxAllele*)
Set maximum allowable allele

`setRate` (*rate, loci=[]*)
Set an array of mutation rates

2.6.2 Class `kamMutator` (Function `KamMutate`)

K-Allele Model mutator. This mutator mutate an allele to another allelic state with equal probability. The specified mutation rate is actually the 'probability to mutate'. So the mutation rate to any other allelic state is actually $\frac{rate}{K-1}$, where K is specified by parameter `maxAllele`.

class kamMutator(*rate=[], loci=[], maxAllele=0, output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create a K-Allele Model mutator Please see class `mutator` for the descriptions of other parameters.

rate: Mutation rate. It is the 'probability to mutate'. The actual mutation rate to any of the other `K-1` allelic states are `rate/(K-1)`.

maxAllele: Maximum allele that can be mutated to. For binary libraries, allelic states will be `[0, maxAllele]`. Otherwise, they are `[1, maxAllele]`.

clone()

Deep copy of a `kamMutator`

mutate(*allele*)

Mutate to a state other than current state with equal probability

2.6.3 Class smmMutator (Function SmmMutate)

The stepwise mutation model. The *Stepwise Mutation Model* (SMM) assumes that alleles are represented by integer values and that a mutation either increases or decreases the allele value by one. For variable number tandem repeats(VNTR) loci, the allele value is generally taken as the number of tandem repeats in the DNA sequence.

class smmMutator(*rate=[], loci=[], maxAllele=0, incProb=0.5, output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create a SMM mutator The SMM is developed for allozymes. It provides better description for these kinds of evolutionary processes.

Please see class `mutator` for the descriptions of other parameters.

incProb: Probability to increase allele state. Default to `0.5`.

clone()

Deep copy of a `smmMutator`

2.6.4 Class gsmMutator (Function GsmMutate)

Generalized stepwise mutation model The *Generalized Stepwise Mutation model* (GSM) is an extension to the stepwise mutation model. This model assumes that alleles are represented by integer values and that a mutation either increases or decreases the allele value by a random value. In other words, in this model the change in the allelic state is drawn from a random distribution. A *geometric generalized stepwise model* uses a geometric distribution with parameter p, which has mean $\frac{p}{1-p}$ and variance $\frac{p}{(1-p)^2}$.

`gsmMutator` implements both models. If you specify a Python function without a parameter, this mutator will use its return value each time a mutation occur; otherwise, a parameter p should be provided and the mutator will act as a geometric generalized stepwise model.

class gsmMutator(*rate=[], loci=[], maxAllele=0, incProb=0.5, p=0, func=None, output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create a `gsmMutator`The GSM model is developed for allozymes. It provides better description for these kinds of evolutionary processes.

Please see class `mutator` for the descriptions of other parameters.

incProb: Probability to increase allele state. Default to `0.5`.

func: A function that returns the number of steps. This function does not accept any parameter.

clone()

Deep copy of a `gsmMutator`

mutate(*allele*)

Mutate according to the GSM model

2.6.5 Class `pyMutator` (Function `PyMutate`)

A hybrid mutator. Parameters such as mutation rate of this operator are set just like others and you are supposed to provide a Python function to return a new allele state given an old state. `pyMutator` will choose an allele as usual and call your function to mutate it to another allele.

class `pyMutator` (*rate=[], loci=[], maxAllele=0, func=None, output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create a `pyMutator`

`clone`()

Deep copy of a `pyMutator`

`mutate`(*allele*)

Mutate according to the mixed model

2.6.6 Class `pointMutator` (Function `PointMutate`)

Point mutator Mutate specified individuals at specified loci to a spcified allele. I.e., this is a non-random mutator used to introduce diseases etc. `pointMutator`, as its name suggest, does point mutation. This mutator will turn alleles at `loci` on the first chromosome copy to `toAllele` for individual `inds`. You can specify `atPloidy` to mutate other, or all ploidy copies.

class `pointMutator` (*loci, toAllele, atPloidy=[], inds=[], output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create a `pointMutator` Please see class `mutator` for the descriptions of other parameters.

inds: Individuals who will mutate

toAllele: Allele that will be mutate to

`apply`(*pop*)

Apply a `pointMutator`

`clone`()

Deep copy of a `pointMutator`

`mutationCount`(*locus*)

Return mutation count at `locus`

`mutationCounts`()

Return mutation counts

2.7 Recombination and gene conversion

2.7.1 Class `recombinator`

In simuPOP, only one recombinator is provided. Recombination events between loci a/b and b/c are independent, otherwise there will be some linkage between loci. Users need to specify physical recombination rate between adjacent loci. In addition, for the recombinator

- it only works for diploid (and for females in haplodiploid) populations.
- the recombination rate must be comprised between `0.0` and `0.5`. A recombination rate of `0.0` means that the loci are completely linked, and thus behave together as a single linked locus. A recombination rate of `0.5` is equivalent to free of recombination. All other values between `0.0` and `0.5` will represent various linkage intensities between adjacent pairs of loci. The recombination rate is equivalent to `1-linkage` and represents the probability that the allele at the next locus is randomly drawn.

- it works for selfing. I.e., when only one parent is provided, it will be recombined twice, producing both maternal and paternal chromosomes of the offspring.
- conversion is allowed. Note that conversion will nullify many recombination events, depending on the parameters chosen.

class recombinator(*intensity=-1, rate=[], loci=[], convMode=NoConversion, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Recombine chromosomes from parents *convMode* can take the following forms NoConversion: no conversion (NumMarkers, prob, n): converts a fixed number of markers (GeometricDistribution, prob, p): An geometric distribution is used to determine how many markers will be converted. (TractLength, prob, n): converts a fixed length of tract. (ExponentialDistribution, prob, p): An exponential distribution with parameter `convLen` will be used to determine track length. The first number is that probability of conversion event among all recombination events. When a recombination event happens, it may become a recombination event if the Holliday junction is resolved/repaired successfully, or a conversion event if the junction is not resolved/repaired. The default `convProb` is 0, meaning no conversion event at all. Note that the ratio of conversion to recombination events varies greatly from study to study, ranging from 0.1 to 15 (Chen et al, Nature Review Genetics, 2007). This translate to 0.1/0.9$\tilde{0}$.1 to 15/16$\tilde{0}$.94 of this parameter. When

Note that

- conversion tract length is usually short, and is estimated to be between 337 and 456 bp, with overall range between maybe 50 - 2500 bp.
- simuPOP does not impose a unit for marker distance so your choice of `convParam` needs to be consistent with your unit. In the HapMap dataset, cM is usually assumed and marker distances are around 10kb (0.001cM ≈ 1kb). Gene conversion can largely be ignored. This is important when you use distance based conversion mode such as `CONVERT_TrackLength` or `CONVERT_ExponentialDistribution`.
- After a track length is determined, if a second recombination event happens within this region, the track length will be shortened. Note that conversion is identical to double recombination under this context.

intensity: Intensity of recombination. The actual recombination rate between two loci is determined by `intensity*locus distance (between them)`.

rate: Recombination rate regardless of locus distance after all `afterLoci`. It can also be an array of recombination rates. Should have the same length as `afterLoci` or `totNumOfLoci()`. The recombination rates are independent of locus distance.

afterLoci: An array of locus indexes. Recombination will occur after these loci. If `rate` is also specified, they should have the same length. Default to all loci (but meaningless for those loci located at the end of a chromosome). If this parameter is given, it should be ordered, and can not include loci at the end of a chromosome.

haplodiploid: If set to true, the first copy of paternal chromosomes is copied directly as the paternal chromosomes of the offspring. This is because haplodiploid male has only one set of chromosome.

Note There is no recombination between sex chromosomes of male individuals if `sexChrom()=True`. This may change later if the exchanges of genes between pseudoautosomal regions of `XY` need to be modeled.

clone()

Deep copy of a recombinator

convCount(*size*)

Return the count of conversion of a certain size (only valid in standard modules)

convCounts()

Return the count of conversions of all sizes (only valid in standard modules)

initialize(*pop*)

Initialize a base genotype operator for a population. This function should be called before any other functions are used to transmit genotype.

recCount (*idx*)
Return recombination counts (only valid in standard modules)

recCounts ()
Return recombination counts (only valid in standard modules)

transmitGenotype (*parent, offspring, ploidy*)
FIXME: No document

2.8 Selection

2.8.1 Class `selector`

A base selection operator for all selectors. Genetic selection is tricky to simulate since there are many different *fitness* values and many different ways to apply selection. simuPOP employs an *'ability-to-mate'* approach. Namely, the probability that an individual will be chosen for mating is proportional to its fitness value. More specifically,

- `PreMating` selectors assign fitness values to each individual, and mark part or all subpopulations as under selection.
- during sexless mating (e.g. `binomialSelection` mating scheme), individuals are chosen at probabilities that are proportional to their fitness values. If there are N individuals with fitness values $f_i, i = 1, ..., N$, individual i will have probability $\frac{f_i}{\sum_j f_j}$ to be chosen and passed to the next generation.
- during `randomMating`, males and females are separated. They are chosen from their respective groups in the same manner as `binomialSelection` and mate.

All of the selection operators, when applied, will set an information field `fitness` (configurable) and then mark part or all subpopulations as under selection. (You can use different selectors to simulate various selection intensities for different subpopulations). Then, a *'selector-aware'* mating scheme can select individuals according to their `fitness` information fields. This implies that

- only mating schemes can actually select individuals.
- a selector has to be a `PreMating` operator. This is not a problem when you use the operator form of the selector since its default stage is `PreMating`. However, if you use the function form of the selector in a `pyOperator`, make sure to set the stage of `pyOperator` to `PreMating`.

Note:

You can not apply two selectors to the same subpopulation, because only one fitness value is allowed for each individual.

class selector (*stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["fitness"]*)
Create a selector

apply (*pop*)
Set fitness to all individuals. No selection will happen!

clone ()
Deep copy of a selector

2.8.2 Class `mapSelector` (Function `MapSelector`)

Selection according to the genotype at one or more loci This map selector implements selection according to genotype at one or more loci. A user provided dictionary (map) of genotypes will be used in this selector to set each individual's fitness value.

class `mapSelector` (*loci, fitness, phase=False, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["fitness"]*)

Create a map selector

locus: The locus index. A shortcut to `loci=[locus]`

loci: The locus indexes. The genotypes at these loci will be used to determine the fitness value.

fitness: A dictionary of fitness values. The genotype must be in the form of `'a-b'` for a single locus, and `'a-b|c-d|e-f'` for multi-loci. In the haploid case, the genotype should be specified in the form of `'a'` for single locus, and `'a|b|c'` for multi-locus models.

phase: If `True`, genotypes `a-b` and `b-a` will have different fitness values. Default to `False`.

output: And other parameters please refer to help (`baseOperator.__init__`)

`clone`()

Deep copy of a map selector

2.8.3 Class `maSelector` (Function `MaSelect`)

Multiple allele selector (selection according to wildtype or diseased alleles) This is called 'multiple-allele' selector. It separates alleles into two groups: wildtype and diseased alleles. Wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased alleles.

This selector accepts an array of fitness values:

- For single-locus, `fitness` is the fitness for genotypes AA, Aa, aa, while A stands for wildtype alleles.
- For a two-locus model, `fitness` is the fitness for genotypes AABB, AABb, AAbb, AaBB, AbBb, Aabb, aaBB, aaBb and aaBb.
- For a model with more than two loci, use a table of length 3^n in a order similar to the two-locus model.

class `maSelector` (*loci, fitness, wildtype=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["fitness"]*)

Create a multiple allele selector Please refer to `baseOperator` for other parameter descriptions.

fitness: For the single locus case, `fitness` is an array of fitness of AA, Aa, aa. A is the wildtype group. In the case of multiple loci, fitness should be in the order of AABB, AABb, AAbb, AaBB, AaBb, Aabb, aaBB, aaBb, aabb.

wildtype: An array of alleles in the wildtype group. Any other alleles are considered to be diseased alleles. Default to `[0]`.

output: And other parameters please refer to help (`baseOperator.__init__`)

Note

- `maSelector` only works for diploid populations.
- `wildtype` alleles at all loci are the same.

`clone`()

Deep copy of a `maSelector`

`indFitness`(*ind, gen*)

Calculate/return the fitness value, currently assuming diploid

2.8.4 Class `mlSelector` (Function `MlSelect`)

Selection according to genotypes at multiple loci in a multiplicative model This selector is a 'multiple-locus model' selector. The selector takes a vector of selectors (can not be another `mlSelector`) and evaluate the fitness of an individual as the product or sum of individual fitness values. The mode is determined by parameter `mode`, which takes one of the following values

- `Multiplicative`: the fitness is calculated as $f = \prod_i f_i$, where f_i is the single-locus fitness value.
- `Additive`: the fitness is calculated as $f = \max(0, 1 - \sum_i (1 - f_i))$. f will be set to `0` when $f < 0$.

class `mlSelector`(*selectors, mode=Multiplicative, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["fitness"]*)

Create a multiple-locus selector Please refer to `mapSelector` for other parameter descriptions.

selectors: A list of selectors

`clone`()

Deep copy of a `mlSelector`

2.8.5 Class `pySelector` (Function `PySelect`)

Selection using user provided function This selector assigns fitness values by calling a user provided function. It accepts a list of loci and a Python function `func`. For each individual, this operator will pass the genotypes at these loci, generation number, and optionally values at some information fields to this function. The return value is treated as the fitness value. The genotypes are arranged in the order of `0-0`, `0-1`, `1-0`, `1-1` etc. where X-Y represents locus X - ploidy Y. More specifically, `func` can be

- `func(geno, gen)` if `infoFields` has length 0 or 1.
- `func(geno, gen, fields)` when `infoFields` has more than 1 fields. Values of fields 1, 2, ... will be passed. Both `geno` and `fields` should be a list.

class `pySelector`(*loci, func, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["fitness"]*)

Create a Python hybrid selector

loci: Susceptibility loci. The genotype at these loci will be passed to `func`.

func: A Python function that accepts genotypes at specified loci, generation number, and optionally information fields. It returns the fitness value.

output: And other parameters please refer to help (`baseOperator.__init__`)

infoFields: If specified, the first field should be the information field to save calculated fitness value (should be 'fitness' in most cases). The values of the rest of the information fields (if available) will also be passed to the user defined penetrance function.

`clone`()

Deep copy of a `pySelector`

2.9 Penetrance

2.9.1 Class `basePenetrance`

Base class of all penetrance operators. Penetrance is the probability that one will have the disease when he has certain genotype(s). An individual will be randomly marked as affected/unaffected according to his/her penetrance value. For example, an individual will have probability 0.8 to be affected if the penetrance is 0.8.

Penetrance can be applied at any stage (default to `DuringMating`). When a penetrance operator is applied, it calculates the penetrance value of each offspring and assigns affected status accordingly. Penetrance can also be used `PreMating` or `PostMating`. In these cases, the affected status will be set to all individuals according to their penetrance values.

Penetrance values are usually not saved. If you would like to know the penetrance value, you need to

- use `addInfoField('penetrance')` to the population to analyze. (Or use `infoFields` parameter of the population constructor), and
- use e.g., `mlPenetrance(...., infoFields=['penetrance'])` to add the penetrance field to the penetrance operator you use. You may choose a name other than `'penetrance'` as long as the field names for the operator and population match.

Penetrance functions can be applied to the current, all, or certain number of ancestral generations. This is controlled by the `ancestralGen` parameter, which is default to `-1` (all available ancestral generations). You can set it to `0` if you only need affection status for the current generation, or specify a number `n` for the number of ancestral generations (n + 1 total generations) to process. Note that the `ancestralGen` parameter is ignored if the penetrance operator is used as a during mating operator.

class `basePenetrance`(*ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create a penetrance operator

ancestralGen: If this parameter is set to be `0`, apply penetrance to the current generation; if `-1`, apply to all generations; otherwise, apply to the specified numbers of ancestral generations.

stage: Specify the stage this operator will be applied. Default to `DuringMating`.

infoFields: If one field is specified, it will be used to store penetrance values.

`apply`(*pop*)

Set penetrance to all individuals and record penetrance if requested

`clone`()

Deep copy of a penetrance operator

2.9.2 Class `mapPenetrance` (Function `MapPenetrance`)

Penetrance according to the genotype at one locus Assign penetrance using a table with keys 'X-Y' where X and Y are allele numbers.

class `mapPenetrance`(*loci, penetrance, phase=False, ancGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create a map penetrance operator

locus: The locus index. Shortcut to `loci=[locus]`

loci: The locus indexes. The genotypes of these loci will be used to determine penetrance.

penet: A dictionary of penetrance. The genotype must be in the form of 'a-b' for a single locus.

phase: If `True`, `a/b` and `b/a` will have different penetrance values. Default to `False`.

output: And other parameters please refer to help(baseOperator.__init__)

clone()
Deep copy of a map penetrance operator

2.9.3 Class maPenetrance (Function MaPenetrance)

Multiple allele penetrance operator This is called 'multiple-allele' penetrance. It separates alleles into two groups: wildtype and diseased alleles. Wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased alleles. `maPenetrance` accepts an array of penetrance for AA, Aa, aa in the single-locus case, and a longer table for the multi-locus case. Penetrance is then set for any given genotype.

class maPenetrance(*loci, penetrance, wildtype=[], ancGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a multiple allele penetrance operator (penetrance according to diseased or wildtype alleles)

locus: The locus index. The genotype of this locus will be used to determine penetrance.

loci: The locus indexes. The genotypes of these loci will be examed.

penet: An array of penetrance values of AA, Aa, aa. A is the wild type group. In the case of multiple loci, penetrance should be in the order of AABB, AABb, AAbb, AaBB, AaBb, Aabb, aaBB, aaBb, aabb.

wildtype: An array of alleles in the wildtype group. Any other alleles will be considered as in the diseased allele group.

output: And other parameters please refer to help(baseOperator.__init__)

clone()
Deep copy of a multi-allele penetrance operator

2.9.4 Class mlPenetrance (Function MlPenetrance)

Penetrance according to the genotype according to a multiple loci multiplicative model This is the 'multiple-locus' penetrnace calculator. It accepts a list of penetrances and combine them according to the `mode` parameter, which takes one of the following values:

- `PEN_Multiplicative`: the penetrance is calculated as $f = \prod f_i$.
- `PEN_Additive`: the penetrance is calculated as $f = \min(1, \sum f_i)$. f will be set to `1` when $f < 0$. In this case, s_i are added, not f_i directly.
- `PEN_Heterogeneity`: the penetrance is calculated as $f = 1 - \prod (1 - f_i)$.

Please refer to Neil Risch (1990) for detailed information about these models.

class mlPenetrance(*peneOps, mode=Multiplicative, ancGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a multiple locus penetrance operator

peneOps: A list of penetrance operators

mode: Can be one of `PEN_Multiplicative`, `PEN_Additive`, and `PEN_Heterogeneity`

clone()
Deep copy of a multi-loci penetrance operator

2.9.5 Class `pyPenetrance` (Function `PyPenetrance`)

Assign penetrance values by calling a user provided function For each individual, the penetrance is determined by a user-defined penetrance function `func`. This function takes genetypes at specified loci, and optionally values of specified information fields. The return value is considered as the penetrance for this individual.

More specifically, `func` can be

- `func(geno)` if `infoFields` has length 0 or 1.
- `func(geno, fields)` when `infoFields` has more than 1 fields. Both parameters should be an list.

class `pyPenetrance` (*loci, func, ancGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Provide locus and penetrance for 11, 12, 13 (in the form of dictionary)

loci: The genotypes at these loci will be passed to the provided Python function in the form of `loc1_1, loc1_2, loc2_1, loc2_2, ...` if the individuals are diploid.

func: A user-defined Python function that accepts an array of genotypes at specified loci and return a penetrance value. The return value should be between `0` and `1`.

infoFields: If specified, the first field should be the information field to save calculated penetrance value. The values of the rest of the information fields (if available) will also be passed to the user defined penetrance function.

output: And other parameters please refer to help (`baseOperator.__init__`)

`clone`()

Deep copy of a Python penetrance operator

2.10 Quantitative Trait

2.10.1 Class `quanTrait`

Base class of quantitative trait Quantitative trait is the measure of certain phenotype for given genotype. Quantitative trait is similar to penetrance in that the consequence of penetrance is binary: affected or unaffected; while it is continuous for quantitative trait.

In simuPOP, different operators or functions were implemented to calculate quantitative traits for each individual and store the values in the information fields specified by the user (default to `qtrait`). The quantitative trait operators also accept the `ancestralGen` parameter to control the number of generations for which the `qtrait` information field will be set.

class `quanTrait` (*ancGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["qtrait"]*)

Create a quantitative trait operator

`apply`(*pop*)

Set `qtrait` to all individual

`clone`()

Deep copy of a quantitative trait operator

2.10.2 Class `mapQuanTrait` (Function `MapQuanTrait`)

Quantitative trait according to genotype at one locus Assign quantitative trait using a table with keys 'X-Y' where X

and Y are allele numbers. If parameter `sigma` is not zero, the return value is the sum of the trait plus $N\left(0,\sigma^2\right)$. This random part is usually considered as the environmental factor of the trait.

class mapQuanTrait (*loci, qtrait, sigma=0, phase=False, ancGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["qtrait"]*)

Create a map quantitative trait operator

locus: The locus index. The quantitative trait is determined by genotype at this locus.

loci: An array of locus indexes. The quantitative trait is determined by genotypes at these loci.

qtrait: A dictionary of quantitative traits. The genotype must be in the form of 'a-b'. This is the mean of the quantitative trait. The actual trait value will be $N\left(mean,\sigma^2\right)$. For multiple loci, the form is 'a-b|c-d|e-f' etc.

sigma: Standard deviation of the environmental factor $N\left(0,\sigma^2\right)$.

phase: If `True`, a/b and b/a will have different quantitative trait values. Default to `False`.

output: And other parameters please refer to help (`baseOperator.__init__`)

clone()

Deep copy of a map quantitative trait operator

2.10.3 Class `maQuanTrait` (Function `MaQuanTrait`)

Multiple allele quantitative trait (quantitative trait according to disease or wildtype alleles) This is called 'multiple-allele' quantitative trait. It separates alleles into two groups: wildtype and diseased alleles. Wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased alleles. `maQuanTrait` accepts an array of fitness. Quantitative trait is then set for any given genotype. A standard normal distribution $N\left(0,\sigma^2\right)$ will be added to the returned trait value.

class maQuanTrait (*loci, qtrait, wildtype, sigma=[], ancGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["qtrait"]*)

Create a multiple allele quantitative trait operator Please refer to `quanTrait` for other parameter descriptions.

qtrait: An array of quantitative traits of AA, Aa, aa. A is the wildtype group

sigma: An array of standard deviations for each of the trait genotype (AA, Aa, aa)

wildtype: An array of alleles in the wildtype group. Any other alleles will be considered as diseased alleles. Default to `[0]`.

output: And other parameters please refer to help(`baseOperator.__init__`)

clone()

Deep copy of a multiple allele quantitative trait

2.10.4 Class `mlQuanTrait` (Function `MlQuanTrait`)

Quantitative trait according to genotypes from a multiple loci multiplicative model Operator `mlQuanTrait` is a 'multiple-locus' quantitative trait calculator. It accepts a list of quantitative traits and combine them according to the `mode` parameter, which takes one of the following values

- `Multiplicative:` the mean of the quantitative trait is calculated as $f=\prod f_i$.
- `Additive:` the mean of the quantitative trait is calculated as $f=\sum f_i$.

Note that all σ_i (for f_i) and σ (for f) will be considered. I.e, the trait value should be

$$f=\sum_i\left(f_i+N\left(0,\sigma_i^2\right)\right)+\sigma^2$$

for `Additive` case. If this is not desired, you can set some of the σ to zero.

class mlQuanTrait(*qtraits, mode=Multiplicative, sigma=0, ancGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["qtrait"]*)

Create a multiple locus quantitative trait operator Please refer to `quanTrait` for other parameter descriptions.

qtraits: A list of quantitative traits

mode: Can be one of `Multiplicative` and `Additive`

clone()

Deep copy of a multiple loci quantitative trait operator

qtrait(*ind*)

Currently assuming diploid

2.10.5 Class pyQuanTrait (Function PyQuanTrait)

Quantitative trait using a user provided function For each individual, a user provided function is used to calculate quantitative trait.

class pyQuanTrait(*loci, func, ancGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=["qtrait"]*)

Create a Python quantitative trait operator Please refer to `quanTrait` for other parameter descriptions.

loci: The genotypes at these loci will be passed to `func`.

func: A Python function that accepts genotypes at specified loci and returns the quantitative trait value.

output: And other parameters please refer to help(`baseOperator.__init__`)

clone()

Deep copy of a Python quantitative trait operator

2.11 Statistics Calculation

2.11.1 Class stat (Function Stat)

Calculate statistics Operator `stat` calculates various basic statistics for the population and sets variables in the local namespace. Other operators or functions can refer to the results from the namespace after `stat` is applied. `Stat` is the function form of the operator.

Note that these statistics are dependent to each other. For example, heterotype and allele frequencies of related loci will be automatically calculated if linkage diseqilibrium is requested.

class stat(*popSize=False, numOfMale=False, numOfMale_param={}, numOfAffected=False, numOfAffected_param={}, numOfAlleles=[], numOfAlleles_param={}, alleleFreq=[], alleleFreq_param={}, heteroFreq=[], expHetero=[], expHetero_param={}, homoFreq=[], genoFreq=[], genoFreq_param={}, haploFreq=[], LD=[], LD_param={}, association=[], association_param={}, Fst=[], Fst_param={}, relGroups=[], relLoci=[], rel_param={}, relBySubPop=False, relMethod=[], relMinScored=10, hasPhase=False, midValues=False, output="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Create an `stat` operator

popSize: Whether or not calculate population and virtual subpopulation sizes. This parameter will set the following variables:

- `numSubPop` the number of subpopulations.

- `subPopSize` an array of subpopulation sizes.
- `virtualSubPopSize` (optional) an array of virtual subpopulation sizes. If a subpopulation does not have any virtual subpopulation, the subpopulation size is returned.
- `popSize`, `subPop[sp]['popSize']` the population/subpopulation size.

numOfMale: Whether or not count the numbers or proportions of males and females. This parameter can set the following variables by user's specification:

- `numOfMale`, `subPop[sp]['numOfMale']` the number of males in the population/subpopulation.
- `numOfFemale`, `subPop[sp]['numOfFemale']` the number of females in the population/subpopulation.
- `propOfMale`, `subPop[sp]['propOfMale']` the proportion of males in the population/subpopulation.
- `propOfFemale`, `subPop[sp]['propOfFemale']` the proportion of females in the population/-subpopulation.

numOfMale_param: A dictionary of parameters of `numOfMale` statistics. Can be one or more items choosen from the following options: `numOfMale`, `propOfMale`, `numOfFemale`, and `propOfFemale`.

numOfAffected: Whether or not count the numbers or proportions of affected and unaffected individuals. This parameter can set the following variables by user's specification:

- `numOfAffected`, `subPop[sp]['numOfAffected']` the number of affected individuals in the population/subpopulation.
- `numOfUnaffected`, `subPop[sp]['numOfUnAffected']` the number of unaffected individuals in the population/subpopulation.
- `propOfAffected`, `subPop[sp]['propOfAffected']` the proportion of affected individuals in the population/subpopulation.
- `propOfUnaffected`, `subPop[sp]['propOfUnAffected']` the proportion of unaffected individuals in the population/subpopulation.

numOfAffected_param: A dictionary of parameters of `numOfAffected` statistics. Can be one or more items choosen from the following options: `numOfAffected`, `propOfAffected`, `numOfUnaffected`, `propOfUnaffected`.

numOfAlleles: An array of loci at which the numbers of distinct alleles will be counted (`numOfAlleles=[loc1, loc2, ...]` where `loc1` etc. are absolute locus indexes). This is done through the calculation of allele frequencies. Therefore, allele frequencies will also be calculated if this statistics is requested. This parameter will set the following variables (`carray` objects of the numbers of alleles for *all loci*). Unrequested loci will have `0` distinct alleles.

- `numOfAlleles`, `subPop[sp]['numOfAlleles']` the number of distinct alleles at each locus. (Calculated only at requested loci.)

numOfAlleles_param: A dictionary of parameters of `numOfAlleles` statistics. Can be one or more items choosen from the following options: `numOfAffected`, `propOfAffected`, `numOfUnaffected`, `propOfUnaffected`.

alleleFreq: An array of loci at which all allele frequencies will be calculated (`alleleFreq=[loc1, loc2, ...]` where `loc1` etc. are loci where allele frequencies will be calculated). This parameter will set the following variables (`carray` objects); for example, `alleleNum[1][2]` will be the number of allele `2` at locus `1`:

- `alleleNum[a]`, `subPop[sp]['alleleNum'][a]`
- `alleleFreq[a]`, `subPop[sp]['alleleFreq'][a]`.

alleleFreq_param: A dictionary of parameters of `alleleFreq` statistics. Can be one or more items choosen from the following options: `numOfAlleles`, `alleleNum`, and `alleleFreq`.

genoFreq: An array of loci at which all genotype frequencies will be calculated (`genoFreq=[loc1, loc2, ...]`. You may use parameter `genoFreq_param` to control if `a/b` and `b/a` are the same genotype. This parameter will set the following dictionary variables. Note that unlike list used for `alleleFreq` etc., the indexes `a`, `b` of `genoFreq[loc][a][b]` are dictionary keys, so you will get a *KeyError* when you used a wrong key. You can get around this problem by using expressions like `genoNum[loc].setDefault(a,{})`.

- `genoNum[loc][allele1][allele2]` and `subPop[sp]['genoNum'][loc][allele1][allele2]`, the number of genotype `allele1-allele2` at locus `loc`.
- `genoFreq[loc][allele1][allele2]` and `subPop[sp]['genoFreq'][loc][allele1][allele2]`, the frequency of genotype `allele1-allele2` at locus `loc`.
- genoFreq_param a dictionary of parameters of `phase` = 0 or 1.

heteroFreq: An array of loci at which observed heterozygosities will be calculated (`heteroFreq=[loc1, loc2, ...]`). For each locus, the number and frequency of allele specific and overall heterozygotes will be calcuated and stored in four population variables. For example, `heteroNum[loc][1]` stores number of heterozygotes at locus `loc`, with respect to allele `1`, which is the number of all genotype `1x` or `x1` where does not equal to `1`. All other genotypes such as `02` are considered as homozygotes when `heteroFreq[loc][1]` is calculated. The overall number of heterozygotes (`HeteroNum[loc]`) is the number of genotype `xy` if `x` does not equal to `y`.

- `HeteroNum[loc]`, `subPop[sp]['HeteroNum'][loc]`, the overall heterozygote count.
- `HeteroFreq[loc]`, `subPop[sp]['HeteroFreq'][loc]`, the overall heterozygote frequency.
- `heteroNum[loc][allele]`, `subPop[sp]['heteroNum'][loc][allele]`, allele-specific heterozygote counts.
- `heteroFreq[loc][allele]`, `subPop[sp]['heteroFreq'][loc][allele]`, allele-specific heterozygote frequency.

homoFreq: An array of loci to calculate observed homozygosities and expected homozygosities (`homoFreq=[loc1, loc2, ...]`). This parameter will calculate the numbers and frequencies of homozygotes **xx** and set the following variables:

- `homoNum[loc]`, `subPop[sp]['homoNum'][loc]`.
- `homoFreq[loc]`, `subPop[sp]['homoFreq'][loc]`.

expHetero: An array of loci at which the expected heterozygosities will be calculated (`expHetero=[loc1, loc2, ...]`). The expected heterozygosity is calculated by

$$h_{exp} = 1 - p_i^2,$$

where p_i is the allele frequency of allele i. The following variables will be set:

- `expHetero[loc]`, `subPop[sp]['expHetero'][loc]`.

expHetero_param: A dictionary of parameters of `expHetero` statistics. Can be one or more items choosen from the following options: `subpop` and `midValues`.

haploFreq: A matrix of haplotypes (allele sequences on different loci) to count. For example, `haploFreq = [ [ 0,1,2 ], [1,2] ]` will count all haplotypes on loci 0, 1 and 2; and all haplotypes on loci 1, 2. If only one haplotype is specified, the outer `[]` can be omitted. I.e., `haploFreq=[0,1]` is acceptable. The following dictionary variables will be set with keys `0-1-2` etc. For example, `haploNum['1-2']['5-6']` is the number of allele pair 5, 6 (on loci 1 and 2 respectively) in the population.

- `haploNum[haplo]` and `subPop[sp]['haploNum'][haplo]`, the number of allele sequencies on loci `haplo`.
- `haploFreq[haplo]`, `subPop[sp]['haploFreq'][haplo]`, the frequency of allele sequencies on loci `haplo`.

LD: Calculate linkage disequilibria LD, LD' and r^2, given `LD=[ [loc1, loc2], [ loc1, loc2, allele1, allele2], ... ]`. For each item `[loc1, loc2, allele1, allele2]`, D, D' and r^2 will be calculated based on `allele1` at `loc1` and `allele2` at `loc2`. If only two loci are given, the LD values are averaged over all allele pairs. For example, for allele A at locus `1` and allele B at locus `2`,

$$D = P_{AB} - P_A P_B$$

$$D' = D/D_{max}$$

$$D_{max} = \min\left(P_A\left(1-P_B\right), \left(1-P_A\right)P_B\right) \text{ if } D > 0 \quad \min\left(P_A P_B, \left(1-P_A\right)\left(1-P_B\right)\right) \text{ if } D < 0$$

$$r^2 = \frac{D^2}{P_A\left(1-P_A\right)P_B\left(1-P_B\right)}$$

If only one item is specified, the outer `[]` can be ignored. I.e., `LD=[loc1, loc2]` is acceptable. This parameter will set the following variables. Please note that the difference between the data structures used for `ld` and `LD`.

- `ld['loc1-loc2']['allele1-allele2']`, `subPop[sp]['ld']['loc1-loc2']['allele1-allele2']`
- `ld_prime['loc1-loc2']['allele1-allele2']`, `subPop[sp]['ld_prime']['loc1-loc2']['allel`
- `r2['loc1-loc2']['allele1-allele2']`, `subPop[sp]['r2']['loc1-loc2']['allele1-allele2']`
- `LD[loc1][loc2]`, `subPop[sp]['LD'][loc1][loc2]`.
- `LD_prime[loc1][loc2]`, `subPop[sp]['LD_prime'][loc1][loc2]`.
- `R2[loc1][loc2]`, `subPop[sp]['R2'][loc1][loc2]`.

LD_param: A dictionary of parameters of `LD` statistics. Can have key `stat` which is a list of statistics to calculate. Default to all. If any statistics is specified, only those specified will be calculated. For example, you may use `LD_param={LD_prime}` to calculate D' only, where `LD_prime` is a shortcut for `'stat':['LD_prime']`. Other parameters that you may use are:

- `subPop` whether or not calculate statistics for subpopulations.
- `midValues` whether or not keep intermediate results.

association: Association measures

association_param: A dictionary of parameters of `association` statistics. Can be one or more items choosen from the following options: `ChiSq`, `ChiSq_P`, `UC_U`, and `CramerV`.

Fst: Calculate F_{st}, F_{is}, F_{it}. For example, `Fst = [0,1,2]` will calculate F_{st}, F_{is}, F_{it} based on alleles at loci `0`, `1`, `2`. The locus-specific values will be used to calculate `AvgFst`, which is an average value over all alleles (Weir & Cockerham, 1984). Terms and values that match Weir & Cockerham are:

- F (F_{IT}) the correlation of genes within individuals (inbreeding);
- θ (F_{ST}) the correlation of genes of difference individuals in the same population (will evaluate for each subpopulation and the whole population)
- f (F_{IS}) the correlation of genes within individuals within populations.

This parameter will set the following variables:

- `Fst[loc]`, `Fis[loc]`, `Fit[loc]`
- `AvgFst`, `AvgFis`, `AvgFit`.

Fst_param: A dictionary of parameters of `Fst` statistics. Can be one or more items choosen from the following options: `Fst`, `Fis`, `Fit`, `AvgFst`, `AvgFis`, and `AvgFit`.

relMethod: Method used to calculate relatedness. Can be either `REL_Queller` or `REL_Lynch`. The relatedness values between two individuals, or two groups of individuals are calculated according to Queller & Goodnight (1989) (`method=REL_Queller`) and Lynch et al. (1999) (`method=REL_Lynch`). The results are pairwise relatedness values, in the form of a matrix. Original group or subpopulation numbers are discarded. There is no subpopulation level relatedness value.

relGroups: Calculate pairwise relatedness between groups. Can be in the form of either `[[1,2,3],[5,6,7],[8,9]]` or `[2,3,4]`. The first one specifies groups of individuals, while the second specifies subpopulations. By default, relatedness between subpopulations is calculated.

relLoci: Loci on which relatedness values are calculated

rel_param: A dictionary of parameters of relatedness statistics. Can be one or more items choosen from the following options: `Fst`, `Fis`, `Fit`, `AvgFst`, `AvgFis`, and `AvgFit`.

hasPhase: If a/b and b/a are the same genotype. Default to `False`.

midValues: Whether or not post intermediate results. Default to `False`. For example, `Fst` will need to calculate allele frequencise. If `midValues` is set to `True`, allele frequencies will be posted as well. This will be helpful in debugging and sometimes in deriving statistics.

apply(*pop*)
Apply the `stat` operator

clone()
Deep copy of a `stat` operator

2.12 Tagging (used for pedigree tracking)

2.12.1 Class `tagger`

Base class of tagging individuals This is a during-mating operator that tags individuals with various information. Potential usages are:

- recording the parental information to track pedigree;
- tagging an individual/allele and monitoring its spread in the population etc.

class tagger(*output="", stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a `tagger`, default to be always active but no output

clone()
Deep copy of a
tagger

2.12.2 Class `inheritTagger`

Inherite tag from parents This during-mating operator will copy the tag (information field) from his/her parents. Depending on `mode` parameter, this tagger will obtain tag, value of the first specified information fields, from his/her father or mother (two tag fields), or both (first tag field from father, and second tag field from mother).

An example may be tagging one or a few parents and examining, at the last generation, how many offspring they have.

class inheritTagger(*mode=TAG_Paternal, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], output="", infoFields=["paternal_tag", "maternal_tag"]*)

Create an `inheritTagger` that inherits a tag from one or both parents

mode: Can be one of `TAG_Paternal`, `TAG_Maternal`, and `TAG_Both`

clone()

Deep copy of a `inheritTagger`

2.12.3 Class `parentTagger`

Tagging according to parental indexes This during-mating operator set `tag()` each individual with indexes of his/her parent in the parental population. Because only one parent is recorded, this is recommended to be used for mating schemes that requires only one parent (such as selfMating).

This tagger record indexes to information field parent_idx, and/or a given file. The usage is similar to parentsTagger.

class parentTagger(*begin=0, end=-1, step=1, at=[], rep=[], subPops=[], output="", infoFields=["parent_idx"]*)

Create a `parentTagger`

apply(*pop*)

At the end of a generation, write population structure information to a file with a newline.

clone()

Deep copy of a `parentTagger`

2.12.4 Class `parentsTagger`

Tagging according to parents' indexes This during-mating operator set `tag()`, currently a pair of numbers, of each individual with indexes of his/her parents in the parental population. This information will be used by pedigree-related operators like `affectedSibpairSample` to track the pedigree information. Because parental population will be discarded or stored after mating, these index will not be affected by post-mating operators.

This tagger record parental index to one or both

- one or two information fields. Default to father_idx and mother_idx. If only one parent is passed in a mating scheme (such as selfing), only the first information field is used. If two parents are passed, the first information field records paternal index, and the second records maternal index.
- a file. Indexes will be written to this file. This tagger will also act as a post-mating operator to add a new-line to this file.

class parentsTagger(*begin=0, end=-1, step=1, at=[], rep=[], subPops=[], output="", infoFields=["father_idx", "mother_idx"]*)

Create a `parentsTagger`

apply(*pop*)

At the end of a generation, write population structure information to a file with a newline.

clone()

Deep copy of a `parentsTagger`

2.12.5 Class `pedigreeTagger`

Pedigree tagger is used to save a complete pedigree to a pedigree file during an evolution process. Because is destroyedof record individuals involved in an evolutioary process. This is a simple post-mating tagger that write given

information fields to a file (or standard output).

class pedigreeTagger(*begin=0, end=-1, step=1, at=[], rep=[], subPops=[], stage=PostMating, output=">", pedigreeFields=[]*)

FIXME: No document

2.12.6 Class pyTagger

Python tagger. This tagger takes some information fields from both parents, pass to a Python function and set the individual field with the return value. This operator can be used to trace the inheritance of trait values.

class pyTagger(*func=None, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], output="", infoFields=[]*)

Creates a `pyTagger` that works on specified information fields

infoFields: Information fields. The user should gurantee the existence of these fields.

func: A Pyton function that returns a list to assign the information fields. e.g., if `fields=['A', 'B']`, the function will pass values of fields `'A'` and `'B'` of father, followed by mother if there is one, to this function. The return value is assigned to fields `'A'` and `'B'` of the offspring. The return value has to be a list even if only one field is given.

clone()

Deep copy of a `pyTagger`

2.13 Terminator

2.13.1 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="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], 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, preceeded with an optional *message* will be written to it.

apply(*pop*)

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

clone()

Deep copy of a `terminateIf` terminator

2.14 The Python operator

2.14.1 Class pyOperator

A python operator that directly operate a population. This operator accepts a function that can take the form of

- `func(pop)` when `stage=PreMating` or `PostMating`, without setting `param`;
- `func(pop, param)` when `stage=PreMating` or `PostMating`, with `param`;
- `func(pop, off, dad, mom)` when `stage=DuringMating` and `passOffspringOnly=False`, without setting `param`;
- `func(off)` when `stage=DuringMating` and `passOffspringOnly=True`, and without setting `param`;
- `func(pop, off, dad, mom, param)` when `stage=DuringMating` and `passOffspringOnly=False`, with `param`;
- `func(off, param)` when `stage=DuringMating` and `passOffspringOnly=True`, with `param`.

For `Pre-` and `PostMating` usages, a population and an optional parameter is passed to the given function. For `DuringMating` usages, population, offspring, its parents and an optional parameter are passed to the given function. Arbitrary operations can be applied to the population and offspring (if `stage=DuringMating`).

class `pyOperator` (*func, param=None, stage=PostMating, formOffGenotype=False, offspringOnly=False, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)

Python operator, using a function that accepts a population object.

func: A Python function. Its form is determined by other parameters.

param: Any Python object that will be passed to `func` after `pop` parameter. Multiple parameters can be passed as a tuple.

formOffGenotype: This option tells the mating scheme this operator will set the genotype of offspring (valid only for `stage=DuringMating`). By default (`formOffGenotype=False`), a mating scheme will set the genotype of offspring before it is passed to the given Python function. Otherwise, a 'blank' offspring will be passed.

passOffspringOnly: If `True`, `pyOperator` will expect a function of form `func(off [,param])`, instead of `func(pop, off, dad, mom [, param])` which is used when `passOffspringOnly` is `False`. Because many during-mating `pyOperator` only need access to offspring, this will improve efficiency. Default to `False`.

Note

- Output to `output` is not supported. That is to say, you have to open/close/append to files explicitly in the Python function. Because files specified by `output` are controlled (opened/closed) by simulators, they should not be manipulated in a `pyOperator` operator.
- This operator can be applied `Pre-`, `During-` or `Post- Mating` and is applied `PostMating` by default. For example, if you would like to examine the fitness values set by a selector, a `PreMating` Python operator should be used.

`apply` (*pop*)

Apply the `pyOperator` operator to one population

2.15 Miscellaneous

2.15.1 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=">", stage=PostMating, begin=0, end=0, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a noneOp.

apply (*pop*)
Apply the noneOp operator to one population

2.15.2 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, ancGen=0, width=1, max=100, loci=[], output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], 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 *ancGen*). 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*.

apply (*pop*)
Apply a dumper operator to population *pop*.

clone ()
Deep copy of a dumper operator.

2.15.3 Class savePopulation

An operator that save populations to specified files.

class savePopulation (*output="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], 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*.

apply (*pop*)
Apply operator to population *pop*.

clone ()
Deep copy of a savePopulation operator.

2.15.4 Class setAncestralDepth

This operator sets the number of ancestral generations to keep during the evolution of a population. This is usually used to start storing ancestral generations at the end of an evolutionary process. A typical usage is setAncestralDepth(1, at=-1) which will cause the parental generation of the present population to be stored at the last generation of an evolutionary process.

class setAncestralDepth (*depth, output=">", stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a setAncestralDepth operator that sets the ancestral depth of an population. It basically calls the

`population.setAncestralDepth` member function of a population.

apply(*pop*)
Apply the `setAncestralDepth` operator to population *pop*.

2.15.5 Class `ifElse`

This operator accepts an expression that will be evaluated when this operator is applied. An if-operator will be applied when the expression returns `True`. Otherwise an else-operator will be applied.

class ifElse(*cond, ifOp=None, elseOp=None, output=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a conditional operator that will apply operator *ifOp* if condition *cond* is met and *elseOp* otherwise. The replicate and generation applicability parameters (*begin*, *end*, *step*, *at* and *rep*) of the *ifOp* and *elseOp* are ignored because their applicability is determined by the `ifElse` operator.

apply(*pop*)
Apply the `ifElse` operator to population *pop*.

2.15.6 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=">", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], 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*.

apply(*pop*)
Apply the `pause` operator to one population

2.15.7 Class `turnOnDebug` (Function `TurnOnDebug`)

Turn on debug. There are several ways to turn on debug information for non-optimized modules, namely

- set environment variable `SIMUDEBUG`.
- use `simuOpt.setOptions(debug)` function.
- use function `TurnOnDebug`
- use the `turnOnDebug` operator

The advantage of using an operator is that you can turn on debug at given generations.

class turnOnDebug(*code, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a `turnOnDebug` operator that turns on debug information *code* when it is applied to a population.

2.15.8 Class turnOffDebug (Function TurnOffDebug)

Turn off certain debug information. Please refer to operator `turnOnDebug` for detailed usages.

class turnOffDebug(*code, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a `turnOffDebug` operator that turns off debug information *code* when it is applied to a population.

2.15.9 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.

class ticToc(*output=">", stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], subPops=[], infoFields=[]*)
Create a `ticToc` operator that outputs the elapsed since the last time it was applied, and the overall time since it was created.

Chapter 3

Python Modules (under revision)

3.1 Module `simuPOP`

3.1.1 Function form of operators

`Dump`(*pop, *args, **kwargs*)
Apply operator `dumper` to population *pop*.

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

`InitByFreq`(*pop, *args, **kwargs*)
Apply operator `initByFreq` to population *pop*.

`InitByValue`(*pop, *args, **kwargs*)
Apply operator `initByValue` to population *pop*.

`TurnOnDebug`(*code*)
Set debug code *code*. Name of available codes are available from `DebugCodes`.

`TurnOffDebug`(*code=DBG_ALL*)
Turn off debug code *code*. Default to turn off all debug codes.

`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*.

> **Note**
>
> Unlike its operator counterpart, this function returns the result of e expr rather than writting it to an output.

`PyExec`(*pop, *args, **kwargs*)
Execute e stmts in population e pop's local namespace

`Migrate`(*pop, *args, **kwargs*)
Function form of operator migrator.

3.1.2 Offspring generators

cloneOffspringGenerator(*ops=[], *args, **kwargs*)
An offspring generator that uses `cloneGenoTransmitter()` as a default genotype transmitter. Additional during mating operators can be specified using the *ops* parameter. Other parameters are passed directly to `offspringGenerator`.

mendelianOffspringGenerator(*ops=[], *args, **kwargs*)
An offspring generator that uses `mendelianGenoTransmitter()` as a default genotype transmitter. Additional during mating operators can be specified using the *ops* parameter. Other parameters are passed directly to `offspringGenerator`.

haplodiploidOffspringGenerator(*ops=[], *args, **kwargs*)
An offspring generator that uses `haplodiploidGenoTransmitter()` as a default genotype transmitter. Additional during mating operators can be specified using the *ops* parameter. Other parameters are passed directly to `offspringGenerator`.

selfingOffspringGenerator(*ops=[], *args, **kwargs*)
An offspring generator that uses `selfingGenoTransmitter()` as a default genotype transmitter. Additional during mating operators can be specified using the *ops* parameter. Other parameters are passed directly to `offspringGenerator`.

3.1.3 Mating schemes

cloneMating(*numOffspring=1, sexMode=None, ops=[], subPopSize=[], subPop=(), weight=0, selectionField=None*)
A homogeneous mating scheme that uses a sequential parent chooser and a clone offspring generator. Please refer to class `offspringGenerator` for parameters *ops* and *numOffspring*, and to class `homoMating` for parameters *subPopSize*, *subPop* and *weight*. Parameters *sexMode* and *selectionField* are ignored because this mating scheme does not support natural selection, and `cloneOffspringGenerator` copies sex from parents to offspring.

randomSelection(*numOffspring=1, sexMode=None, ops=[], subPopSize=[], subPop=(), weight=0, selectionField='fitness'*)
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. Please refer to class `randomParentChooser` for parameter *selectionField*, to class `offspringGenerator` for parameters *ops* and *numOffspring*, and to class `homoMating` for parameters *subPopSize*, *subPop* and *weight*. Parameter *sexMode* is ignored because `cloneOffspringGenerator` copies sex from parents to offspring.

randomMating(*numOffspring=1, sexMode=31, ops=[], subPopSize=[], subPop=(), weight=0, selectionField='fitness'*)
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. Please refer to class `randomParentsChooser` for parameter *selectionField*, to class `offspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `homoMating` for parameters *subPopSize*, *subPop* and *weight*.

monogamousMating(*numOffspring=1, sexMode=31, ops=[], subPopSize=[], subPop=(), weight=0, selectionField=None*)
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. Please refer to class `offspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `homoMating` for parameters *subPopSize*, *subPop* and *weight*. Parameter *selectionField* is ignored because this mating scheme does not support natural selection.

polygamousMating(*polySex=1, polyNum=1, numOffspring=1, sexMode=31, ops=[], subPopSize=[], subPop=(), weight=0, selectionField='fitness'*)

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. 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*, *subPop* and *weight*.

alphaMating(*alphaSex=1, alphaNum=0, alphaField='', numOffspring=1, sexMode=31, ops=[], subPopSize=[], subPop=(), weight=0, selectionField='fitness'*)

A homogeneous mating scheme that uses a alpha-individual parents chooser and a Mendelian offspring generator. It differs from the basic random mating scheme in that selection of parents of sex *alphaSex* is limited to certain alpha individuals, which are chosen either randomly (parameter *alphaNum*) or from an information field (parameter *alphaField*). This mating scheme is usually used to simulate animal population where only a few alpha individuals have the right to mate. Please refer to class `alphaParentsChooser` for parameters *alphaSex*, *alphaNum*, *alphaField* and *selectionField*, to class `offspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `homoMating` for parameters *subPopSize*, *subPop* and *weight*.

haplodiploidMating(*numOffspring=1.0, sexMode=31, ops=[], subPopSize=[], subPop=(), weight=0, selectionField='fitness'*)

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. Please refer to class `randomParentsChooser` for parameter *selectionField*, to class `offspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `homoMating` for parameters *subPopSize*, *subPop* and *weight*.

selfMating(*replacement=True, numOffspring=1, sexMode=31, ops=[], subPopSize=[], subPop=(), weight=0, selectionField='fitness'*)

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. 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*, *subPop* and *weight*.

consanguineousMating(*infoFields=[], func=None, param=None, replacement=False, numOffspring=1.0, sexMode=31, ops=[], subPopSize=[], subPop=(), weight=0, selectionField='fitness'*)

A homogeneous mating scheme that uses an information parents chooser and a Mendelian offspring generator. A function *func* should be defined to locate certain types of relative to each individual and save their indexes to information fields *infoFields*. This mating scheme will then choose a parent randomly and then another parent from his/her relatives using their saved indexes. Please refer to class `infoParentsChooser` for parameters *infoFields*, *func*, *param* and *selectionField*, to class `offspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `homoMating` for parameters *subPopSize*, *subPop* and *weight*.

controlledRandomMating(*loci=[], alleles=[], freqFunc=None, numOffspring=1, sexMode=31, ops=[], subPopSize=[], subPop=(), weight=0, selectionField='fitness'*)

A homogeneous mating scheme that uses a random sexual parents chooser with replacement and a controlled offspring generator using Mendelian genotype transmitter. At each generation, function *freqFunc* will be 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. 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*, *subPop* and *weight*.

3.1.4 Ascertainment operators

3.1.5 Class `randomSample`

This operator draws random individuals from a population repeatedly and forms a number of random samples. These samples can be put in the population's local namespace, or save to disk files. The function form of this operator returns a list of samples directly.

class `randomSample` (*size, *args, **kwargs*)
Draw *size* random samples from a population *times* times. *size* can be a number or a list of numbers. In the former case, individuals are drawn from the whole population and the samples has only one subpopulation. In the latter case, a given number of individuals are drawn from each subpopulation and the result sample has the same number of subpopulation as the population from which samples are drawn. The samples are saved in the population's local namespace if *name* or *nameExpr* is given, and are saved as diskfiles if *saveAs* or *saveAsExpr* is given.

`RandomSample` (*pop, *args, **kwargs*)
Function version of operator `randomSample`.

3.1.6 Class `caseControlSample`

This operator chooses random cases and controls from a population repeatedly. These samples can be put in the population's local namespace, or save to disk files. The function form of this operator returns a list of samples directly.

class `caseControlSample` (*cases, controls, *args, **kwargs*)
Draw *cases* affected and *controls* unaffected individuals from a population repeatedly. *cases* can be a number or a list of numbers. In the former case, affected individuals are drawn from the whole population. In the latter case, a given number of individuals are drawn from each subpopulation. The same hold for *controls*. The resulting samples have two subpopulations that hold cases and controls respectively. The samples are saved in the population's local namespace if *name* or *nameExpr* is given, and are saved as diskfiles if *saveAs* or *saveAsExpr* is given.

`CaseControlSample` (*pop, *args, **kwargs*)
Function version of operator caseControlSample whose __init__function is

Draw *cases* affected and *controls* unaffected individuals from a population repeatedly. *cases* can be a number or a list of numbers. In the former case, affected individuals are drawn from the whole population. In the latter case, a given number of individuals are drawn from each subpopulation. The same hold for *controls*. The resulting samples have two subpopulations that hold cases and controls respectively. The samples are saved in the population's local namespace if *name* or *nameExpr* is given, and are saved as diskfiles if *saveAs* or *saveAsExpr* is given.

3.1.7 Class `affectedSibpairSample`

This operator chooses affected sibpairs and their parents from a population repeatedly. These samples can be put in the population's local namespace, or save to disk files. The function form of this operator returns a list of samples directly.

The population to be sampled needs to have at least one ancestral generation. In addition, parents of each offspring is needed so information fields, most likely *father_idx* and *mother_idx* should be used to track parents in the parental generation. An during mating operator *parentsTagger* is designed for such a purpose. In addition, because it is very unlikely for two random offspring to share parents, affected sibpairs can only be ascertained from populations that are generated using a mating scheme that produes more than one offspring at each mating event.

class `affectedSibpairSample` (*size, infoFields=['father_idx', 'mother_idx'], *args, **kwargs*)
Draw *size* families, including two affected siblings and their parents from a population repeatedly. The population to be sampled must have at least one ancestral generation. It should also have two information fields specified by parameter *infoFields* (Default to `['father_idx', 'mother_idx']`. Parameter *size* can be

a number or a list of numbers. In the former case, affected sibpairs are drawn from the whole population. In the latter case, a given number of affected sibpairs are drawn from each subpopulation. In both cases, affected sibpairs in the resulting sample form their own subpopulations (of size two). The samples are saved in the population's local namespace if *name* or *nameExpr* is given, and are saved as diskfiles if *saveAs* or *saveAsExpr* is given.

AffectedSibpairSample(*pop, size, *args, **kwargs*)
Function version of operator affectedSibpairSample whose __init__function is

Draw *size* families, including two affected siblings and their parents from a population repeatedly. The population to be sampled must have at least one ancestral generation. It should also have two information fields specified by parameter *infoFields* (Default to `['father_idx', 'mother_idx']`. Parameter *size* can be a number or a list of numbers. In the former case, affected sibpairs are drawn from the whole population. In the latter case, a given number of affected sibpairs are drawn from each subpopulation. In both cases, affected sibpairs in the resulting sample form their own subpopulations (of size two). The samples are saved in the population's local namespace if *name* or *nameExpr* is given, and are saved as diskfiles if *saveAs* or *saveAsExpr* is given.

3.1.8 Miscellaneous functions

AlleleType()
Return the allele type of the current module. Can be `binary`, `short`, or `long`.

AvailableRNGs()
List the names of all available random number generators

Limits()
Print out system limits

DebugCodes()
Return names of all debug codes

LoadPopulation(*file*)
Load a population from a file.

LoadSimulator(*file, matingScheme*)
Load a simulator from a file with the specified mating scheme. The file format is by default determined by file extension (`format="auto"`). Otherwise, `format` can be one of `txt`, `bin`, or `xml`.

MaxAllele()
Return the maximum allowed allele state of the current simuPOP module, which is `1` for binary modules, `255` for short modules and `65535` for long modules.

ModuleCompiler()
Return the compiler used to compile this simuPOP module

ModuleDate()
Return the date when this simuPOP module is compiled

ModulePlatForm()
Return the platform on which this simuPOP module is compiled

ModulePyVersion()
Return the Python version this simuPOP module is compiled for

Optimized()
Return `True` if this simuPOP module is optimized

SetRNG(*rng="", seed=0*)
Set random number generator. If `seed=0` (default), a random seed will be given. If `rng=""`, seed will be set to the current random number generator.

rng()
Return the currently used random number generator

simuRev()
Return the revision number of this simuPOP module. Can be used to test if a feature is available.

simuVer()
Return the version of this simuPOP module

3.2 Module simuOpt

Module simuOpt can be used to control which simuPOP module to load, and how it is loaded using function setOptions . It also provides a simple way to set simulation options, from user input, command line, configuration file or a parameter dialog. All you need to do is to define an option description list that lists all parameters in a given format, and call the getParam function.

This module, if loaded, pre-process the command line options. More specifically, it checks command line option:

-c configfile: read from a configuration file

–config configfile: the same as -c

–optimized: load optimized modules, unless setOption explicitly use non-optimized modules.

-q: Do not display banner information when simuPOP is loaded

–quiet: the same as -q

–useTkinter: force the use of Tcl/Tk dialog even when wxPython is available. By default, wxPython is used whenever possible.

–noDialog: do not use option dialog. If the options can not be obtained from command line or configuraiton file, users will be asked to input them interactively.

Because these options are reserved, you can not use them in your simuPOP script.
printConfig(*opt, param, out=<open file '<stdout>', mode 'w' at 0x2b2eb8248198>*)
Print configuration.

opt: option description list

param: parameters returned from getParam()

out: output
valueNot(*t*)
Return a function that returns true if passed option does not passes validator t

valueOr(*t1, t2*)
Return a function that returns true if passed option passes validator t1 or t2

valueAnd(*t1, t2*)
Return a function that returns true if passed option passes validator t1 and t2

valueOneOf(*t*)
Return a function that returns true if passed option is one of the values list in t

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

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

valueGT(*a*)
Return a function that returns true if passed option is greater than a

valueGE(*a*)
Return a function that returns true if passed option is greater than or equal to a

valueLT(*a*)
Return a function that returns true if passed option is less than a

valueLE(*a*)
Return a function that returns true if passed option is less than or equal to a

valueEqual(*a*)
Return a function that returns true if passed option equals a

valueNotEqual(*a*)
Return a function that returns true if passed option does not equal a

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

valueIsList()
Return a function that returns true if passed option is a list (or tuple)

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

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

setOptions(*optimized=None, mpi=None, chromMap=[], alleleType=None, quiet=None, debug=[]*)
set options before simuPOP is loaded to control which simuPOP module to load, and how the module should be loaded.

optimized: whether or not load optimized version of a module. If not set,: environmental variable SIMUOPTIMIZED, and commandline option --optimized will be used if available. If nothing is defined, standard version will be used.

mpi: obsolete.

chromMap: obsolete.

alleleType: 'binary', 'short', or 'long'. 'standard' can be used as 'short': for backward compatibility. If not set, environmental variable SIMUALLELETYPE will be used if available. if it is not defined, the short allele version will be used.

quiet: If True, supress banner information when simuPOP is loaded.

debug: a list of debug code (as string). If not set, environmental variable: SIMUDEBUG will be used if available.

requireRevision(*rev*)
Compare the revision of this simuPOP module with given revision. Raise an exception if current module is out of date.

3.3 Module `simuUtil`

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

3.3.1 Class `simuProgress`

This class defines a very simple text based progress bar. It will display a character (default to "`.`") for each change of progress (default to 2%), and a number (1, 2, ..., 9) for each 10% of progress, and print a message (default to "`Done.\n`") when the job is finished.

This class is used as follows:

```
progress = simuProgress("Start simulation", 500)
for i in range(500):
    progress.update(i+1)
# if you would like to make sure the done message is displayed.
progress.done()
```

class simuProgress (*message, totalCount, progressChar='.', block=2, done=' Done.\n'*)

totalCount: Total expected steps.

progressChar: Character to be displayed for each progress.

block: display progress at which interval (in terms of percentage)?

done: Message displayed when the job is finished.

done ()
Finish progressbar, print 'done' message.

update (*count*)
Update the progreebar.

3.4 Module simuRPy

This module helps the use of "rpy" package with simuPOP. It defines an operator "varPlotter" that can be used to plot population expressions when "rpy" is installed.

3.4.1 Class varPlotter

This class defines a Python operator that uses R to plot a simuPOP express. During the evolution, this express is evaluated in each replicate's local namespace. How this expression is plotted depends on the dimension of the return value (if a sequence is returned), number of replicates, whether or not historical values (collected over several generations) are plotted, and plot type (lines or images).

The default behavior of this operator is to plot the history of an expression. For example, when operator

varPlotter(var='expr')

is used in simulator::evolve, the value of `expr` will be recorded each time when this operator is applied. A line will be draw in a figure with x-axis being the generation number. Parameters `ylim` can be used to specify the range of y-axis.

If the return value of expression `expr` is a sequence (tuple or list), parameter `varDim` has to be used to indicate the dimension of this expression. For example,

varPlotter(var='expr', varDim=3)

will plot three lines, corresponding to the histories of each item in the array.

If the expression returns a number and there are several replicates, parameter `numRep` ' should be used. In this case, each line will correspond to a replicate.

If the expression returns a vector and there are several replicates, several subplots will be used. Parameter `byRep` or `byVar` should be used to tell `varPlotter` whether the subplots should be divided by replicate or by variable. For example,

varPlotter(var='expr', varDim=8, numRep=5, byRep=1)

will use an appropriate layout for your subplots, which is, in this case, 2x3 for 5 replicates. Each subplot will have 8 lines. If byVal is `True`, there will be 3x3 subplots for 8 items in an array, and each subplot will have 5 lines. Note that `byRep` or `byVal` can also be used when there is only one replicate or if the dimension of the expression is one.

When `history=False`, histories of each variable will be discarded so the figure will always plot the current value of the expression.

class varPlotter(*expr, history=True, varDim=1, numRep=1, win=0, ylim=[0, 0], update=1, title='', xlab='generation', ylab='', axes=True, lty=[], col=[], mfrow=[1, 1], separate=False, byRep=False, byVal=False, plotType='plot', level=20, saveAs='', leaveOpen=True, dev='', width=0, height=0, *args, **kwargs*)

expr: expression that will be evaluate at each replicate's local namespace when the operator is applied.

history: whether or not record and plot the history of an expression. Default to True.

varDim: If the return value of `expr` is a sequence, `varDim` should be set to the length of this sequence. Default to 1.

numRep: Number of replicates of the simulator. Default to 1.

win: Window of generations. I.e., how many generations to keep in a figure. This is useful when you want to keep track of only recent changes of an expression. The default value is 0, which will keep all histories.

ylim: The range of y-axis.

update: Update figure after update generations. This is used when you do not want to update the figure every time when this operator is applied.

title, xlab, ylab: Title, label at x and y axes of your figure(s). xtitle is defaulted to 'generation'.

axes: Whether or not plot axes. Default to `True`.

lty: A list of line type for each line in the figure.

col: A list of colors for each line in the figure.

level: level of image colors (default to 20).

saveAs: save figures in files saveAs.ext. If ext (such as pdf, jpeg) is given, a corresponding device will be used. Otherwise, a postscript driver will be used. Generation number will be inserted before file extension.

separate: plot data lines in separate panels.

image: use R image function to plot image, instead of lines.

leaveOpen: whether or not leave the plot open when plotting is done. Default to True.

3.5 Utility Classes

3.5.1 Class RNG

Random number generator This random number generator class wraps around a number of random number generators from GNU Scientific Library. You can obtain and change system random number generator through the `rng()` function. Or create a separate random number generator and use it in your script.

class RNG(*rng=None, seed=0*)

RNG used by simuPOP.

max()

Maximum value of this RNG.

maxSeed()

Return the maximum allowed seed value

name()
Return RNG name

pvalChiSq(*chisq, df*)
Right hand side (single side) p-value for ChiSq value

randBinomial(*n, p*)
Binomial distribution B(n, p).

randBit()
FIXME: No document

randExponential(*v*)
FIXME: No document

randGeometric(*p*)
Geometric distribution.

randGet()
Return a random number in the range of [0, 2, ... max()-1]

randInt(*n*)
Return a random number in the range of [0, 1, 2, ... n-1]

randMultinomial(*N, p, n*)
Multinomial distribution.

randMultinomialVal(*N, p*)
FIXME: No document

randNormal(*m, v*)
Normal distribution.

randPoisson(*p*)
Poisson distribution.

randUniform01()
Uniform distribution [0,1).

seed()
Return the seed of this RNG

setRNG(*rng=None, seed=0*)
Choose an random number generator, or set seed to the current RNG

rng: Name of the RNG. If rng is not given, environmental variable GSL_RNG_TYPE will be used if it is available. Otherwise, RNG`mt19937` will be used.

seed: Random seed. If not given, `/dev/urandom`, `/dev/random`, system time will be used, depending on availability, in that order. Note that windows system does not have `/dev` so system time is used.

setSeed(*seed*)
If seed is 0, method described in `setRNG` is used.