

---

# simuPOP Reference Manual

*Release 0.8.6 (Rev: 1688 )*

Bo Peng

December 2004

Last modified  
July 26, 2008

**Department of Epidemiology, U.T. M.D. Anderson Cancer Center**

**Email:** [bpeng@mdanderson.org](mailto:bpeng@mdanderson.org)

**URL:** <http://simupop.sourceforge.net>

**Mailing List:** [simupop-list@lists.sourceforge.net](mailto:simupop-list@lists.sourceforge.net)

## Acknowledgements:

Dr. Marek Kimmel  
Dr. François Balloux  
Dr. William Amos  
SWIG user community  
Python user community  
Keck Center for Computational and Structural Biology  
U.T. M.D. Anderson Cancer Center

© 2004-2007 Bo Peng

---

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies. Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled Copying and GNU General Public License are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one. Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

## 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.

The core of simuPOP is a scripting language (Python) that 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. The full power of simuPOP and Python (even R) can be utilized to simulate arbitrarily complex evolutionary scenarios.

simuPOP is written in C++ and is provided as Python modules. Besides a front-end providing an interactive shell and a scripting language, Python is used extensively to pass dynamic parameters, calculate complex statistics and write operators. Because of the openness of simuPOP and Python, users can make use of external programs, such as R, to perform statistical analysis, gene mapping and visualization. Depending on machine configuration, simuPOP can simulate large (think of millions) populations at reasonable speed.

This is a reference manual to all variables, functions, and objects of simuPOP. To learn different components of simuPOP and how to write simuPOP scripts, please refer to the *simuPOP User's Guide*.

### How to cite simuPOP:

Bo Peng and Marek Kimmel (2005) simuPOP: a forward-time population genetics simulation environment. *bioinformatics*, **21**(18): 3686-3687



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Loading simuPOP	2
1.2	References and the <code>clone()</code> member function	4
1.3	Zero-based indexes, ranges, absolute and relative indexes	4
1.4	Function form of an operator	5
1.5	The <code>carray</code> type	5
1.6	Random Number Generator	7
1.6.1	Random seed	7
1.6.2	Class <code>RNG</code>	7
1.7	Name Conventions	8
1.8	Online resources	9
<b>2</b>	<b>simuPOP Components</b>	<b>11</b>
2.1	Genotypic structure	11
2.1.1	Class <code>GenoStruTrait</code>	12
2.2	Population	13
2.2.1	Class <code>population</code>	14
2.2.2	Ancestral populations	22
2.2.3	Save and Load a Population	23
2.2.4	View a population (GUI, wxPython required)	24
2.3	Virtual subpopulations	24
2.3.1	Class <code>sexSplitter</code>	26
2.3.2	Class <code>affectionSplitter</code>	26
2.3.3	Class <code>infoSplitter</code>	26
2.3.4	Class <code>proportionSplitter</code>	27
2.3.5	Class <code>rangeSplitter</code>	27
2.3.6	Class <code>genotypeSplitter</code>	28
2.3.7	Class <code>combinedSplitter</code>	28
2.4	Individuals	29
2.4.1	Class <code>individual</code>	29
2.5	Mating Scheme	32
2.5.1	Class <code>mating</code>	32
2.5.2	Class <code>noMating</code>	33
2.5.3	Class <code>cloneMating</code>	33
2.5.4	Class <code>binomialSelection</code>	34
2.5.5	Class <code>baseRandomMating</code>	34
2.5.6	Class <code>randomMating</code>	35
2.5.7	Class <code>selfMating</code>	36
2.5.8	Class <code>monogamousMating</code>	36

2.5.9	Class <code>polygamousMating</code>	37
2.5.10	Class <code>consanguineousMating</code>	37
2.5.11	Class <code>alphaMating</code>	38
2.5.12	Class <code>haplodiploidMating</code>	39
2.5.13	Class <code>pedigreeMating</code>	39
2.5.14	Class <code>pyMating</code>	40
2.5.15	Class <code>heteroMating</code>	40
2.5.16	Class <code>sequentialParentChooser</code>	40
2.5.17	Class <code>sequentialParentsChooser</code>	41
2.5.18	Class <code>randomParentChooser</code>	41
2.5.19	Class <code>randomParentsChooser</code>	41
2.5.20	Class <code>infoParentsChooser</code>	42
2.5.21	Class <code>pedigreeParentsChooser</code>	43
2.5.22	Class <code>pyParentsChooser</code>	43
2.5.23	Class <code>cloneOffspringGenerator</code>	44
2.5.24	Class <code>selfingOffspringGenerator</code>	44
2.5.25	Class <code>haplodiploidOffspringGenerator</code>	44
2.5.26	Class <code>mendelianOffspringGenerator</code>	45
2.5.27	Determine the number of offspring during mating	45
2.5.28	Determine offspring sex	46
2.5.29	Determine subpopulation sizes of the next generation	47
2.5.30	Demographic change functions	47
2.5.31	Sex chromosomes	48
2.5.32	Parent choosers and offspring generators	48
2.5.33	Homogeneous and hybrid mating schemes	49
2.5.34	Heterogeneous mating schemes	50
2.6	Operators	52
2.6.1	Class <code>baseOperator</code>	52
2.7	Simulator	54
2.7.1	Class <code>simulator</code>	54
2.7.2	Generation number	56
2.7.3	Operator calling sequence	56
2.7.4	Save and Load	57
2.8	Population variables	57
2.8.1	<code>vars()</code> and <code>dvars()</code> functions	57
2.8.2	Local namespace, <code>pyEval</code> and <code>pyExec</code> operators	58
2.9	Information fields	60
2.10	Pedigree	62
2.10.1	Class <code>pedigree</code>	62
<b>3</b>	<b>Operator References</b>	<b>65</b>
3.1	Python operators	65
3.1.1	Class <code>pyOperator</code>	65
3.1.2	Class <code>pyIndOperator</code>	68
3.2	Initialization	68
3.2.1	Class <code>initializer</code>	68
3.2.2	Class <code>initSex</code> (Function form: <code>InitSex</code> )	69
3.2.3	Class <code>initByFreq</code> (Function form: <code>InitByFreq</code> )	69
3.2.4	Class <code>initByValue</code> (Function form: <code>InitByValue</code> )	70
3.2.5	Class <code>spread</code> (Function form: <code>Spread</code> )	71
3.2.6	Class <code>pyInit</code> (Function form: <code>PyInit</code> )	71
3.3	Migration	72
3.3.1	Class <code>migrator</code>	72
3.3.2	Functions (Python) <code>MigrIslandRates</code> , <code>MigrSteppingStoneRates</code> ( <code>simuUtil.py</code> )	73

3.3.3	Class <code>pyMigrator</code> . . . . .	74
3.3.4	Class <code>splitSubPop</code> (Function form: <code>SplitSubPop</code> ) . . . . .	75
3.3.5	Class <code>mergeSubPops</code> (Function form: <code>MergeSubPops</code> ) . . . . .	75
3.3.6	Class <code>resizeSubPops</code> (Function form: <code>ResizeSubPops</code> ) . . . . .	76
3.4	Mutation . . . . .	76
3.4.1	Class <code>mutator</code> . . . . .	76
3.4.2	Class <code>kamMutator</code> (Function form: <code>KamMutate</code> ) . . . . .	77
3.4.3	Class <code>smmMutator</code> (Function form: <code>SmmMutate</code> ) . . . . .	78
3.4.4	Class <code>gsmMutator</code> (Function form: <code>GsmMutate</code> ) . . . . .	78
3.4.5	Class <code>pyMutator</code> (Function form: <code>PyMutate</code> ) . . . . .	79
3.4.6	Class <code>pointMutator</code> (Function form: <code>PointMutate</code> ) . . . . .	79
3.5	Recombination and gene conversion . . . . .	80
3.5.1	Class <code>recombinator</code> . . . . .	80
3.5.2	Gene conversion . . . . .	82
3.6	Selection . . . . .	83
3.6.1	Mechanism . . . . .	83
3.6.2	Class <code>selector</code> . . . . .	84
3.6.3	Class <code>mapSelector</code> (Function form: <code>MapSelector</code> ) . . . . .	85
3.6.4	Class <code>maSelector</code> (Function form: <code>MaSelect</code> ) . . . . .	85
3.6.5	Class <code>mlSelector</code> (Function form: <code>MLSelect</code> ) . . . . .	86
3.6.6	Class <code>pySelector</code> (Function form: <code>PySelect</code> ) . . . . .	87
3.7	Penetrance . . . . .	87
3.7.1	Class <code>penetrance</code> . . . . .	87
3.7.2	Class <code>mapPenetrance</code> (Function form: <code>MapPenetrance</code> ) . . . . .	88
3.7.3	Class <code>maPenetrance</code> (Function form: <code>MaPenetrance</code> ) . . . . .	89
3.7.4	Class <code>mlPenetrance</code> (Function form: <code>MLPenetrance</code> ) . . . . .	89
3.7.5	Class <code>pyPenetrance</code> (Function form: <code>PyPenetrance</code> ) . . . . .	90
3.8	Quantitative Trait . . . . .	91
3.8.1	Class <code>quanTrait</code> . . . . .	91
3.8.2	Class <code>mapQuanTrait</code> (Function form: <code>MapQuanTrait</code> ) . . . . .	91
3.8.3	Class <code>maQuanTrait</code> (Function form: <code>MaQuanTrait</code> ) . . . . .	92
3.8.4	Class <code>mlQuanTrait</code> (Function form: <code>MLQuanTrait</code> ) . . . . .	92
3.8.5	Class <code>pyQuanTrait</code> (Function form: <code>PyQuanTrait</code> ) . . . . .	93
3.9	Ascertainment . . . . .	94
3.9.1	Class <code>sample</code> . . . . .	94
3.9.2	Class <code>pySubset</code> (Function form: <code>PySubset</code> ) . . . . .	95
3.9.3	Class <code>pySample</code> (Function form: <code>PySample</code> ) . . . . .	95
3.9.4	Class <code>randomSample</code> (Function form: <code>RandomSample</code> ) . . . . .	96
3.9.5	Class <code>caseControlSample</code> (Function form: <code>CaseControlSample</code> ) . . . . .	96
3.9.6	Class <code>affectedSibpairSample</code> (Function form: <code>AffectedSibpairSample</code> ) . . . . .	97
3.9.7	Class <code>largePedigreeSample</code> . . . . .	98
3.9.8	Class <code>nuclearFamilySample</code> . . . . .	98
3.10	Statistics Calculation . . . . .	99
3.10.1	Class <code>stator</code> . . . . .	99
3.10.2	Class <code>stat</code> (Function form: <code>Stat</code> ) . . . . .	99
3.11	Expression and Statements . . . . .	103
3.11.1	Class <code>dumper</code> . . . . .	103
3.11.2	Class <code>savePopulation</code> . . . . .	104
3.11.3	Class <code>pyOutput</code> . . . . .	104
3.11.4	Class <code>pyEval</code> (Function form: <code>PyEval</code> ) . . . . .	105
3.11.5	Class <code>pyExec</code> (Function form: <code>PyExec</code> ) . . . . .	105
3.11.6	Class <code>infoEval</code> (Function form: <code>infoEval</code> ) . . . . .	106
3.11.7	Class <code>infoExec</code> (Function form: <code>infoExec</code> ) . . . . .	107
3.12	Tagging (used for pedigree tracking) . . . . .	107

3.12.1	Class <code>tagger</code>	107
3.12.2	Class <code>inheritTagger</code>	108
3.12.3	Class <code>parentTagger</code>	108
3.12.4	Class <code>parentsTagger</code>	108
3.12.5	Class <code>sexTagger</code>	109
3.12.6	Class <code>affectionTagger</code>	109
3.12.7	Class <code>infoTagger</code>	110
3.12.8	Class <code>pyTagger</code>	110
3.13	Terminator	110
3.13.1	Class <code>terminator</code>	110
3.13.2	Class <code>terminateIf</code>	111
3.13.3	Class <code>continueIf</code>	111
3.14	Conditional operator	112
3.14.1	Class <code>ifElse</code>	112
3.15	Debug-related operators/functions	113
3.15.1	Class <code>turnOnDebug</code> (Function form: <code>TurnOnDebug</code> )	113
3.15.2	Class <code>turnOffDebug</code> (Function form: <code>TurnOffDebug</code> )	114
3.16	Miscellaneous	114
3.16.1	Class <code>noneOp</code>	114
3.16.2	Class <code>pause</code>	115
3.16.3	Class <code>ticToc</code> (Function form: <code>TicToc</code> )	115
3.16.4	Class <code>setAncestralDepth</code>	116
<b>4</b>	<b>Global and Python Utility functions</b>	<b>117</b>
4.1	Global functions	117
4.2	Utility Modules	118
4.2.1	Module <code>simuOpt</code>	118
4.2.2	Module <code>simuUtil</code>	122
4.2.3	Module <code>simuRPy</code>	127
4.2.4	Module <code>hapMapUtil</code>	128
	<b>Index</b>	<b>131</b>



# LIST OF EXAMPLES

1.1	Getting help using the <code>help()</code> function	1
1.2	Use of standard <code>simuPOP</code> modules	2
1.3	Use of optimized <code>simuPOP</code> modules	2
1.4	Use <code>simuOpt</code> to control which <code>simuPOP</code> module to load	3
1.5	Reference to a population of a simulator	4
1.6	Conversion between absolute and relative indexes	5
1.7	Function <code>InitByFreq</code>	5
1.8	Usage of the <code>carray</code> type	6
1.9	Random number generator	7
2.1	Ancestral populations	23
2.2	Save population variables	24
2.3	Use <code>simuViewPop</code> to view a population	24
2.4	Virtual subpopulation related functions	25
2.5	Function <code>population::individual()</code>	29
2.6	Function <code>population::individuals()</code>	29
2.7	A generator function that mimicks random mating	48
2.8	Save and load a simulator	57
2.9	Population variables	58
2.10	Local namespaces of populations	58
2.11	Use of operators <code>pyEval</code> and <code>pyExec</code>	59
2.12	Use regular information field function	61
2.13	Use <code>infoExec</code> and <code>infoEval</code> operators	61
3.1	Define a python operator	66
3.2	Use of python operator	67
3.3	Turn on/off debug information	113



# Introduction

This reference manual assumes that you have read the *simuPOP User's Guide* and know the basic concepts of simuPOP. It is also recommended that you learn some basics of Python before you continue. I have listed a few Python resources at the end of this chapter, along with links to some simuPOP tutorials.

Almost all information contained in this manual can be accessed from command line, after you install and import the simuPOP module. For example, you can use `help(population.addInfoField)` to view the help information of member function `addInfoField` of class `population`.

Example 1.1: Getting help using the `help()` function

```
>>> help(population.addInfoField)
Help on method population_addInfoField in module _simuPOP_la:

population_addInfoField(...) unbound simuPOP_la.population method
    Description:

        add an information field to a population

    Usage:

        x.addInfoField(field, init=0)

    Arguments:

        field:          new information field. If it already exists, it
                        will be re-initialized.
        init:           initial value for the new field.

>>>
```

It is important that you understand that

- The constructor of a class is named `__init__` in python. That is to say, you should use the following command to display the help information of the constructor of class `population`:

```
>>> help(population.__init__)
```

- Some classes are derived from other classes and have access to member functions of their base classes. For example, class `population`, `individual` and `simulator` are all derived from class `GenoStruTrait`. Therefore, you can use all `GenoStruTrait` member functions from these classes.

The constructor of a derived class also calls the constructor of its base class so you may have to refer to the base class for some parameter definitions. For example, parameters `begin`, `end`, `step`, `at` etc are shared by all operators, and are explained in details only in class `baseOperator`.

## 1.1 Loading simuPOP

simuPOP is composed of six modules: standard short, long and binary alleles, each of them have standard and optimized modules. A Message Passing Interface (MPI) version is under development but not yet available. The short modules use 1 byte to store each allele which limits the possible allele states to 256. This is enough most of the times but not so if you need to simulate models such as the infinite allele model. In those cases, you can use the long allele version of the modules, which use 2 bytes for each allele and can have  $2^{16}$  possible allele states. On the other hand, if you would like to simulate a large number of binary (SNP) markers, binary modules can save you a lot of RAM. Depending on applications, binary modules can be faster or slower than other modules.

Standard modules have detailed debug and run-time validation mechanism to make sure the simulations run correctly. Whenever something unusual is detected, simuPOP would terminate with a detailed error message. The cost of such run-time checking varies from application to application but can be very high under some extreme circumstances. Because of this, optimized versions for all modules are provided. They bypass all parameter checking and run-time validations and will simply crash if things go wrong. It is recommended that you use standard modules whenever possible and only use the optimized version when performance is needed and you are confident that your simulation is running as expected.

Example 1.2 and 1.3 demonstrate the differences between standard and optimized modules, by executing two invalid commands. The standard module returns proper error messages, while the optimized module returns erroneous results and even crashes.

### Example 1.2: Use of standard simuPOP modules

```
>>> pop = population(10, loci=[2])
>>> pop.locusPos(10)
Traceback (most recent call last):
  File "refManual.py", line 1, in <module>
    #
IndexError: src/genoStru.h:460 absolute locus index (10) out of range of 0 - 1
>>> pop.individual(20).setAllele(1, 0)
Traceback (most recent call last):
  File "refManual.py", line 1, in <module>
    #
IndexError: src/population.h:463 individual index (20) is out of range of 0 ~ 9
>>>
```

### Example 1.3: Use of optimized simuPOP modules

```
% setenv SIMUOPTIMIZED
% python
>>> from simuPOP import *
simuPOP : Copyright (c) 2004-2006 Bo Peng
Developmental Version (May 21 2007) for Python 2.3.4
[GCC 3.4.6 20060404 (Red Hat 3.4.6-3)]
Random Number Generator is set to mt19937 with random seed 0x2f04b9dc5ca0fc00
This is the optimized short allele version with 256 maximum allelic states.
For more information, please visit http://simupop.sourceforge.net,
or email simupop-list@lists.sourceforge.net (subscription required).
>>> pop = population(10, loci=[2])
>>> pop.locusPos(10)
1.2731974748756028e-313
>>> pop.individual(20).setAllele(1, 0)
Segmentation fault
```

You can control the choice of modules in the following ways:

- Set environment variable `SIMUALLELETYPE` to 'short', 'long' or 'binary', and `SIMUOPTIMIZED` to use the optimized version. The default module is the standard short module.
- Before you load `simuPOP`, set options using `simuOpt.setOptions(optimized, alleleType, quiet, debug)`. `alleleType` can be short, long or binary. `quiet=True` suppresses banner information when `simuPOP` is loaded, and `debug` is a comma-separated list of debug options specified by `ListDebugCode()`. Debug information is only available for standard modules.
- If you are running a `simuPOP` script that conforms to `simuPOP` convention, you should be able to use optimized module using command line option `--optimized`.

After a `simuPOP` module is loaded, you can use the following functions to determine some module and platform dependent information.

- `AlleleType()`: return 'binary', 'short', or 'long'.
- `Optimized()`: return True or False.
- `MaxAllele()`: return 1 for binary modules, usually 255 for short modules and  $2^{16} - 1$  for long modules.
- `simuVer()`: return the version string
- `simuRev()`: `simuPOP` revision number. If your script needs a recent version of `simuPOP`, it is a good idea to test `simuRev()` against the revision when the feature you need becomes available.
- `Limits()`: print the limits of this module on this platform, such as the maximum number of loci in a population.

Example 1.4: Use `simuOpt` to control which `simuPOP` module to load

```
>>> import simuOpt
>>> simuOpt.setOptions(optimized=False, alleleType='long', quiet=True)
>>> from simuPOP import *
>>> # remember that global functions start with captical letters
>>> print AlleleType()
long
>>> print Optimized()
False
>>>
```

After `simuPOP` is installed. It is recommended that you run the test scripts under the `tests` directory. This will make sure that your system is working correctly. To run all tests, run

```
sh run_tests.sh
```

Or, if you do not install RPy and R, run

```
sh run_tests.sh norpy
```

Windows users have to run

```
set SIMUALLELETYPE=short
python run_tests.py
```

repeatedly, with `SIMUALLELETYPE` set to short, long and binary.

## 1.2 References and the `clone()` member function

Assignment in Python only creates a new reference to the existing object. For example,

```
pop = population(...)
pop1 = pop
```

will create a reference `pop1` to population `pop`. Modifying `pop1` will modify `pop` as well. If you would like to have an independent copy, use

```
pop1 = pop.clone()
```

All `simuPOP` classes (objects) have a `clone` function that can be used to create an independent copy of the object. Because cloning a large population can be costly, a few methods are provided to access populations inside a simulator. Assuming that `simu` is a simulator with several populations,

1. `simu.population(rep)` returns a reference to the `rep`'th population. You can, although not recommended, modify simulator through this `pop` reference. Be cautious though, that the following seemingly innocent usage of this function will crash `simuPOP`, because the simulator `simu` will be destroyed after the call to `func()` is ended, leaving `pop` as a reference to an invalid population object.

Example 1.5: Reference to a population of a simulator

```
def func():
    simu = simulator(
        population(10),
        randomMating())
    # evolve simu ..., then return population
    return simu.population(0)

pop = func()
pop.popSize()
```

2. To get an independent copy of a population, you can use `pop = simu.getPopulation(rep)`, which returns an independent copy of population `rep` of `simu`. `simu` is untouched.
3. If the simulator will be destroyed as in Example 1.5,

```
pop = simu.getPopulation(rep, destructive=True)
```

can be used. This function will *extract* population `rep` from the simulator instead of copying it, and bypassing a potentially very costly process.

## 1.3 Zero-based indexes, ranges, absolute and relative indexes

**All arrays in `simuPOP` start at index 0.** This conforms to Python and C++ indexes. To avoid confusion, I will refer the first locus as locus zero, the second locus as locus one; the first individual in a population as individual zero, and so on.

Ranges in `simuPOP` also conforms to Python ranges. That is to say, a range has the form of `[a,b)` where `a` belongs to the range, and `b` does not. For example, `pop.chromBegin(1)` refers to the index of the first locus on chromosome 1 (actually exists), and `pop.chromEnd(1)` is the index of the last locus on chromosome 1 **plus 1**, which might or might not be a valid index. In this way

```
for locus in range(pop.chromBegin(1), pop.chromEnd(1)):
    print locus
```

will iterate through all locus on chromosome 1.

Another two important concepts are the *absolute index* and the *relative index* of a locus. The former index ignores chromosome structure. For example, if there are 5 and 7 loci on the first two chromosomes, the absolute indexes of the two chromosomes are (0, 1, 2, 3, 4), (5, 6, 7, 8, 9, 10, 11) and the relative indexes are (0, 1, 2, 3, 4), (0, 1, 2, 3, 4, 5, 6). Absolute indexes are more frequently used because they avoid the trouble of having to use two numbers (chrom, index) to refer to a locus. Two functions `chromLocusPair(absIndex)` and `absLocusIndex(chrom, index)` are provided to convert between these two kinds of indexes. An individual can also be referred by its *absolute index* and *relative index* where *relative index* is the index in its subpopulation.

Example 1.6: Conversion between absolute and relative indexes

```
>>> pop = population(size=[20, 30], loci=[5, 6])
>>> print pop.chromLocusPair(7)
(1, 2)
>>> print pop.absLocusIndex(1,1)
6
>>> print pop.absIndIndex(10, 1)
30
>>> print pop.subPopIndPair(40)
(1, 20)
>>>
```

## 1.4 Function form of an operator

Operators are usually applied to populations through a simulator. They are created and passed as parameters to the `evolve` function of a simulator. During evolution, the `evolve()` function determines if an operator can be applied to a population and apply it when appropriate. More details about operators will be described in section 2.6.

You can ignore the specialties of an operator and call its `apply()` function directly. For example, you can initialize a population outside a simulator by

```
initByFreq( [.3, .2, .5] ).apply(pop)
```

or dump the content of a population by

```
dumper().apply(pop)
```

This usage is used so often that it deserves some simplification. Equivalent functions are defined for most of the operators. For example, function `InitByFreq` is defined for operator `initByFreq` as follows

Example 1.7: Function `InitByFreq`

```
>>> def InitByFreq(pop, *args, **kwargs):
...     initByFreq(*args, **kwargs).apply(pop)
...
>>> InitByFreq(pop, [.2, .3, .4, .1])
>>>
```

The function form of an operator is listed after its class name in this reference manual.

## 1.5 The `carray` type

The return value of `simuPOP` functions with names starting with `arr` is of a special Python type `carray`. This object reflects the underlying C/C++ array which can be modified through this list-like interface, with the exception

that you can not change the size of the array. Only `count` and `index` list functions can be used, but all comparison, assignment and slice operations are allowed.

#### Example 1.8: Usage of the `carray` type

```
>>> # obtain an object using one of the arrXXX functions
>>> pop = population(loci=[3,4], lociPos=[1,2,3,4,5,6,7])
>>> arr = pop.arrLociPos()
>>> # print and expression (just like list)
>>> print arr
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> str(arr)
'[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]'
>>> # count
>>> arr.count(2)
1
>>> # index
>>> arr.index(2)
1
>>> # can read write
>>> arr[0] = 0.5
>>> # the underlying locus position is also changed
>>> print pop.lociPos()
(0.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
>>> # convert to list
>>> arr.tolist()
[0.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> # or simply
>>> list(arr)
[0.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> # compare to list directly
>>> arr == [0.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
True
>>> # you can also convert and compare
>>> list(arr) == [0.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
True
>>> # slice
>>> arr[:] = [1,2,3,4,5,6,7]
>>> # assign from another part
>>> arr[1:3] = arr[3:5]
>>> # arr1 is 1,2,3
>>> arr1 = arr[:3]
>>> # assign slice from a number
>>> # arr will also be affected since arr1 point to a part of arr
>>> arr1[:] = 10
>>> # assign vector of the same length
>>> len(arr1)
3
>>> arr1[:] = [30,40, 50]
>>>
```

**Important note:** Objects returned from `arrXXX` functions should be considered temporary. There is no guarantee that the underlying array will still be valid after any population operation.



## 1.6 Random Number Generator

There are many random number generators (RNG) with different properties. Using a bad RNG can seriously compromise the validity of simulation results. Although the default RNG `mt19937` has good performance, `simuPOP` allows you to choose from a number of RNGs, all from GNU Scientific Library (GSL). Please refer to the documentation of GSL for more details about these RNGs.

If you need to use a RNG in your `simuPOP` script, you can either use Python `random` module (`import random`), use `rng()` function to get the RNG of `simuPOP`, or create a separate RNG using `RNG(name, seed)` function.

Example 1.9: Random number generator

```
>>> print ListAllRNG()
('gfsr4', 'mt19937', 'mt19937_1999', 'mt19937_1998', 'r250', 'rand', 'rand48', 'random12
>>> # get system RNG
>>> print rng().name()
mt19937
>>> # set system RNG
>>> SetRNG("taus2", seed=10)
>>> print rng().name()
taus2
>>> # create a separate RNG instance
>>> r=RNG()
>>> for n in range(1,5):
...     print r.randBinomial(10, .7),
...
7 9 6 9
>>>
```

### 1.6.1 Random seed

When `simuPOP` is loaded, it creates a random number generator. This RNG gets its seed from system random number generator that guarantees random seed for all instances of `simuPOP` even if they are initialized at the same time.

Random seed can be set manually after `simuPOP` is loaded. This can be used to repeat a simulation with its random seed saved by the function `rng().seed()`.

### 1.6.2 Class RNG

Random number generator

#### Details

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.

#### Initialization

Create a RNG object. You can also use `rng()` function to get the RNG used by `simuPOP`.

```
RNG(rng=None, seed=0)
```

#### Member Functions

**x.max()** Maximum value of this RNG.

**x.maxSeed()** Return the maximum allowed seed value

**x.name()** Return RNG name

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

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

**x.randExponential(v)** `SimuPOP::RNG::randExponential`

**x.randGeometric(p)** Geometric distribution.

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

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

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

**x.randMultinomialVal(N, p)** `SimuPOP::RNG::randMultinomialVal`

**x.randNormal(m, v)** Normal distribution.

**x.randPoisson(p)** Poisson distribution.

**x.randUniform01()** Uniform distribution  $[0,1)$ .

**x.seed()** Return the seed of this RNG

**x.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, `RNGmt19937` 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.

**x.setSeed(seed)** Set random seed for this random number generator if seed is 0, method described in `setRNG` is used.

## 1.7 Name Conventions

`simuPOP` follows the following naming conventions.

- Classes (objects), member functions and parameter names start with small character and use capital character for the first character of each word afterward. For example

```
population, population::subPopSize(), individual::setInfo()
```

- Most standalone (global) functions start with capital character. This is how you can differ an operator from its function version. For example, `initByFreq(vars)` is an operator and `InitByFreq(pop, vars)` is its function version.
- Constants start with capital characters. For example

```
MigrByProportion, StatNumOfFemale
```

- The following words in function names are abbreviated:

```
pos (position), info (information), migr (migration), subPop (subpopulation),
(rep) replicate, gen (generation), grp (group(s)), ops (operators),
expr (expression), stmts (statements)
```

## 1.8 Online resources

There are several excellent Python books and tutorials. If you are new to Python, you can start with

1. The Python tutorial (<http://docs.python.org/tut/tut.html>)
2. Other online tutorials listed at <http://www.python.org/doc/>

The PDF version of this reference manual is distributed with simuPOP. You can also get the latest version of this file from the simuPOP subversion repository. To access it, go to <http://simupop.sourceforge.net>, click SF.net summary > Code > SVN Browse > trunk > doc > refManual.pdf and download the HEAD version. You can also find some tutorials that are not distributed with simuPOP from the subversion repository, such as

1. Forward-time simulations using simuPOP, a tutorial: a tutorial that was given in a simuPOP workshop held at University of Alabama at Birmingham.
2. Forward-time simulations using simuPOP, an in-depth course: a in-depth course about simuPOP components, with a lot of examples.

The filenames are `tutorial.pdf` and `course.pdf`, respectively. Note that these presentations will not be updated so their content can become out of date. This reference manual should be considered as the authoritative resource of simuPOP.



# simuPOP Components

## 2.1 Genotypic structure

Genotypic structure refers to

- ploidy, the number of copies of basic number of chromosomes (c.f. `ploidy()`, `ploidyName()`). A haplodiploid population will return 2 as ploidy number.
- the number of chromosomes (c.f. `numChrom()`)
- the existence of sex chromosome (c.f. `sexChrom()`)
- the number of loci on each chromosome (c.f. `numLoci(ch)`, `totNumLoci()`)
- the locus position on its chromosome (c.f. `locusPos(loc)`, `arrLociPos()`)
- allele name(s), default to allele number (c.f. `alleleName(allele)`, `alleleNames()`)
- the maximum allele state (c.f. `maxAllele()`)
- the names of the information fields (c.f. `infoField(idx)`, `infoFields()`)

*Information fields* refer to float numbers attached to each individual, such as fitness value, parent index, age. They are used to store auxiliary information of individuals, and are essential to the operations of some simuPOP components. For example, 'fitness' field is required by all selectors. Details please refer to section 2.9.

If `sexChrom()` is false, all chromosomes are assumed to be autosomes. You can also create populations with a sex chromosome. Currently, simuPOP only models the XY chromosomes in diploid population. This is to say,

- sex chromosome is always the last chromosome.
- sex chromosome can only be specified for diploid population (`ploidy()=2`).
- sex chromosomes (XY) may differ in length. You should specify the length of the longer one as the chromosome length. If there are more loci on X than Y, the rest of the Y chromosome is unused. Mutation may still occur at this unused part of chromosome to simplify implementation and usage.
- it is assumed that males have XY and females have XX chromosomes. The sex chromosomes of male individuals are in the order of XY.

A population can be haplodiploid (Females with two sets of chromosomes, and males with one set of chromosomes) if you specify `ploidy=Haplodiploid` when a population is created. Such a population actually store two copies of

chromosomes for both male and female individuals. The difference between a diploid and a haplodiploid population is that some operators, such as a recombinator, will recognize a haplodiploid population and act accordingly.

Individuals in the same population share the same genotypic structure. Consequently, *the genotypic information can be accessed from individual, population and simulator levels.*

### 2.1.1 Class `GenoStruTrait`

Genotypic structure related functions, can be accessed from individuals, populations and simulator levels.

#### Details

Genotypic structure refers to the number of chromosomes, the number and position of loci on each chromosome, and allele and locus names etc. All individuals in a population share the same genotypic structure. Because class `GenoStruTrait` is inherited by class `population`, class `individual`, and class `simulator`, functions provided in this class can be accessed at the individual, population and simulator levels. This object can not be created directly. It is created by a population.

#### Initialization

`SimuPOP::GenoStruTrait::GenoStruTrait`

```
GenoStruTrait()
```

#### Member Functions

**x.absLocusIndex(chrom, locus)** Return the absolute index of a locus on a chromosome. c.f. `chromLocusPair`

**x.alleleName(allele)** Return the name of an allele (if previously specified). Default to allele index.

**x.alleleNames()** Return an array of allele names

**x.arrLociPos()** Return a `carray` of loci positions of all loci

**Note:** Modifying loci position directly using this function is strongly discouraged.

**x.arrLociPos(chrom)** Return a `carray` of loci positions on a given chromosome

**Note:** Modifying loci position directly using this function is strongly discouraged.

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

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

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

**x.chromLocusPair(locus)** Return a `(chrom, locus)` pair of an absolute locus index, c.f. `absLocusIndex`

**x.chromName(chrom)** Return the name of an chrom

**x.chromNames()** Return an array of chrom names

**x.distLeft(loc)** Distance left to the right of the loc, till the end of chromosome

**x.genoSize()** Return the total number of loci times ploidy

**x.haplodiploid()** `SimuPOP::GenoStruTrait::haplodiploid`

**x.hasInfoField(name)** Determine if an information field exists

**x.infoField(idx)** Obtain the name of information field `idx`

**x.infoFields()** Return an array of all information fields

**x.infoIdx(name)** Return the index of the field `name`, return `-1` if not found

**x.infoSize()** Obtain the number of information fields

**x.lociByNames(names)** Return an array of locus indexes by locus names

**x.lociCovered(loc, dist)** Starting from `loc`, how many markers are covered by distance `dist` ( $\geq 0$ ) the result will be at least 1, even if `dist = 0`.

**x.lociDist(loc1, loc2)** Distance between loci `loc1` and `loc2`. These two loci should be on the same chromosome. The distance will be negative if `loc1` is after `loc2`.

**x.lociLeft(loc)** Return the number of loci left on that chromosome, including locus `loc`

**x.lociNames()** Return names of all loci

**x.lociPos()** Return loci positions

**x.locusByName(name)** Return the index of a locus by its locus name

**x.locusName(loc)** Return the name of a locus

**x.locusPos(locus)** Return the position of a locus

**x.maxAllele()** Return the maximum allele value for all loci. Default to maximum allowed allele state.  
Maximum allele value has to be 1 for binary modules. `maxAllele` is the maximum possible allele value, which allows `maxAllele+1` alleles 0, 1, ..., `maxAllele`.

**x.numChrom()** Return the number of chromosomes

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

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

**x.ploidy()** Return ploidy, the number of homologous sets of chromosomes

**x.ploidyName()** Return ploidy name, `haploid`, `diploid`, or `triploid` etc.

**x.sexChrom()** Determine whether or not the last chromosome is sex chromosome

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

## 2.2 Population

`population` objects are essential to `simuPOP`. They are composed of subpopulations each with certain number of individuals having the same genotypic structure. Class `population` has a large number of member functions, ranging from reviewing simple properties to generating a new population from the current one. Fortunately, you do not have to know all the member functions to use a population unless you need to write pure Python functions/operators that involves complicated manipulation of populations.

`simuPOP` supports subpopulations with boundary, and virtual subpopulations within subpopulations. Mating is within subpopulations only. Exchanges of genetic information across subpopulations can only be done through migration. Population and subpopulation sizes can be changed, as a result of mating or migration. More specifically,

- migration can change subpopulation size; create or remove subpopulations. Since migration can not generate new individuals, the total population size will not be changed.

- mating can fill any population/subpopulation structure with offspring. Both population and subpopulation sizes can be changed. Since mating is within subpopulations, you can not create a new subpopulation through mating.
- a special operator `pySubset` can shrink the population size. It removes individuals according to their `subPopID()` status. (Will explain later.) This can be used to model a sudden population decrease due to some natural disaster.
- subpopulations can be split or merged.

Note that migration will most likely change the subpopulation sizes. To keep the subpopulation sizes constant, you can set the subpopulation sizes during mating so that the next generation will have desired subpopulation sizes.

## 2.2.1 Class `population`

A collection of individuals with the same genotypic structure.

### Details

#### Details

- **subpopulations.** A population is divided into subpopulations (unstructured population has a single subpopulation, which is the whole population itself). Subpopulation structure limits the usually random exchange of genotypes between individuals by disallowing mating between individuals from different subpopulations. In the presence of subpopulation structure, exchange of genetic information across subpopulations can only be done through migration. Note that `simuPOP` uses one-level population structure, which means there is no sub-subpopulation or family in subpopulations.
- **variables.** Every population has its own variable space, or *local namespace* in `simuPOP` term. This namespace is a Python dictionary that is attached to each population and can be exposed to the users through `vars()` or `dvars()` function. Many functions and operators work and store their results in this namespace. For example, function `Stat` sets variables such as `alleleFreq[loc]`, and you can access it via `pop.dvars().alleleFreq[loc][allele]`.
- **ancestral generations.** A population can save arbitrary number of ancestral generations. During evolution, the latest several (or all) ancestral generations are saved. Functions to switch between ancestral generations are provided so that one can examine and modify ancestral generations.

### Initialization

Create a population object with given size and genotypic structure.

```
population(size=[], ploidy=2, loci=[], sexChrom=False, lociPos=[],
ancestralDepth=0, chromNames=[], alleleNames=[], lociNames=[],
maxAllele=ModuleMaxAllele, infoFields=[])
```

**alleleNames** An array of allele names. For example, for a locus with alleles A, C, T, G, you can specify `alleleNames` as `('A', 'C', 'T', 'G')`.

**ancestralDepth** Number of most recent ancestral generations to keep during evolution. Default to 0, which means only the current generation will be available. You can set it to a positive number `m` to keep the latest `m` generations in the population, or `-1` to keep all ancestral populations. Note that keeping track of all ancestral generations may quickly exhaust your computer RAM. If you really need to do that, using `savePopulation` operator to save each generation to a file is a much better choice.

**chromNames** An array of chromosome names.



**infoFields** Names of information fields that will be attached to each individual. For example, if you need to record the parents of each individual using operator `parentTagger()`, you will need two fields `father_idx` and `mother_idx`.

**loci** An array of numbers of loci on each chromosome. The length of parameter `loci` determines the number of chromosomes. Default to `[1]`, meaning one chromosome with a single locus.

The last chromosome can be sex chromosome. In this case, the maximum number of loci on X and Y should be provided. I.e., if there are 3 loci on Y chromosome and 5 on X chromosome, use 5.

**lociNames** An array or a matrix (separated by chromosomes) of names for each locus. Default to `"locX-Y"` where X is the chromosome index and Y is the locus number, both starting from 1.

**lociPos** A 1-d or 2-d array specifying positions of loci on each chromosome. You can use a nested array to specify loci position for each chromosome. For example, you can use `lociPos=[1,2,3]` when `loci=[3]` or `lociPos=[[1,2],[1.5,3,5]]` for `loci=[2,3]`. `simuPOP` does not assume a unit for these positions, although they are usually interpreted as centiMorgans. The default values are 1, 2, etc. on each chromosome.

**maxAllele** Maximum allele number. Default to the maximum allowed allele state of the current library. This will set a cap for all loci. For individual locus, you can specify `maxAllele` in mutation models, which can be smaller than the global `maxAllele` but not larger. Note that this number is the number of allele states minus 1 since allele number starts from 0.

**ploidy** Number of sets of homologous copies of chromosomes. Default to 2 (diploid). Please use `Haplodiploid` to specify a haplodiploid population. Note that the ploidy number returned for such a population will be 2 and male individuals will store two copies of chromosomes. Operators such as a recombinator will recognize this population as haplodiploid and act accordingly.

**sexChrom** Diploid population only. If this parameter is `True`, the last homologous chromosome will be treated as sex chromosome. (XY for male and XX for female.) If X and Y have different numbers of loci, the number of loci of the longer one of the last (sex) chromosome should be specified in `loci`.

**size** An array of subpopulation sizes. If a single number is given, it will be the size of a single subpopulation of the whole population.

**subPop** Obsolete parameter

## Member Functions

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

**index** index of an individual in a subpopulation `subPop`

**subPop** subpopulation index (start from 0)

**x.activateVirtualSubPop(subPop, virtualSubPop=InvalidSubPopID, type=vspSplitter::Visible)**  
Activate a virtual subpopulation.

**id** subpopulation id

**vid** virtual subpopulation id

**Note:** this function is currently not recommended to be used.

**x.addInfoField(field, init=0)** Add an information field to a population

**field** new information field. If it already exists, it will be re-initialized.

**init** initial value for the new field.

**x.addInfoFields(fields, init=0)** Add one or more information fields to a population

**fields** an array of new information fields. If one or more of the fields already exist, they will be re-initialized.

**init** initial value for the new fields.

**x.ancestor(ind, gen)** Reference to an individual `ind` in an ancestral generation

This function gives access to individuals in an ancestral generation. It will refer to the correct generation even if the current generation is not the latest one. That is to say, `ancestor(ind, 0)` is not always `individual(ind)`.

**x.ancestor(ind, subPop, gen)** Reference to an individual `ind` in a specified subpopulation or an ancestral generation

This function gives access to individuals in an ancestral generation. It will refer to the correct generation even if the current generation is not the latest one. That is to say, `ancestor(ind, 0)` is not always `individual(ind)`.

**x.ancestralDepth()** Ancestral depth of the current population

**Note:** The return value is the number of ancestral generations exist in the population, not necessarily equals to the number set by `setAncestralDepth()`.

**x.ancestralGen()** Currently used ancestral population (0 for the latest generation)

Current ancestral population activated by `useAncestralPop()`. There can be several ancestral generations in a population. 0 (current), 1 (parental) etc. When `useAncestralPop(gen)` is used, current generation is set to one of the parental generations, which is the information returned by this function. `useAncestralPop(0)` should always be used to set a population to its usual ancestral order after operations to the ancestral generation are done.

**x.arrGenotype(order)** Get the whole genotypes

Return an editable array of all genotypes of the population. You need to know how these genotypes are organized to safely read/write genotype directly.

**order** if `order` is `true`, individuals will be ordered such that `pop.individual(x).arrGenotype() == pop.arrGenotype()[x*pop.genoSize():(x+1)*pop.genoSize()]`.

**x.arrGenotype(subPop, order)** Get the whole genotypes of individuals in a subpopulation

Return an editable array of all genotype in a subpopulation.

**order** if `order` is `true`, individuals will be ordered.

**subPop** index of subpopulation (start from 0)

**x.clone(keepAncestralPops=-1)** Deep copy of a population. (In python, `pop1 = pop` will only create a reference to `pop`.)

This function by default copies all ancestral generations, but you can copy only one (current, `keepAncestralPops=0`), or specified number of ancestral generations.

**x.deactivateVirtualSubPop(subPop)** deactivate virtual subpopulations in a given subpopulation. In another word, all individuals will become visible.

**Note:** this function is currently not recommended to be used.

**x.evaluate(expr="", stmts="")** Evaluate a Python statement/expression in the population's local namespace

This function evaluates a Python statement( `stmts` )/expression( `expr` ) and return its result as a string. Optionally run statement( `stmts` ) first.

**x.execute(stmts="")** Execute a statement (can be a multi-line string) in the population's local namespace

**x.gen()** Current generation during evolution

**x.grp()** Current group ID in a simulator which is not meaningful for a stand-alone population.

**x.hasVirtualSubPop()** If a population has any virtual subpopulation

**x.indInfo(idx)** Get information field `idx` of all individuals  
     **idx** index of the information field

**x.indInfo(name)** Get information field `name` of all individuals  
     **name** name of the information field

**x.indInfo(idx, subPop)** Get information field `idx` of all individuals in a subpopulation `subPop`  
     **idx** index of the information field  
     **subPop** subpopulation index

**x.indInfo(name, subPop)** Get information field `name` of all individuals in a subpopulation `subPop`  
     **name** name of the information field  
     **subPop** subpopulation index

**x.individual(ind, subPop=0)** Reference to individual `ind` in subpopulation `subPop`  
     This function is named `individual` in the Python interface.  
     **ind** individual index within `subPop`  
     **subPop** subpopulation index

**x.individuals()** Return an iterator that can be used to iterate through all individuals  
     Typical usage is  

```
for ind in pop.individuals():
```

**x.individuals(subPop)** Return an iterator that can be used to iterate through all individuals in subpopulation `subPop`

**x.individuals(subPop, virtualSubPop)** `SimuPOP::population::individuals`

**x.insertAfterLoci(idx, pos, names=[])** Append loci after given positions  
     Append loci at some given locations. Alleles at inserted loci are initialized with zero allele.  
     **idx** an array of locus index. The loci will be added *after* each index. If you need to append to the first locus of a chromosome, use `insertBeforeLoci` instead. If your index is the last locus of a chromosome, the appended locus will become the last locus of that chromosome. If you need to append multiple loci after a locus, repeat that locus number.  
     **names** an array of locus names. If this parameter is not given, some unique names such as "insX\_Y" will be given.  
     **pos** an array of locus positions. The positions of the appended loci have to be between adjacent markers.

**x.insertAfterLocus(idx, pos, name=string)** Append an locus after a given position  
     `insertAfterLocus(idx, pos, name)` is a shortcut to `insertAfterLoci([idx], [pos], [name])`.

**x.insertBeforeLoci(idx, pos, names=[])** Insert loci before given positions  
     Insert loci at some given locations. Alleles at inserted loci are initialized with zero allele.  
     **idx** an array of locus index. The loci will be inserted *before* each index. If you need to append to the last locus, use `insertAfterLoci` instead. If your index is the first locus of a chromosome, the inserted locus will become the first locus of that chromosome. If you need to insert multiple loci before a locus, repeat that locus number.

**names** an array of locus names. If this parameter is not given, some unique names such as "insX\_Y" will be given.

**pos** an array of locus positions. The positions of the appended loci have to be between adjacent markers.

**x.insertBeforeLocus(idx, pos, name=string)** Insert an locus before a given position

`insertBeforeLocus(idx, pos, name)` is a shortcut to `insertBeforeLoci([idx], [pos], [name])`

**x.locateRelatives(relType, relFields, gen=-1, relSex=AnySex, parentFields=[])**

Find relatives of each individual and fill the given information fields with their indexes.

This function locates relatives of each individual and store their indexes in given information fields.

**gen** Find relatives for individuals for how many generations. Default to -1, meaning for all generations. If a non-negative number is given, up till generation `gen` will be processed.

**parentFields** information fields that stores parental indexes. Default to ['father\_idx', 'mother\_idx']

**relFields** information fields to hold relatives. The number of these fields limits the number of relatives to locate.

**relSex** Whether or not only locate relative or certain sex. It can be AnySex (do not care, default), MaleOnly, FemaleOnly, or OppositeSex (only locate relatives of opposite sex).

**relType** Relative type, can be

- REL\_Self index of individual themselves
- REL\_Spouse index of spouse in the current generation. Spouse is defined as two individuals having an offspring with shared `parentFields`. If more than one `infoFields` is given, multiple spouses can be identified.
- REL\_Offspring index of offspring in the offspring generation. If only one parent is given, only paternal or maternal relationship is considered. For example, `parentFields=['father_idx']` will locate offspring for all fathers.
- REL\_FullSibling all siblings with the same parents
- REL\_Sibling all sibs with at least one shared parent

**x.mergePopulation(pop, newSubPopSizes=[], keepAncestralPops=-1)** Merge populations by individuals

Merge individuals from `pop` to the current population. Two populations should have the same genotypic structures. By default, subpopulations of the merged populations are kept. I.e., if you merge two populations with one subpopulation, the resulting population will have two subpopulations. All ancestral generations are also merged by default.

**keepAncestralPops** ancestral populations to merge, default to all (-1)

**newSubPopSizes** subpopulation sizes can be specified. The overall size should be the combined size of the two populations. Because this parameter will be used for all ancestral generations, it may fail if ancestral generations have different sizes. To avoid this problem, you can run `mergePopulation` without this parameter, and then adjust subpopulation sizes generation by generation.

**Note:** Population variables are not copied to `pop`.

**x.mergePopulationByLoci(pop, newNumLoci=[], newLociPos=[], byChromosome=False)**

Merge populations by loci

Two populations should have the same number of individuals. This also holds for any ancestral generations. By default, chromosomes of `pop` are appended to the current population. You can change this arrangement in two ways

- specify new chromosome structure using parameter `newLoci` and `newLociPos`. Loci from new and old populations are still in their original order, but chromosome number and positions can be changed in this way.

- specify `byChromosome=true` so that chromosomes will be merged one by one. In this case, loci position of two populations are important because loci will be arranged in the order of loci position; and identical loci position of two loci in two populations will lead to error.

**byChromosome** merge chromosome by chromosome, loci are ordered by loci position Default to `False`.

**newLociPos** the new loci position if number of loci on each chromosomes are changed with `newNumLoci`.

New loci positions should be in order on the new chromosomes.

**newNumLoci** the new number of loci for the combined genotypic structure.

**Note:**

- Information fields are not merged.
- All ancestral generations are merged because all individuals in a population have to have the same genotypic structure.

**x.mergeSubPops(subPops=[], removeEmptySubPops=False)** Merge given subpopulations

Merge subpopulations, the first subpopulation ID (the first one in array `subPops`) will be used as the ID of the new subpopulation. That is to say, all merged subpopulations will take the ID of the first one. The subpopulation ID of the empty subpopulations will be kept (so that other subpopulations are unaffected, unless they are removed by `removeEmptySubPops = True`).

**x.newPopByIndID(keepAncestralPops=-1, id=[], removeEmptySubPops=False)** Form a new population according to individual subpopulation ID. Individuals with negative subpopulation ID will be removed.

**x.newPopWithPartialLoci(remove=[], keep=[])** Obtain a new population with selected loci

Copy current population to a new one with selected loci `keep` or remove specified loci `remove` (no change on the current population), equivalent to

```
y=x.clone
y.removeLoci(remove, keep)
```

**x.numSubPop()** Number of subpopulations in a population.

**x.numVirtualSubPop()** Number of virtual subpopulations.

**x.popSize()** Total population size

**x.pushAndDiscard(rhs, force=False)** Absorb `rhs` population as the current generation of a population

This function is used by a simulator to push offspring generation `rhs` to the current population, while the current population is pushed back as an ancestral population (if `ancestralDepath() != 0`). Because `rhs` population is swapped in, `rhs` will be empty after this operation.

**x.removeEmptySubPops()** Remove empty subpopulations by adjusting subpopulation IDs

**x.removeIndividuals(inds=[], subPop=-1, removeEmptySubPops=False)** Remove individuals. If a valid `subPop` is given, remove individuals from this subpopulation. Indexes in `inds` will be treated as relative indexes.

**x.removeLoci(remove=[], keep=[])** Remove some loci from the current population. Only one of the two parameters can be specified.

**x.removeSubPops(subPops=[], shiftSubPopID=True, removeEmptySubPops=False)**

Remove subpopulations and adjust subpopulation IDs so that there will be no 'empty' subpopulation left

Remove specified subpopulations (and all individuals within). If `shiftSubPopID` is false, `subPopID` will be kept intactly.

**x.reorderSubPops (order=[], rank=[], removeEmptySubPops=False)** Reorder subpopulations by order or by rank

**order** new order of the subpopulations. For examples, 3 2 0 1 means subpop3, subpop2, subpop0, subpop1 will be the new layout.

**rank** you may also specify a new rank for each subpopulation. For example, 3,2,0,1 means the original subpopulations will have new IDs 3,2,0,1, respectively. To achieve order 3,2,0,1, the rank should be 1 0 2 3.

**x.rep()** Current replicate in a simulator which is not meaningful for a stand-alone population

**x.resize(newSubPopSizes, propagate=False)** Resize current population

Resize population by giving new subpopulation sizes.

**newSubPopSizes** an array of new subpopulation sizes. If there is only one subpopulation, use [newPopSize].

**propagate** if propagate is true, copy individuals to new comers. I.e., 1, 2, 3 ==> 1, 2, 3, 1, 2, 3, 1

**Note:** This function only resizes the current generation.

**x.savePopulation(filename, format="auto", compress=True)** Save population to a file

**filename** save to filename

**format** format to save. Can be one of the following: 'txt', 'bin', or 'xml', or 'auto' which is determined by the extension of filename.

**x.setAncestralDepth(depth)** Set ancestral depth

**depth** 0 for none, -1 for unlimited, a positive number sets the number of ancestral generations to save.

**x.setIndInfo(values, idx)** Set individual information for the given information field idx

**idx** index to the information field.

**values** an array that has the same length as population size.

**x.setIndInfo(values, name)** Set individual information for the given information field name

x.setIndInfo(values, name) is equivalent to the idx version x.setIndInfo(values, x.infoIdx(name)).

**x.setIndSubPopID(id, ancestralPops=False)** Set subpopulation ID with given ID

Set subpopulation ID of each individual with given ID. Individuals can be rearranged afterwards using setSubPopByIndID.

**ancestralPops** If true (default to False), set subpop id for ancestral generations as well.

**id** an array of the same length of population size, representing subpopulation ID of each individual. If the length of id is less than population size, it is repeated to fill the whole population.

**x.setIndSubPopIDWithID(ancestralPops=False)** Set subpopulation ID of each individual with their current subpopulation ID

**ancestralPops** If true (default to False), set subpop id for ancestral generations as well.

**x.setIndexesOfRelatives(pathGen, pathFields, pathSex=[], resultFields=[])** Trace a relative path in a population and record the result in the given information fields.

For example, `setInfoWithRelatives(pathGen = [0, 1, 1, 0], pathFields = [['father_idx', 'mother_idx'], ['sib1', 'sib2'], ['off1', 'off2']], pathSex = [AnySex, MaleOnly, FemaleOnly], resultFields = ['cousin1', 'cousin2'])` This function will 1. locate `father_idx` and `mother_idx` for each individual at generation 0 (`pathGen[0]`) 2. find AnySex individuals referred by `father_idx` and `mother_idx` at generation 1 (`pathGen[1]`) 3. find information fields `sib1` and `sib2` from these parents 4. locate MaleOnly individuals referred by `sib1` and `sib2` from generation 1 (`pathGen[2]`) 5. find information fields `off1` and `off2` from these individuals, and 6. locate FemaleOnly individuals referred by `off1` and from generation 0 (`pathGen[3]`) 7. Save index of these individuals to information fields `cousin1` and `cousin2` at generation `pathGen[0]`. In short, this function locates father or mother's brother's daughters.

**pathFields** A list of list of information fields forming a path to trace a certain type of relative.

**pathGen** A list of generations that form a relative path. This array is one element longer than `pathFields`, with `gen_i`, `gen_i+1` indicating the current and destinating generation of information fields `path_i`.

**pathSex** (Optional) A list of sex choices, AnySex, Male, Female or OppositeSex, that is used to choose individuals at each step. Default to AnySex.

**resultFields** Where to store located relatives. Note that the result will be saved in the starting generation specified in `pathGen[0]`, which is usually 0.

**x.setInfoFields(fields, init=0)** Set information fields for an existing population. The existing fields will be removed.

**fields** an array of fields

**init** initial value for the new fields.

**x.setSubPopByIndID(id=[])** Move individuals to subpopulations according to individual subpopulation IDs. Rearrange individuals to their new subpopulations according to their subpopulation ID (or the new given ID). Order within each subpopulation is not respected.

**id** new subpopulation ID, if given, current individual subpopulation ID will be ignored.

**Note:** Individual with negative info will be removed!

**x.setSubPopStru(newSubPopSizes)** Set population/subpopulation structure given subpopulation sizes

**newSubPopSizes** an array of new subpopulation sizes. The overall population size should not changed.

**x.setVirtualSplitter(vsp)** Set a virtual splitter to the population. If multiple splitter is needed for different subpopulations, use a combined splitter.

**vsp** a virtual subpop splitter

**x.splitSubPop(which, sizes, subPopID=[])** Split a subpopulation into subpopulations of given sizes. The sum of given sizes should be equal to the size of the split subpopulation. Subpopulation IDs can be specified. The subpopulation IDs of non-split subpopulations will be kept. For example, if subpopulation 1 of 0 1 2 3 is split into three parts, the new subpop id will be 0 (1 4 5) 2 3.

**Note:** subpop with negative ID will be removed. So, you can shrink one subpop by splitting and setting one of the new subpop with negative ID.

**x.splitSubPopByProportion(which, proportions, subPopID=[])** Split a subpopulation into subpopulations of given proportions

The sum of given proportions should add up to one. Subpopulation IDs can be specified.

**Note:** subpop with negative ID will be removed. So, you can shrink one subpop by splitting and setting one of the new subpop with negative ID.

**x.subPopBegin(subPop)** Index of the first individual of a subpopulation `subPop`

**x.subPopEnd(subPop)** Return the value of the index of the last individual of a subpopulation `subPop` plus 1

**x.subPopIndPair(ind)** Return the subpopulation ID and relative index of an individual with absolute index `ind`

**x.subPopSize(subPop)** Return size of a subpopulation `subPop`.  
     **subPop** index of subpopulation (start from 0)

**x.subPopSizes()** Return an array of all subpopulation sizes.

**x.swap(rhs)** Swap the content of two populations

**x.turnOffSelection()** Turn off selection for all subpopulations  
     This is only used when you would like to apply two selectors. Maybe using two different information fields.

**x.useAncestralPop(idx)** Use an ancestral generation. 0 for the latest generation.  
     **idx** Index of the ancestral generation. 0 for current, 1 for parental, etc. `idx` can not exceed ancestral depth (see `setAncestralDepth()`).

**x.vars(subPop=-1)** Return variables of a population. If `subPop` is given, return a dictionary for specified subpopulation.

**x.virtualSubPopName(subPop, virtualSubPop=InvalidSubPopID)** Name of the given virtual subpopulation.  
     **id** subpopulation id  
     **vid** virtual subpopulation id

**x.virtualSubPopSize(subPop, virtualSubPop=InvalidSubPopID)** return the size of virtual subpopulation `subPop`. if `subPop` is activated, and `subPop` does not specify which virtual subpopulation to count, the currently activated virtual subpop is returned. Therefore, When it is not certain if a subpopulation has activated virtual subpopulation, this function can be used.  
     **id** subpopulation id  
     **vid** virtual subpopulation id. If not given, current subpopulation, or current actived subpopulation size will be returned.

## 2.2.2 Ancestral populations

By default, a population object only holds the current generation. All ancestral populations (generations) will be discarded. You can, however, keep as many ancestral generations as you wish, provided that you have enough RAM to store all these extra information.

Parameter `ancestralDepth` is used to specify the number of generations to keep. This parameter is default to 0, meaning keeping no ancestral population. You can specify a positive number `n` to store most recent `n` generations; or -1 to store all populations.

Several important usage of ancestral generations:

- `dumper()` operator and `Dump()` function has a parameter `ancestralPops`. If set to `True`, they will dump all ancestral generations.
- function `population::setAncestralDepth()` and operator `setAncestralDepth()` set the number of ancestral generations to keep for a population. A typical use of `setAncestralDepth()` is



```

simu.evolve(...
    setAncestralDepth(3, at=[-3])
)

```

which saves the last three generations in populations so that pedigree based sampling schemes can be used.

- `pop.useAncestralPop(idx)` set the current generation of population `pop` to `idx` generation. `idx` = 1 for the first ancestral generation, 2 for second ancestral ..., and 0 for the current generation. After this function, all functions, operators will be applied to this ancestral generation. You should always call `setAncestralPop(0)` after you examined the ancestral generations.

A typical use of this function is demonstrated in example 2.1. In this example, a population with two loci is created and with initial genotype 0. Two `kamMutator` with different mutation rates are applied to these two loci. Five most recent populations are kept. The allele frequencies at these generations are calculated afterward. (Note that this is not the best way to exam the changes of allele frequencies, a `stat` operator should be used.)

#### Example 2.1: Ancestral populations

```

>>> simu = simulator(population(10000, loci=[2]), randomMating())
>>> simu.evolve(
...     ops = [
...         setAncestralDepth(5, at=[-5]),
...         kamMutator(rate=0.01, loci=[0], maxAllele=1),
...         kamMutator(rate=0.001, loci=[1], maxAllele=1)
...     ],
...     end = 20
... )
Parameter end is obsolete in simulator::evolve(), please use gen instead
True
>>> pop = simu.population(0)
>>> # start from current generation
>>> for i in range(pop.ancestralDepth()+1):
...     pop.useAncestralPop(i)
...     Stat(pop, alleleFreq=[0,1])
...     print '%d      %5f      %5f' % \
...           (i, pop.dvars().alleleFreq[0][1], pop.dvars().alleleFreq[1][1])
...
0      0.170200      0.020100
1      0.163700      0.017450
2      0.154900      0.015500
3      0.147400      0.014500
4      0.142200      0.013950
5      0.137800      0.012500
>>> # restore to the current generation
>>> pop.useAncestralPop(0)
>>>

```

### 2.2.3 Save and Load a Population

Internally, population can be saved/loaded in “txt”, “xml” or “bin” formats using `savePopulation(file, format, compress=True)` member function, global `SavePopulation(pop, file, format)` and `LoadPopulation`. (Yes, it is `Load..` not `load..` since `savePopulation` is a member function and `LoadPopulation` is a global function.) These formats have their own advantages and disadvantages:

- `xml` : most readable, easy transformation to other formats, largest file size

- `bin`: not readable, small file size. May not be portable.
- `txt`: human readable with no structure, portable, median file size.
- `auto`: the format is determined by the filename extension specified.

Populations are by default compressed in gzip format. If you are interested in viewing the content of the file, you can use `compress=False` when saving a population, or decompress the saved files using `gzip -d` command.

Populations can also be saved in other formats such as FSTAT so that they can be directly analyzed by other programs. These formats are not supported internally. They are handled in Python in the form of Python function or pure-Python operator. If you would like to save/load simuPOP population in your own format, you can do it by mimicking these functions in `simuUtil.py`.

Shared variables (c.f section 2.8) are also saved (except for big objects like samples). Since the number of shared variables can be very large, it maybe a good idea to clear these variables before you save a population. On the other hand, you may want to save key parameters used to generate this population in the local namespace so that you will know these parameters after the population is loaded. For example, you can do

Example 2.2: Save population variables

```
pop.vars().clear()
pop.dvars().migrationRate = 0.002
pop.dvars().diseaseLocs = [4, 30]
SavePopulation(pop, 'pop.bin')
```

## 2.2.4 View a population (GUI, wxPython required)

Introduced in version 0.6.9, `simuViewPop.py` can be used to view a population. It can be used as a standalone application, or in an interactive session. First, you can use this script as a standalone application, simply run

```
simuViewPop.py mypop.bin
```

will fire a GUI and allow you to exam population property, genotype and calculate statistics.

In a Python session, import this module will provide a function `viewPop`, apply it on a in-memory population or a filename will have the same effect. For example,

Example 2.3: Use `simuViewPop` to view a population

```
import simuViewPop
simuViewPop.viewPop(myPop)
simuViewPop.viewPop(filename='mypop.bin')
```

## 2.3 Virtual subpopulations

simuPOP 0.8.2 introduces the concept *virtual subpopulations*. Virtual subpopulations are groups of individuals in a subpopulation, defined by certain criteria. For example, all male individuals, all unaffected individuals, all individuals with information field `age > 20`, all individuals with genotype 0, 0 at a given locus, can form virtual subpopulations. Virtual subpopulations do not have to add up to the whole subpopulation, nor do they have to be distinct. Because properties of individuals are variable, virtual subpopulations do not have fixed sizes as subpopulations do.

Virtual subpopulations allow easy handling of heterogeneous populations, and can facilitate some computations that are previously very difficult to do. For example, mating schemes can work on virtual subpopulations. This allows complicated mating schemes such as mating in aged population, and mixed mating schemes. By limiting operators to virtual subpopulations, one can apply different genetic forces to different groups of individuals. A good example is to

migrant only male from a subpopulation to other subpopulations. It is also easy to calculate statistics at a finer scale, such as allele frequency of all males.

Virtual subpopulations are defined by virtual splitters. A splitter splits a subpopulation into pre-determined number of virtual subpopulations. It also assign a name, such as `age=5` to each virtual subpopulation. For example

- A `sexSplitter` splits the population into male and female virtual subpopulations. A `affectionSplitter` splits the population into unaffected and affected virtual subpopulations.
- A `infoSplitter` splits the population according to values of a given information field. It can split the population by given values, or by some cut-off values.
- A `proportionSplitter` splits the population with given proportions, and a `rangeSplitters` choose individuals from given ranges.
- A `genotypeSplitter` splits the population with genotype values at given loci. Multiple genotypes are allowed for a virtual subpopulation. For example, `genotypeSplitter(1, [0, 0, 0, 1])` defines a virtual subpopulation with individuals having genotype 0, 0 or 0, 1 at locus 1.
- A `combinedSplitter` allows the specification of multiple splitter at the same subpopulation. For example, the unaffected and affected virtual subpopulation of a subpopulation split by `combinedSplitter([sexSplitter(), affectionSplitter()])` are 2 and 3, respectively.

There is currently no easy way to get the intersection or superset of two virtual subpopulations, such as a virtual subpopulation with male and/or affected individuals. It is possible, though, to define an information field that reflect these logics and define a virtual subpopulation according to this information field.

The population class provides several functions to assign a splitter to a given population, retrieve virtual subpopulation sizes and names. Note that one splitter is used for all subpopulations. If different splitters are needed for different subpopulations, a combined splitter can be used. More interestingly, the `individuals(subPop, virtualSubPop)` member function allows you to iterate through all individuals of a virtual subpopulation.

Member functions `pop.numVirtualSubPop(sp)`, `pop.virtualSubPopSize(sp, vsp)` can be used to determine the number of virtual subpopulation a subpopulation has, and the size of the virtual subpopulation. Operator `stat(popSize=True)` also calculates virtual subpopulation sizes, and save them in a variable `virtualPopSize`.

Example 2.4 demonstrates how to assign virtual splitter, and how to use them.

#### Example 2.4: Virtual subpopulation related functions

```
>>> import random
>>> pop = population(1000, loci=[2, 3], infoFields=['age'])
>>> InitByFreq(pop, [0.2, 0.8])
>>> for ind in pop.individuals():
...     ind.setInfo(random.randint(0,5), 'age')
...
>>> # split by age
>>> pop.setVirtualSplitter(infoSplitter('age', values=[2,4]))
>>> pop.virtualSubPopSize(0, 0)
170
>>> pop.virtualSubPopName(0, 1)
'age = 4'
>>>
>>> # split by genotype
>>> a = pop.setVirtualSplitter(
...     genotypeSplitter(locus=2, alleles=[[0,1], [1,1]], phase=True))
>>> pop.virtualSubPopSize(0, 0)
178
```

```

>>> pop.virtualSubPopSize(0, 1)
620
>>>
>>> for ind in pop.individuals(0, 0):
...     assert ind.allele(2, 0) == 0 and ind.allele(2, 1) == 1
...
>>>

```

### 2.3.1 Class `sexSplitter`

#### Details

Split the population into Male and Female virtual subpopulations

#### Initialization

`SimuPOP::sexSplitter::sexSplitter`

```
sexSplitter()
```

#### Member Functions

**x.clone()** `SimuPOP::sexSplitter::clone`

**x.name(sp)** Name of a virtual subpopulation

**x.numVirtualSubPop()** Number of virtual subpops of subpopulation `sp`

### 2.3.2 Class `affectionSplitter`

#### Details

Split a subpopulation into unaffected and affected virtual subpopulations.

#### Initialization

`SimuPOP::affectionSplitter::affectionSplitter`

```
affectionSplitter()
```

#### Member Functions

**x.clone()** `SimuPOP::affectionSplitter::clone`

**x.name(sp)** Name of a virtual subpopulation

**x.numVirtualSubPop()** Number of virtual subpops of subpopulation `sp`

### 2.3.3 Class `infoSplitter`

#### Details

Split the population according to the value of an information field. A list of distinct values, or a cutoff vector can be given to determine how the virtual subpopulations are divided. Note that in the first case, an individual does not have to belong to any virtual subpopulation.

## Initialization

SimuPOP::infoSplitter::infoSplitter

```
infoSplitter(info, values=[], nfo, cutoff=[])
```

**cutoff** A list of cutoff values. For example, cutoff=[1, 2] defines three virtual subpopulations with  $v < 1$ ,  $1 \leq v < 2$ , and  $v \geq 2$ .

**info** Name of the information field

**values** A list of values, each defines a virtual subpopulation

## Member Functions

**x.name(sp)** Name of a virtual subpopulation

**x.numVirtualSubPop()** Number of virtual subpops of subpopulation sp

### 2.3.4 Class proportionSplitter

#### Details

Split the population according to a proportion

#### Initialization

SimuPOP::proportionSplitter::proportionSplitter

```
proportionSplitter(proportions=[])
```

**proportions** A list of float numbers (between 0 and 1) that defines the proportion of individuals in each virtual subpopulation. These numbers should add up to one.

## Member Functions

**x.clone()** SimuPOP::proportionSplitter::clone

**x.name(sp)** Name of a virtual subpopulation

**x.numVirtualSubPop()** Number of virtual subpops of subpopulation sp

### 2.3.5 Class rangeSplitter

#### Details

Split the population according to individual range. The ranges can overlap and does not have to add up to the whole subpopulation.

#### Initialization

SimuPOP::rangeSplitter::rangeSplitter

```
rangeSplitter(ranges)
```

**range** A shortcut for ranges=[range]

**ranges** A list of ranges

## Member Functions

**x.clone()** SimuPOP::rangeSplitter::clone

**x.name(sp)** Name of a virtual subpopulation

**x.numVirtualSubPop()** Number of virtual subpops of subpopulation sp

## 2.3.6 Class `genotypeSplitter`

### Details

Split the population according to given genotype

### Initialization

```
genotypeSplitter(loci, alleles, phase=False)
```

For example, Genotype Aa or aa at locus 1: locus = 1, alleles = [0, 1] Genotype Aa at locus 1 (assuming A is 1): locus = 1, alleles = [1, 0], phase = True Genotype AaBb at loci 1 and 2: loci = [1, 2], alleles = [1, 0, 1, 0], phase = True Two virtual subpopulations with Aa and aa locus = 1, alleles = [[1, 0], [0, 0]], phase = True A virtual subpopulation with Aa or aa locus = 1, alleles = [1, 0, 0, 0] Two virtual subpopulation with genotype AA and the rest locus = 1, alleles = [[1, 1], [1, 0, 0, 0]], phase = False

**alleles** A list (for each virtual subpopulation), of a list of alleles at each locus. If phase if true, the order of alleles is significant. If more than one set of alleles are given, individuals having either of them is qualified.

**loci** A list of locus at which alleles are used to classify individuals

**locus** A shortcut to loci=[locus]

**phase** Whether or not phase is respected.

## Member Functions

**x.clone()** SimuPOP::genotypeSplitter::clone

**x.name(sp)** Name of a virtual subpopulation

**x.numVirtualSubPop()** Number of virtual subpops of subpopulation sp

## 2.3.7 Class `combinedSplitter`

### Details

This plitter takes several splitters, and stacks their virtual subpopulations together. For example, if the first splitter has three vsp, the second has two. The two vsp from the second splitter will be the fourth (index 3) and fifth (index 4) of the combined splitter.

### Initialization

SimuPOP::combinedSplitter::combinedSplitter

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

## Member Functions

**x.clone()** SimuPOP::combinedSplitter::clone

**x.name(sp)** Name of a virtual subpopulation

**x.numVirtualSubPop()** Number of virtual subpops of subpopulation sp

## 2.4 Individuals

Individuals of a population can be accessed through `individual()`, or its iteration form `individuals()` function:

- `individual(ind)` returns the `ind`'th individual (absolute index) of the whole population.
- `individual(ind, subPop)` returns the `ind`'th (relative index) individual in the `subPop`'th subpopulation.
- `individuals()` return an 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 the `subPop`'th subpopulations.
- `ancestor(ind, gen)` returns the `ind`'th individual (absolute index) of the `gen`'th ancestral generation.
- `ancestor(ind, subPop, gen)` returns the `ind`'th (relative index) individual in the `subPop`'th subpopulation.

For example, example 2.5 iterates through all individuals in subpopulation 2 using `population::individual()` function, while 2.6 uses `population::individuals()`. The latter is usually easier to use.

You can also access individuals from the ancestral generations directly. There is no batch access functions such as `individuals()`. If they are needed, use `useAncestralPop()` to switch to that ancestral generation and run `individuals()` for the current generation.

Example 2.5: Function `population::individual()`

```
for i in range(pop.subPopSize(2)):
    ind = pop.individual(i, 2)
    print ind.affected()
```

Example 2.6: Function `population::individuals()`

```
for ind in pop.individuals(2):
    # do something to ind
    print ind.affected()
```

### 2.4.1 Class `individual`

Individuals with genotype, affection status, sex etc.

#### Details

Individuals are the building blocks of populations, each having the following individual information:

- shared genotypic structure information

- genotype
- sex, affection status, subpopulation ID
- optional information fields

Individual genotypes are arranged by locus, chromosome, ploidy, in that order, and can be accessed from a single index. For example, for a diploid individual with two loci on the first chromosome, one locus on the second, its genotype is arranged as 1-1-1 1-1-2 1-2-1 2-1-1 2-1-2 2-2-1 where x-y-z represents ploidy x chromosome y and locus z. An allele 2-1-2 can be accessed by `allele(4)` (by absolute index), `allele(1, 1)` (by index and ploidy) or `allele(1, 1, 0)` (by index, ploidy and chromosome). Individuals are created by populations automatically. Do not call the constructor function directly.

## Initialization

`SimuPOP::individual::individual`

```
individual()
```

## Member Functions

**x.affected()** Whether or not an individual is affected

**x.affectedChar()** Return A (affected) or U (unaffected) for affection status

**x.allele(index)** Return the allele at locus `index`

**index** absolute index from the beginning of the genotype, ranging from 0 to `totNumLoci()*ploidy()`

**x.allele(index, p)** Return the allele at locus `index` of the `p`-th copy of the chromosomes

**index** index from the beginning of the `p`-th set of the chromosomes, ranging from 0 to `totNumLoci()`

**p** index of the ploidy

**x.allele(index, p, ch)** Return the allele at locus `index` of the `ch`-th chromosome in the `p`-th chromosome set

**ch** index of the chromosome in the `p`-th chromosome set

**index** index from the beginning of chromosome `ch` of ploidy `p`, ranging from 0 to `numLoci(ch)`

**p** index of the ploidy

**x.alleleChar(index)** Return the name of `allele(index)`

**x.alleleChar(index, p)** Return the name of `allele(index, p)`

**x.alleleChar(index, p, ch)** Return the name of `allele(idx, p, ch)`

**x.arrGenotype()** Return an editable array (a carry of length `totNumLoci()*ploidy()`) of genotypes of an individual

This function returns the whole genotype. Although this function is not as easy to use as other functions that access alleles, it is the fastest one since you can read/write genotype directly.

**x.arrGenotype(p)** Return a carry with the genotypes of the `p`-th copy of the chromosomes

**x.arrGenotype(p, ch)** Return a carry with the genotypes of the `ch`-th chromosome in the `p`-th chromosome set

**x.arrInfo()** Return a carry of all information fields (of size `infoSize()`) of this individual



**x.info(idx)** Get information field `idx`  
**idx** index of the information field

**x.info(name)** Get information field `name`  
Equivalent to `info(infoIdx(name))`.  
**name** name of the information field

**x.intInfo(idx)** Get information field `idx` as an integer. This is the same as `int(info(idx))`  
**idx** index of the information field

**x.intInfo(name)** Get information field `name` as an integer  
Equivalent to `int(info(name))`.  
**name** name of the information field

**x.setAffected(affected)** Set affection status

**x.setAllele(allele, index)** Set the allele at locus `index`  
**allele** allele to be set  
**index** index from the beginning of genotype, ranging from 0 to `totNumLoci()*ploidy()`

**x.setAllele(allele, index, p)** Set the allele at locus `index` of the `p`-th copy of the chromosomes  
**allele** allele to be set  
**index** index from the beginning of the ploidy `p`, ranging from 0 to `totNumLoci(p)`  
**p** index of the ploidy

**x.setAllele(allele, index, p, ch)** Set the allele at locus `index` of the `ch`-th chromosome in the `p`-th chromosome set  
**allele** allele to be set  
**ch** index of the chromosome in ploidy `p`  
**index** index from the beginning of the chromosome, ranging from 0 to `numLoci(ch)`  
**p** index of the ploidy

**x.setInfo(value, idx)** Set information field by `idx`

**x.setInfo(value, name)** Set information field by `name`

**x.setSex(sex)** Set sex. `sex` can be Male or Female.

**x.setSubPopID(id)** Set new subpopulation ID, `pop.rearrangeByIndID` will move this individual to that population

**x.sex()** Return the sex of an individual, 1 for males and 2 for females.

**x.sexChar()** Return the sex of an individual, M or F

**x.subPopID()** Return the ID of the subpopulation to which this individual belongs  
**Note:** `subPopID` is not set by default. It only corresponds to the subpopulation in which this individual resides after `pop::setIndSubPopID` is called.

**x.unaffected()** Equals to `not affected()`

## 2.5 Mating Scheme

### 2.5.1 Class `mating`

The base class of all mating schemes - a required parameter of `simulator`

#### Details

Mating schemes specify how to generate offspring from the current population. It must be provided when a simulator is created. Mating can perform the following tasks:

- change population/subpopulation sizes;
- randomly select parent(s) to generate offspring to populate the offspring generation;
- apply *during-mating* operators;
- apply selection if applicable.

#### Initialization

Create a mating scheme (do not use this base mating scheme, use one of its derived classes instead)

```
mating(newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None,  
subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID, weight=0)
```

By default, a mating scheme keeps a constant population size, generates one offspring per mating event. These can be changed using certain parameters. `newSubPopSize`, `newSubPopSizeExpr` and `newSubPopSizeFunc` can be used to specify subpopulation sizes of the offspring generation.

**newSubPopSize** An array of subpopulations sizes, should have the same number of subpopulations as the current population

**newSubPopSizeExpr** An expression that will be evaluated as an array of new subpopulation sizes

**newSubPopSizeFunc** A function that takes parameters `gen` (generation number) and `oldsize` (an array of current population size) and return an array of subpopulation sizes of the next generation. This is usually easier to use than its expression version of this parameter.

**subPop** If this parameter is given, the mating scheme will be applied only to the given (virtual) subpopulation. This is only used in `heteroMating` where mating schemes are passed to.

**weight** When `subPop` is virtual, this is used to determine the number of offspring for this mating scheme. Weight can be

- 0 (default) the weight will be proportional to the current (virtual) subpopulation size. If other virtual subpopulation has non-zero weight, this virtual subpopulation will produce no offspring (weight 0).
- any negative number `-n`: the size will be  $n \cdot m$  where  $m$  is the size of the (virtual) subpopulation of the parental generation.
- any positive number `n`: the size will be determined by weights from all (virtual) subpopulations.

#### Member Functions

**`x.clone()`** Deep copy of a mating scheme

## 2.5.2 Class `noMating`

A mating scheme that does nothing

### Details

In this scheme, there is

- no mating. Parent generation will be considered as offspring generation.
- no subpopulation change. *During-mating* operators will be applied, but the return values are not checked. I.e., subpopulation size parameters will be ignored although some during-mating operators might be applied.

Note that because the offspring population is the same as parental population, this mating scheme can not be used with other mating schemes in a heterogeneous mating scheme. `cloneMating` is recommended for that purpose.

### Initialization

Create a scheme with no mating

```
noMating(numOffspring=1.0, numOffspringFunc=None, maxNumOffspring=0,  
mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex,  
newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None,  
subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID, weight=0)
```

### Note

All parameters are ignored!

### Member Functions

**x.clone()** Deep copy of a scheme with no mating

## 2.5.3 Class `cloneMating`

A clone mating that copy everyone from parental to offspring generation.

### Details

Note that

- selection is not considered (fitness is ignored)
- sequentialParentMating is used. If offspring (virtual) subpopulation size is smaller than parental subpopulation size, not all parents will be cloned. If offspring (virtual) subpopulation size is larger, some parents will be cloned more than once.
- numOffspring interface is respected.
- during mating operators are applied.

### Initialization

Create a binomial selection mating scheme

```
cloneMating(numOffspring=1., numOffspringFunc=None,  
maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5,  
sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeExpr="",  
newSubPopSizeFunc=None, subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID,  
weight=0)
```

Please refer to class `mating` for parameter descriptions.

### Member Functions

**x.clone()** Deep copy of a binomial selection mating scheme

## 2.5.4 Class `binomialSelection`

A mating scheme that uses binomial selection, regardless of sex

### Details

No sex information is involved (binomial random selection). Offspring is chosen from parental generation by random or according to the fitness values. In this mating scheme,

- `numOffspring` protocol is honored;
- population size changes are allowed;
- selection is possible;
- haploid population is allowed.

### Initialization

Create a binomial selection mating scheme

```
binomialSelection(numOffspring=1., numOffspringFunc=None,
maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5,
sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeExpr="",
newSubPopSizeFunc=None, subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID,
weight=0)
```

Please refer to class `mating` for parameter descriptions.

### Member Functions

**x.clone()** Deep copy of a binomial selection mating scheme

## 2.5.5 Class `baseRandomMating`

### Details

This base class defines a general random mating scheme that makes full use of a general random parents chooser, and a Mendelian offspring generator. A general random parents chooser allows selection without replacement, polygamous parents selection (a parent with more than one partners), and the definition of several alpha individuals. Direct use of this mating scheme is not recommended. `randomMating`, `monogamousMating`, `polygamousMating`, `alphaMating` are all special cases of this mating scheme. They should be used whenever possible.

### Initialization

`SimuPOP::baseRandomMating::baseRandomMating`

```
baseRandomMating(replacement=True, replenish=False,
polySex=Male, polyNum=1, alphaSex=Male, alphaNum=0,
alphaField=string, numOffspring=1., numOffspringFunc=None,
```

```

maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5,
sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeExpr="",
newSubPopSizeFunc=None, contWhenUniSex=True, subPop=InvalidSubPopID,
virtualSubPop=InvalidSubPopID, weight=0)

```

**alphaField** If an information field is given, individuals with non-zero values at this information field are alpha individuals. Note that these individuals must have `alphaSex`.

**alphaNum** Number of alpha individuals. If `infoField` is not given, `alphaNum` random individuals with `alphaSex` will be chosen. If selection is enabled, individuals with higher+ fitness values have higher probability to be selected. There is by default no alpha individual (`alphaNum = 0`).

**alphaSex** The sex of the alpha individual, i.e. alpha male or alpha female who be the only mating individuals in their sex group.

**polyNum** Number of sex partners.

**polySex** Sex of polygamous mating. Male for polygyny, Female for polyandry.

**replacement** If set to `True`, a parent can be chosen to mate again. Default to `False`.

**replenish** In case that `replacement=True`, whether or not replenish a sex group when it is exhausted.

## Member Functions

**x.clone()** Deep copy of a random mating scheme

## 2.5.6 Class randomMating

A mating scheme of basic sexually random mating

### Details

In this scheme, sex information is considered for each individual, and ploidy is always 2. Within each subpopulation, males and females are randomly chosen. Then randomly get one copy of chromosomes from father and mother. If only one sex exists in a subpopulation, a parameter (`contWhenUniSex`) can be set to determine the behavior. Default to continuing without warning.

### Initialization

```

randomMating(numOffspring=1., numOffspringFunc=None,
maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5,
sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None,
newSubPopSizeExpr="", contWhenUniSex=True, subPop=InvalidSubPopID,
virtualSubPop=InvalidSubPopID, weight=0)

```

Please refer to class `mating` for descriptions of other parameters.

**contWhenUniSex** Continue when there is only one sex in the population. Default to `True`.

## Member Functions

**x.clone()** Deep copy of a random mating scheme

## 2.5.7 Class `selfMating`

A mating scheme of selfing

### Details

In this mating scheme, a parent is chosen randomly, acts both as father and mother in the usual random mating. The parent is chosen randomly, regardless of sex. If selection is turned on, the probability that an individual is chosen is proportional to his/her fitness.

### Initialization

Create a self mating scheme

```
selfMating(numOffspring=1., numOffspringFunc=None,
maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5,
sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None,
newSubPopSizeExpr="", contWhenUniSex=True, subPop=InvalidSubPopID,
virtualSubPop=InvalidSubPopID, weight=0)
```

Please refer to class `mating` for descriptions of other parameters.

**contWhenUniSex** Continue when there is only one sex in the population. Default to `True`.

### Member Functions

**x.clone()** Deep copy of a self mating scheme

## 2.5.8 Class `monogamousMating`

A mating scheme of monogamy

### Details

This mating scheme is identical to random mating except that parents are chosen without replacement. Under this mating scheme, offspring share the same mother must share the same father. In case that all parental pairs are exhausted, parameter `replenish=True` allows for the replenishment of one or both sex groups.

### Initialization

```
monogamousMating(replenish=False, numOffspring=1.,
numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring,
sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[],
newSubPopSizeFunc=None, newSubPopSizeExpr="", contWhenUniSex=True,
subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID, weight=0)
```

**replenish** This parameter allows replenishment of one or both parental sex groups in case that they are exhausted. Default to `False`. Please refer to class `mating` for descriptions of other parameters.

### Member Functions

**x.clone()** Deep copy of a random mating scheme

## 2.5.9 Class polygamousMating

A mating scheme of polygyny or polyandry

### Details

This mating scheme is composed of a random parents chooser that allows for polygamous mating, and a mendelian offspring generator. In this mating scheme, a male (or female) parent will have more than one sex partner (`numPartner`). Parents returned from this parents chooser will yield the same male (or female) parents, each with varying partners.

### Initialization

`SimuPOP::polygamousMating::polygamousMating`

```
polygamousMating(polySex=Male, polyNum=1, replacement=False,
replenish=False, numOffspring=1., numOffspringFunc=None,
maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5,
sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None,
newSubPopSizeExpr="", contWhenUniSex=True, subPop=InvalidSubPopID,
virtualSubPop=InvalidSubPopID, weight=0)
```

**polyNum** Number of sex partners.

**polySex** Sex of polygamous mating. Male for polygyny, Female for polyandry.

**replacement** If set to `True`, a parent can be chosen to mate again. Default to `False`.

**replenish** In case that `replacement=True`, whether or not replenish a sex group when it is exhausted. Please refer to class `mating` for descriptions of other parameters.

### Member Functions

**`x.clone()`** Deep copy of a random mating scheme

## 2.5.10 Class consanguineousMating

A mating scheme of consanguineous mating

### Details

In this mating scheme, a parent is chosen randomly and mate with a relative that has been located and written to a number of information fields.

### Initialization

Create a consanguineous mating scheme

```
consanguineousMating(relativeFields=[], func=None, param=None,
replacement=False, replenish=True, numOffspring=1.,
numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring,
sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[],
newSubPopSizeFunc=None, newSubPopSizeExpr="", contWhenUniSex=True,
subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID, weight=0)
```

This mating scheme randomly choose a parent and then choose his/her spouse from indexes stored in `infoFields`. Please refer to `infoParentsChooser` and `mendelianOffspringGenerator` for other parameters.

**func** A python function that can be used to prepare the indexes of these information fields. For example, functions `population::locateRelatives` and/or `population::setIndexesOfRelatives` can be used to locate certain types of relatives of each individual.

**param** An optional parameter that can be passed to `func`.

**relativeFields** The information fields that stores indexes to other individuals in a population. If more than one valid (positive value) indexes exist, a random index will be chosen. (c.f. `infoParentsChooser` ) If there is no individual having any valid index, the second parent will be chosen randomly from the whole population.

## Member Functions

**x.clone()** Deep copy of a consanguineous mating scheme

### 2.5.11 Class `alphaMating`

Only a number of alpha individuals can mate with individuals of opposite sex.

#### Details

This mating scheme is composed of an random parents chooser with alpha individuals, and a Mendelian offspring generator. That is to say, a certain number of alpha individual (male or female) are determined by `alphaNum` or an information field. Then, only these alpha individuals are able to mate with random individuals of opposite sex.

#### Initialization

```
alphaMating(alphaSex=Male, alphaNum=0, alphaField=string,
numOffspring=1., numOffspringFunc=None, maxNumOffspring=0,
mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex,
newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="",
subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID, weight=0)
```

Please refer to class `mating` for descriptions of other parameters. Note: If selection is enabled, it works regularly on on-alpha sex, but works twice on alpha sex. That is to say, `alphaNum` alpha individuals are chosen selectively, and selected again during mating.

**alphaField** If an information field is given, individuals with non-zero values at this information field are alpha individuals. Note that these individuals must have `alphaSex`.

**alphaNum** Number of alpha individuals. If `infoField` is not given, `alphaNum` random individuals with `alphaSex` will be chosen. If selection is enabled, individuals with higher+ fitness values have higher probability to be selected. There is by default no alpha individual (`alphaNum = 0`).

**alphaSex** The sex of the alpha individual, i.e. alpha male or alpha female who be the only mating individuals in their sex group.

## Member Functions

**x.clone()** Deep copy of a random mating scheme



## 2.5.12 Class `haplodiploidMating`

Haplodiploid mating scheme of many hymenopterans

### Details

This mating scheme is composed of an `alphaParentChooser` and a `haplodiploidOffspringGenerator`. The `alphaParentChooser` chooses a single Female randomly or from a given information field. This female will mate with random males from the colony. The offspring will have one of the two copies of chromosomes from the female parent, and the first copy of chromosomes from the male parent. Note that if a recombinator is used, it should disable recombination of male parent.

### Initialization

```
haplodiploidMating(alphaSex=Female, alphaNum=1, alphaField=string,  
numOffspring=1., numOffspringFunc=None, maxNumOffspring=0,  
mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex,  
newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="",  
subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID, weight=0)
```

Please refer to class `mating` for descriptions of other parameters.

**alphaField** Information field that identifies the queen of the colony. By default, a random female will be chosen.

**alphaNum** Number of alpha individual. Default to one.

**alphaSex** Sex of the alpha individual. Default to Female.

### Member Functions

**x.clone()** Deep copy of a random mating scheme

## 2.5.13 Class `pedigreeMating`

A mating scheme that follows a given pedigree

### Details

In this scheme, a pedigree is given and the mating scheme will choose parents and produce offspring strictly following the pedigree. Parameters setting number of offspring per mating event, and size of the offspring generations are ignored. To implement this mating scheme in `pyMating`, 1.) a `newSubPopSizeFunc` should be given to return the exact subpopulation size, returned from `pedigree.subPopSizes(gen)`. 2.) use `pedigreeChooser` to choose parents 3.) use a suitable offspring generator to generate offspring. This `pedigreeMating` helps you do 1 and 2, and use a `mendelianOffspringGenerator` as the default offspring generator. You can use another offspring generator by setting the generator parameter. Note that the offspring generator can generate one and only one offspring each time.

### Initialization

```
pedigreeMating(generator, ped, newSubPopSize=[], newSubPopSizeFunc=None,  
newSubPopSizeExpr="", subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID,  
weight=0)
```

Please refer to class `mating` for descriptions of other parameters.

### Member Functions

**x.clone()** Deep copy of a random mating scheme

### 2.5.14 Class `pyMating`

A Python mating scheme

#### Details

This hybrid mating scheme does not have to involve a python function. It requires a parent chooser, and an offspring generator. The parent chooser chooses parent(s) and pass them to the offspring generator to produce offspring.

#### Initialization

Create a Python mating scheme

```
pyMating(chooser, generator, newSubPopSize=[], newSubPopSizeExpr="",
newSubPopSizeFunc=None, subPop=InvalidSubPopID, virtualSubPop=InvalidSubPopID,
weight=0)
```

**chooser** A parent chooser that chooses parent(s) from the parental generation.

**generator** An offspring generator that produce offspring of given parents.

#### Member Functions

**x.clone()** Deep copy of a Python mating scheme

### 2.5.15 Class `heteroMating`

#### Details

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

#### Initialization

Create a heterogeneous Python mating scheme

```
heteroMating(matingSchemes, newSubPopSize=[], newSubPopSizeExpr="",
newSubPopSizeFunc=None, shuffleOffspring=True, subPop=InvalidSubPopID,
virtualSubPop=InvalidSubPopID, weight=0)
```

Parameter subpop, virtualSubPOP and weight of this mating scheme is ignored.

**matingSchemes** A list of mating schemes. If parameter subPop of an mating scheme is specified, it will be applied to specific subpopulation. If virtualSubPop if specified, it will be applied to specific virtual subpopulations.

#### Member Functions

**x.clone()** Deep copy of a Python mating scheme

### 2.5.16 Class `sequentialParentChooser`

#### Details

This parent chooser chooses a parent linearly, regardless of sex or fitness values (selection is not considered).

#### Initialization

`SimuPOP::sequentialParentChooser::sequentialParentChooser`

```
sequentialParentChooser()
```

### Member Functions

**x.clone()** SimuPOP::sequentialParentChooser::clone

## 2.5.17 Class sequentialParentsChooser

### Details

This parents chooser chooses two parents sequentially. The parents are chosen from their respective sex groups. Selection is not considered.

### Initialization

SimuPOP::sequentialParentsChooser::sequentialParentsChooser

```
sequentialParentsChooser()
```

### Member Functions

**x.clone()** SimuPOP::sequentialParentsChooser::clone

## 2.5.18 Class randomParentChooser

### Details

This parent chooser chooses a parent randomly from the parental generation. If selection is turned on, parents are chosen with probabilities that are proportional to their fitness values. Sex is not considered. Parameter `replacement` determines if a parent can be chosen multiple times. In case that `replacement=false`, parameter `replenish=true` allows restart of the process if all parents are exhausted. Note that selection is not allowed when `replacement=false` because this poses a particular order on individuals in the offspring generation.

### Initialization

SimuPOP::randomParentChooser::randomParentChooser

```
randomParentChooser(replacement=True, replenish=False)
```

**replacement** If replacement is false, a parent can not be chosen more than once.

**replenish** If all parent has been chosen, choose from the whole parental population again.

### Member Functions

**x.clone()** SimuPOP::randomParentChooser::clone

## 2.5.19 Class randomParentsChooser

### Details

This parent chooser chooses two parents randomly, a male and a female, from their respective sex groups randomly. If selection is turned on, parents are chosen from their sex groups with probabilities that are proportional to their fitness

values. If parameter `replacement` is false, a chosen pair of parents can no longer be selected. This feature can be used to simulate monopoly. If `replenish` is true, a sex group can be replenished when it is exhausted. Note that selection is not allowed in the case of monopoly because this poses a particular order on individuals in the offspring generation. This parents chooser also allows polygamous mating by reusing a parent multiple times when returning parents, and allows specification of a few alpha individuals who will be the only mating individuals in their sex group.

### Initialization

```
randomParentsChooser(replacement=True, replenish=False, polySex=Male,
polyNum=1, alphaSex=Male, alphaNum=0, alphaField=string)
```

Note: If selection is enabled, it works regularly on on-alpha sex, but works twice on alpha sex. That is to say, `alphaNum` alpha individuals are chosen selectively, and selected again during mating.

**alphaField** If an information field is given, individuals with non-zero values at this information field are alpha individuals. Note that these individuals must have `alphaSex`.

**alphaNum** Number of alpha individuals. If `infoField` is not given, `alphaNum` random individuals with `alphaSex` will be chosen. If selection is enabled, individuals with higher fitness values have higher probability to be selected. There is by default no alpha individual (`alphaNum = 0`).

**alphaSex** The sex of the alpha individual, i.e. alpha male or alpha female who be the only mating individuals in their sex group.

**polyNum** Number of sex partners.

**polySex** Male (polygyny) or Female (polyandry) parent that will have `polyNum` sex partners.

**replacement** Choose with (`True`, default) or without (`False`) replacement. When choosing without replacement, parents will be paired and can only mate once.

**replenish** If set to true, one or both sex groups will be replenished if they are exhausted.

### Member Functions

**x.clone()** `SimuPOP::randomParentsChooser::clone`

## 2.5.20 Class `infoParentsChooser`

### Details

This parents chooser choose an individual randomly, but choose his/her spouse from a given set of information fields, which stores indexes of individuals in the same generation. A field will be ignored if its value is negative, or if sex is compatible. Depending on what indexes are stored in these information fields, this parent chooser can be used to implement consanguineous mating where close relatives are located for each individual, or certain non-random mating schemes where each individual can only mate with a small number of pre-determinable individuals. This parent chooser (currently) uses `randomParentChooser` to choose one parent and randomly choose another one from the information fields. Because of potentially non-even distribution of valid information fields, the overall process may not be as random as expected, especially when selection is applied. Note: if there is no valid individual, this parents chooser works like a double parentChooser.

### Initialization

`SimuPOP::infoParentsChooser::infoParentsChooser`

```
infoParentsChooser(infoFields=[], replacement=True, replenish=False)
```

**infoFields** Information fields that store index of matable individuals.

**replacement** If replacement is false, a parent can not be chosen more than once.

**replenish** If all parent has been chosen, choose from the whole parental population again.

## Member Functions

**x.clone()** SimuPOP::infoParentsChooser::clone

## 2.5.21 Class pedigreeParentsChooser

### Details

This parents chooser chooses one or two parents from a given pedigree. It works even when only one parent is needed.

### Initialization

SimuPOP::pedigreeParentsChooser::pedigreeParentsChooser

```
pedigreeParentsChooser (ped)
```

## Member Functions

**x.clone()** SimuPOP::pedigreeParentsChooser::clone

**x.subPopSizes (gen)** SimuPOP::pedigreeParentsChooser::subPopSizes

## 2.5.22 Class pyParentsChooser

### Details

This parents chooser accept a Python generator function that yields repeatedly an index (relative to each subpopulation) of a parent, or indexes of two parents as a Python list of tuple. The generator function is responsible for handling sex or selection if needed.

### Initialization

SimuPOP::pyParentsChooser::pyParentsChooser

```
pyParentsChooser (parentsGenerator)
```

**parentsGenerator** A Python generator function

## Member Functions

**x.clone()** SimuPOP::pyParentsChooser::clone

**x.finalize (pop, sp)** SimuPOP::pyParentsChooser::finalize

### 2.5.23 Class `cloneOffspringGenerator`

#### Details

Clone offspring generator copies parental genotype to a number of offspring. Only one parent is accepted. The number of offspring produced is controlled by parameters `numOffspring`, `numOffspringFunc`, `maxNumOffspring` and `mode`. Parameters `sexParam` and `sexMode` is ignored.

#### Initialization

`SimuPOP::cloneOffspringGenerator::cloneOffspringGenerator`

```
cloneOffspringGenerator(numOffspring=1, numOffspringFunc=None,
maxNumOffspring=1, mode=MATE_NumOffspring, sexParam=0.5,
sexMode=MATE_RandomSex)
```

**sexMode** Ignored because sex is copied from the parent.

**sexParam** Ignored because sex is copied from the parent.

#### Member Functions

**x.clone()** `SimuPOP::cloneOffspringGenerator::clone`

### 2.5.24 Class `selfingOffspringGenerator`

#### Details

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 diploid 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.

#### Initialization

`SimuPOP::selfingOffspringGenerator::selfingOffspringGenerator`

```
selfingOffspringGenerator(numOffspring=1, numOffspringFunc=None,
maxNumOffspring=1, mode=MATE_NumOffspring, sexParam=0.5,
sexMode=MATE_RandomSex)
```

#### Member Functions

**x.clone()** `SimuPOP::selfingOffspringGenerator::clone`

### 2.5.25 Class `haplodiploidOffspringGenerator`

#### Details

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 considered 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.

## Initialization

`SimuPOP::haplodiploidOffspringGenerator::haplodiploidOffspringGenerator`

```
haplodiploidOffspringGenerator(numOffspring=1, numOffspringFunc=None,  
maxNumOffspring=1, mode=MATE_NumOffspring, sexParam=0.5,  
sexMode=MATE_RandomSex)
```

## Member Functions

**x.clone()** `SimuPOP::haplodiploidOffspringGenerator::clone`

**x.copyParentalGenotype(parent, it, ploidy, count)** `SimuPOP::haplodiploidOffspringGenerator::copyParentalGe`

## 2.5.26 Class mendelianOffspringGenerator

### Details

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. This offspring generator only works for diploid populations.

### Initialization

`SimuPOP::mendelianOffspringGenerator::mendelianOffspringGenerator`

```
mendelianOffspringGenerator(numOffspring=1, numOffspringFunc=None,  
maxNumOffspring=1, mode=MATE_NumOffspring, sexParam=0.5,  
sexMode=MATE_RandomSex)
```

## Member Functions

**x.clone()** `SimuPOP::mendelianOffspringGenerator::clone`

## 2.5.27 Determine the number of offspring during mating

Parameters `numOffspring`, `maxNumOffspring`, `numOffspringFunc` and `mode` are provided for each mating scheme (each offspring generator, to be exact) to determine the number of offspring produced at each mating event.

The default value of `numOffspring` parameter makes a mating scheme produces one offspring per mating event. This is required by random mating schemes and should be used whenever possible. However, various situations require a larger family size or even changing family sizes. `simuPOP` provides a comprehensive way to deal with this problem.

As described in the class reference, the method to determine the number of offspring is to set the `mode` parameter:

- **MATE\_NumOffspring:** Produce `numOffspring` offspring all the time.
- **MATE\_PyNumOffspring:** When `numOffspringFunc` is defined, this mode is automatically used. A user provided function is called whenever a mating event happens. The return value determines the number of offspring to use.

- **MATE\_GeometricDistribution:** `numOffspring` is considered as  $p$  for a geometric distribution. The number of offspring for each mating is determined by

$$P(k) = p(1-p)^{k-1} \text{ for } k \geq 1$$

- **MATE\_PoissonDistribution:** `numOffspring` is considered as  $p$  for a Poisson distribution. The number of offspring for each mating is determined by

$$P(k) = \frac{p^{k-1}}{(k-1)!} e^{-p} \text{ for } k \geq 1$$

Since the mean of this shifted Poisson distribution is  $p + 1$ , you need to specify, for example, 2, if you want a mean family size 3.

- **MATE\_BinomialDistribution:** `numOffspring` is considered as  $p$  for a Binomial distribution. Let  $N = \text{maxNumOffspring}$ , the number of offspring for each mating is determined by

$$P(k) = \frac{(n-1)!}{(k-1)!(n-k)!} p^{k-1} (1-p)^{n-k} \text{ for } N \geq k \geq 1$$

- **MATE\_UniformDistribution:** `numOffspring` is be considered as  $a, b$  for a Uniform distribution, respectively. The number of offspring for each mating is determined by

$$P(k) = \frac{1}{b-a} \text{ for } b \geq k \geq a$$

Note that all these distributions are adjusted to produce at least one offspring.

## 2.5.28 Determine offspring sex

When the last chromosome is a sex chromosome (`sexChrom=True`), offspring sex is determined by his/her genotype. If an offspring is cloned from his/her parent using a `cloneOffspringGenerator()`, offspring sex is the same as his/her parent. Otherwise, offspring is by default assigned to Male and Female with equal probability 0.5.

More advanced sex assignment mode is determined by parameters `sexMode` and `sexParam` of a mating scheme or an offspring generator (see later section). `sexMode` can be

- **MATE\_RandomSex** This is the default mode where offspring can be Male or Female with equal probability.
- **MATE\_ProbOfMale** In this mode, parameter `sexParam` is considered as the probability of a Male offspring.
- **MATE\_NumOfMale** In this mode, parameter `sexParam` is the number of male in the family. If the number of offspring at a mating event is less than this number, all offspring will be male.
- **MATE\_NumOfFemale** Similar to **MATE\_NumOfMale** but parameter `sexParam` is considered as the number of female in the family.

`MATE_NumOfMale` and `MATE_NumOfFemale` are useful in theoretical studies where the sex ratio of a population needs to be controlled strictly, or in special mating schemes, usually for animal populations, where only a certain number of male or female individuals are allowed in a family.



## 2.5.29 Determine subpopulation sizes of the next generation

The default behavior of `simuPOP` is to use the same population/subpopulation sizes as those of the parent generation. You can change this behavior by setting one of `newSubPopSize`, `newSubPopSizeExpr`, and `newSubPopSizeFunc` parameters:

- If you would like to have fixed subpopulation sizes, use `newSubPopSize=some_fixed_values`. This is useful when subpopulation sizes are changed by migration and you do want to keep constant subpopulation sizes.
- If subpopulation sizes can be easily calculated through an expression, you can use `newSubPopSizeExpr` to determine the new subpopulation sizes. For example, `newSubPopSizeExpr='[gen+10]'` uses the generation number + 10 as the new population size. More complicated expressions can be used, maybe along with `pyExec` operators, but in these cases, a specialized function and `newSubPopSizeFunc` are recommended.
- A more organized (and thus recommended) way to set new population/subpopulation sizes is through parameter `newSubPopSizeFunc`. To use this parameter, you need to define a Python function that takes two parameters: the generation number and the current subpopulation sizes, and return an array of new subpopulation sizes (return `[newsize]` instead of `newsize` when you do not have any subpopulation structure). The example of `class Mating` demonstrates the use of this parameter.

## 2.5.30 Demographic change functions

`newSubPopSizeFunc` can take a function with parameters `gen` and `oldSize`. A few functions are defined in `simuUtil.py` that will return such a function with given parameters. All these functions support a burnin stage and then split to equal sized subpopulations. For all these functions, you can test them by

```
func = oneOfTheDemographicFunc(parameters)
gen = range(0, yourEndGen)
r.plot(gen, [func(x)[0] for x in gen])
```

`numSubPop` is default to 1. `split` is default to 0 or given `burnin` value. Population size change happens **after** burnin (start at `burnin+1`) and split happens at `split`.

```
ConstSize(size, split, numSubPop, bottleneckGen, bottleneckSize)
```

The population size is constant, but will split into `numSubPop` subpopulations at generation `split`. If `bottleneckGen` is specified, population size will be `bottleneckSize` at that generation.

```
LinearExpansion(initSize, endSize, end, burnin, split, numSubPop,
    bottleneckGen, bottleneckSize)
```

Linearly expand the population size from `initSize` to `endSize` after `burnin`, split the population at generation `split`. If `bottleneckGen` is specified, population size will be `bottleneckSize` at that generation.

```
ExponentialExpansion(initSize, endSize, end, burnin, split,
    numSubPop, bottleneckGen, bottleneckSize)
```

Exponentially expand the population size from `initSize` to `endSize` after `burnin`, split the population at generation `split`. If `bottleneckGen` is specified, population size will be `bottleneckSize` at that generation.

```
InstantExpansion(initSize, endSize, end, burnin, split,
    numSubPop, bottleneckGen, bottleneckSize)
```

Instantaneously expand the population size from `initSize` to `endSize` after `burnin`, split the population at generation `split`. If `bottleneckGen` is specified, population size will be `bottleneckSize` at that generation.

### 2.5.31 Sex chromosomes

Currently, only `randomMating()` in diploid population supports sex chromosomes. When `sexChrom()` is `False`, the sex of an offspring is determined randomly with probability 1/2. Otherwise, it is determined by the existence of Y chromosome, I.e., what kind of sex chromosome an offspring get from his father.

Recombinations on sex chromosomes of females (XX) are just like those on autosomes. However, this is not true in males. Currently, recombinations between male sex chromosomes (XY) are *not* allowed (a bug/feature of recombinators). This may change later if exchanges of genes between pseudoautosomal regions of XY need to be modeled.

### 2.5.32 Parent choosers and offspring generators

To implement more complex mating schemes, some concepts need to be understood. The first one is *parent chooser*. Parent chooser determines how parent or parents are chosen from a given subpopulation. There are several predefined parent choosers such as `linearParentChooser`, `randomParentChooser`, `randomParentsChooser`, and the most powerful one is called `pyParentsChooser`.

A `pyParentsChooser` accepts a Python generator function, instead of a normal python function. When this generator function is called, it returns a *generator* object that provides an iterator interface. Each time when the `next()` member function of this object is called, this function resumes where it was stopped last time, executes and returns what the next `yield` statement returns. An example of generator is given in `simuPOP` user's guide.

Example 2.7: A generator function that mimicks random mating

```
>>> from random import randint
>>>
>>> def randomChooser(pop, sp):
...     males = [x for x in range(pop.subPopSize(sp)) \
...               if pop.individual(x, sp).sex() == Male \
...               and pop.individual(x, sp).info('age') > 30]
...     females = [x for x in range(pop.subPopSize(sp)) \
...                 if pop.individual(x, sp).sex() == Female \
...                 and pop.individual(x, sp).info('age') > 30]
...     nm = len(males)
...     nf = len(females)
...     while True:
...         yield males[randint(0, nm-1)], females[randint(0, nf-1)]
...
>>> pop = population(size=[1000, 200], loci=[1], infoFields=['age'])
>>> # this will initialize sex randomly
>>> InitByFreq(pop, [0.2, 0.8])
>>> for ind in pop.individuals():
...     ind.setInfo(randint(0, 60), 'age')
...
>>> rc1 = randomChooser(pop, 0)
>>> for i in range(5):
...     print rc1.next(),
...
(190, 819) (507, 233) (733, 517) (734, 871) (406, 870)
>>> rc2 = randomChooser(pop, 1)
>>> for i in range(5):
...     print rc2.next(),
...
(184, 81) (48, 30) (32, 139) (68, 30) (68, 105)
>>>
```

A user defined parents chooser can be very complicated, involving user defined information such as geometric locations. An example is given in `scripts/demoNonRandomMating.py`. In example 2.7, the parents chooser `randomChooser` collects indexes of males and females that are over the age of 30 and return a pair of random male and female repeatedly. That is to say, individuals with age < 30 is not involved in mating. Of course, to completely implement age-dependent mating, other factors need to be considered. For example, a `pyTagger` is likely to be used to assign age to offspring.

A parents chooser can yield a pair of parents, or a single parent. Obviously, a single diploid parent can not produce offspring using the usual Medelian fashion, so here comes another concept: *offspring generator*, which determines how to produce offspring from given parent or parents. Currently, there are three standard offspring generators.

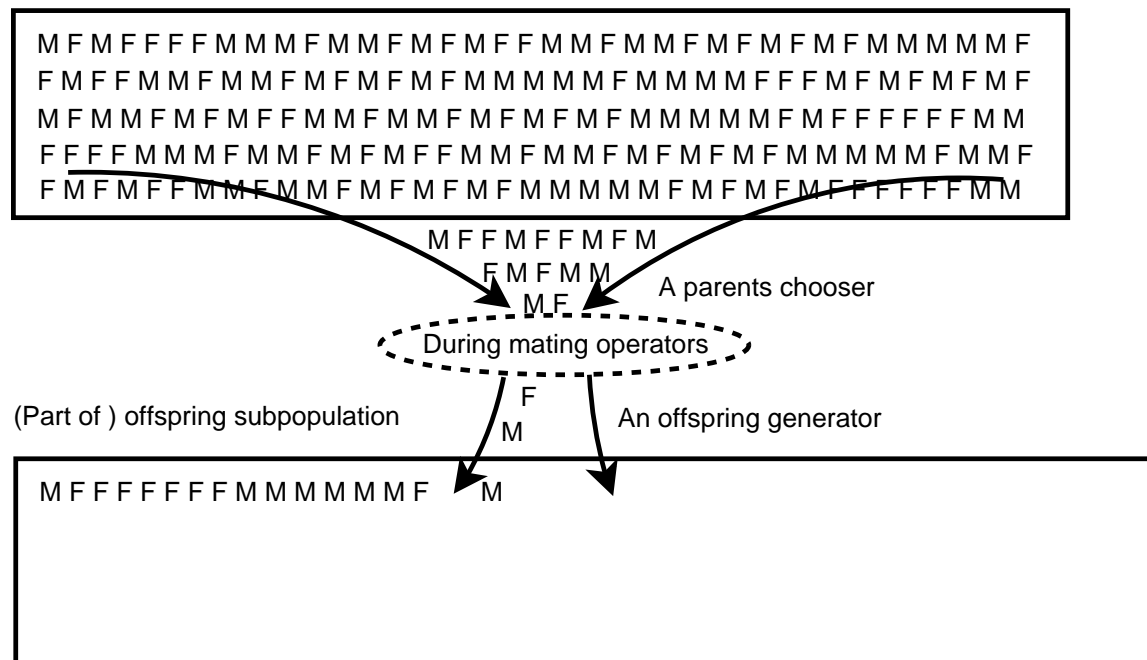
These offspring generator defines only the default way to fill offspring genotype. When a during-mating operator is involved, it may override what an offspring generator does. For example, a `recombinator` recombines parental chromosomes to fill offspring genotype. In the diploid case, it will behave the same for `cloneOffspringGenerator` and `selfingOffspringGenerator`.

### 2.5.33 Homogeneous and hybrid mating schemes

Parent choosers and offspring generators can be combined to form homogeneous mating schemes, which work identically on all (virtual) subpopulations it is applied. The only limit is that they have to be compatible in that a parent chooser that choose one parent can not be used with an offspring generator that needs two parents. A homogenous mating scheme is illustrated in Figure

Figure 2.1: A homogeneous mating scheme

Parental (virtual) subpopulation



A homogeneous mating scheme is responsible to choose parent(s) from a subpopulation or a virtual subpopulation, and population part or all of the corresponding offspring subpopulation. A parent chooser is used to choose one or two parents from the parental generation, and pass it to an offspring generator, which produces one or more offspring. During mating operators such as taggers and recombinator can be applied when offspring is generated.

The basic usage of a `pyMating` operator is as follows

```
pyMating(randomParentChooser(),
         selfingOffspringGenerator(numOffspring=2))
```

or

```
pyMating(linearParentChooser(),
         cloneOffspringGenerator())
```

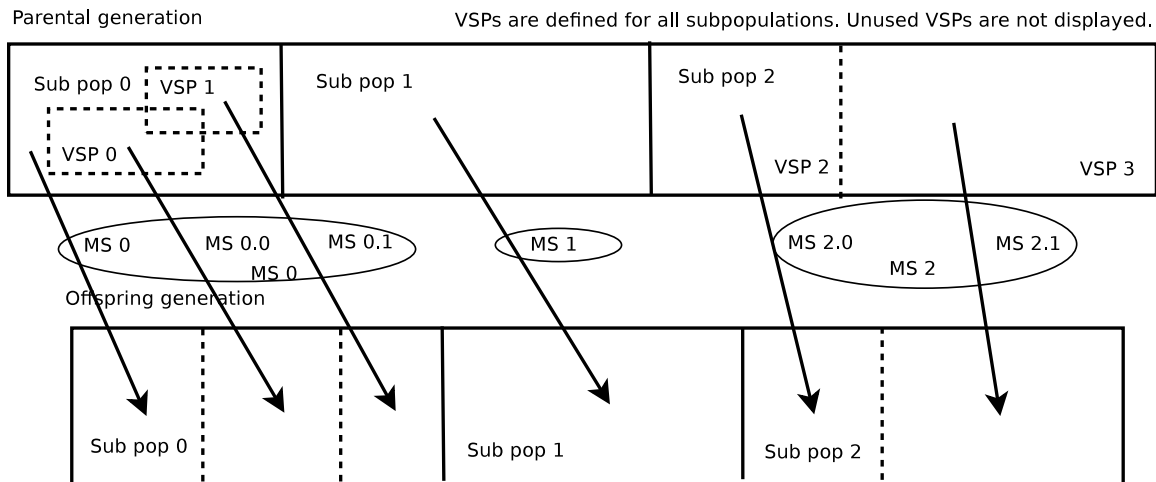
The later simply copy everyone from the parental to the offspring generation.

### 2.5.34 Heterogeneous mating schemes

Non-random mating can also be introduced by mating individuals from different groups differently. Different subpopulations, or different virtual subpopulations, can have varying fecundity, represented by different numbers of offspring generated per mating event. For example, it is possible that only adults (may be defined by age > 30 and age < 50) in a subpopulation can produce offspring, where other individual will either be copied to the offspring generation or die. It is also quite common in plant genetics that a certain portion of trees go through selfing, while others go through random mating.

A `heteroMating` mating scheme accepts a list of mating schemes that works separately on different subpopulation, or virtual subpopulations. In this way, many homogenous mating schemes can be applied to different (virtual) subpopulations. This is illustrated in Figure 2.2.

Figure 2.2: Illustration of a heteogeneous mating scheme



A heterogeneous mating scheme that applies homogenous mating schemes MS0, MS0.0, MS0.1, MS1, MS2.0 and MS2.1 to subpopulation 0, the first and second virtual subpopulation in subpopulation 0, subpopulation 1, the first and second virtual subpopulation in subpopulation 2, respectively. Note that VSP 0 and 1 in subpopulation 0 overlap, and do not add up to subpopulation 0.

For example,

```
heteroMating([randomMating(numOffspring=2, subPop=0),
             randomMating(numOffspring=4, subPop=1)])
```

define a heterogeneous mating scheme that mating events in subpopulation 0 produces two offspring, while producing four in subpopulation 1.

```
pop.setVirtualSplitter(proportionSplitter([0.2, 0.8]), 0)
heteroMating([selfMating(numOffspring=2, subPop=0, virtualSubPop=0),
```

```

        randomMating(subPop=0, virtualSubPop=1)],
        shuffleOffspring=True
    )

```

allows different mating schemes in one subpopulation. In this example, the first subpopulation is splitted into two virtual subpopulations by proportion. Then, a selfing mating scheme is applied to the first virtual subpopulation, and a random mating scheme is applied to the second. In case that there are more than one mating schemes working on the same subpopulation, offspring are shuffled randomly by default, unless this is turned off by `shuffleOffspring=False`. Randomization of the order of offspring is usually desired because otherwise, taking this example, the first 20% of individuals will always go through selfing, and the rest will always go through random mating. When offspring are shuffled, each individual will have probability 0.2 to be selfing, and probability 0.8 to mate randomly.

simuPOP determines if a mating scheme will be applied to a particular subpopulation using the following rules

- If neither `subPop`, nor `virtualSubPop` is specified, the mating scheme is applied to all subpoulations (as a whole, not any virtual subpopulation).
- If `subPop`, but not `virtualSubPop` is specified, the mating scheme is applied to the specified subpopulation (as a whole).
- If `subPop` and `virtualSubPop` are both specified, the mating scheme is applied to the specified virtual subpopulation.
- If `subPop` is not specified, but `virtualSubPop` is, the mating scheme is applied to spcified virtual subpopulation of all subpopulations. Note that simuPOP will report an error if a subpopulation does not define such a virtual subpopulation.

If one mating scheme is specified for each parental subpopulation, offspring subpopulation sizes are determined as usual, through parameters `newSubPopSize`, `newSubPopSizeFunc`, etc. However, if multiple mating schemes will be applied to the same subpopulation, they have to share the same offspring subpopulation. This problem is addressed by a weight system. That is to say, each mating scheme can be given a weight using parameter `weight`. A weight can be positive, zero (default) or negative. The number of offspring each mating scheme will produce is determined by these weights.

This weighting scheme is best explained by an example. Assuming that there are three mating schemes working on the same parental subpopulation

- Mating scheme A works on the whole subpopulation of size 1000
- Mating scheme B works on a virtual subpopulation of size 500
- Mating scheme C works on another virtual subpopulation of size 800

Assuming the corresponding offspring subpopulation has  $N$  individuals,

- If all weights are 0, the offspring subpopulation is divided in proportion to parental (virtual) subpopulation sizes. In this example, the mating schemes will produce  $\frac{10}{23}N$ ,  $\frac{5}{23}N$ ,  $\frac{8}{23}N$  individuals respectively.
- If all weights are negative, they are multiplied to their parental (virtual) subpopulation sizes to get a fixed size. For example, weight (-1, -2, -0.5) will lead to sizes (1000, 1000, 400) in the offspring subpopulation. If  $N \neq 2400$  in this case, an error will be raised.
- If all weights are positive, the number of offspring produced from each mating scheme is proportional to these weights. For example, weights (1, 2, 3) will lead to  $\frac{1}{6}N$ ,  $\frac{2}{6}N$ ,  $\frac{3}{6}N$  individuals respectively. In this case, 0 weights will produce no offspring.
- If there are mixed positive and negative weights, the negative weights are first processed, and the rest of the individuals are divided using positive weights. For example, three mating schemes with weights (-1, 2, 3) will produce 1000,  $\frac{2}{5}(N - 1000)$ ,  $\frac{3}{5}(N - 1000)$  individuals respectively.

## 2.6 Operators

### 2.6.1 Class `baseOperator`

Base class of all classes that manipulate populations

#### Details

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.

There are three kinds of operators:

- **built-in:** written in C++, the fastest. They do not interact with Python shell except that some of them set variables that are accessible from Python.
- **hybrid:** written in C++ but calls a Python function during execution. Less efficient. For example, a hybrid `mutatorpyMutator` will go through a population and mutate alleles with given mutation rate. How exactly the allele will be mutated is determined by a user-provided Python function. More specifically, this operator will pass the current allele to a user-provided Python function and take its return value as the mutant allele.
- **pure Python:** written in Python. The same speed as Python. For example, a `varPlotter` can plot Python variables that are set by other operators. Usually, an individual or a population object is passed to a user-provided Python function. Because arbitrary operations can be performed on the passed object, this operator is very flexible.

Operators can be applied at different stages of the life cycle of a generation. It is possible for an operator to apply multiple times in a life cycle. For example, a `savePopulation` operator might be applied before and after mating to trace parental information. More specifically, operators can be applied at *pre-*, *during-*, *post-mating*, or a combination of these stages. Applicable stages are usually set by default but you can change it by setting `stage=(PreMating|PostMating|DuringMating|PrePostMating|PreDuringMating|DuringPostMating)` parameter. Some operators ignore `stage` parameter because they only work at one stage.

Operators do not have to be applied at all generations. You can specify starting and/or ending generations (parameters `start`, `end`), gaps between applicable generations (parameter `step`), or specific generations (parameter `at`). For example, you might want to start applying migrations after certain burn-in generations, or calculate certain statistics only sparsely. Generation numbers can be counted from the last generation, using negative generation numbers.

Most operators are applied to every replicate of a simulator during evolution. However, you can apply operators to one (parameter `rep`) or a group of replicates only (parameter `grp`). For example, you can initialize different replicates with different initial values and then start evolution. c.f. `simulator::setGroup`. Operators can have outputs, which can be standard (terminal) or a file. Output can vary with replicates and/or generations, and outputs from different operators can be accumulated to the same file to form table-like outputs.

Filenames can have the following format:

- `'filename'` this file will be overwritten each time. If two operators output to the same file, only the last one will succeed;
- `'>filename'` the same as `'filename'`;
- `'>>filename'` the file will be created at the beginning of evolution (`simulator::evolve`) and closed at the end. Outputs from several operators are appended;
- `'>>>filename'` the same as `'>>filename'` except that the file will not be cleared at the beginning of evolution if it is not empty;

- `'>'` standard output (terminal);
- `"` suppress output.

The output filename does not have to be fixed. If parameter `outputExpr` is used (parameter `output` will be ignored), it will be evaluated when a filename is needed. This is useful when you need to write different files for different replicates/generations.

### Initialization

Common interface for all operators (this base operator does nothing by itself.)

```
baseOperator(output, outputExpr, stage, begin, end, step, at, rep,
             grp, infoFields)
```

**at** An array of active generations. If given, `stage`, `begin`, `end`, and `step` will be ignored.

**begin** The starting generation. Default to 0. A negative number is allowed.

**end** Stop applying after this generation. A negative numbers is allowed.

**grp** Applicable group. Default to `GRP_ALL`. A group number for each replicate is set by `simulator.__init__` or `simulator::setGroup()`.

**output** A string of the output filename. Different operators will have different default `output` (most commonly `'>'` or `"`).

**outputExpr** An expression that determines the output filename dynamically. This expression will be evaluated against a population's local namespace each time when an output filename is required. For example, `">>>out%s_%s.xml' % (gen, rep)"` will output to `>>>out1_1.xml` for replicate 1 at generation 1.

**rep** Applicable replicates. It can be a valid replicate number, `REP_ALL` (all replicates, default), or `REP_LAST` (only the last replicate). `REP_LAST` is useful in adding newlines to a table output.

**step** The number of generations between active generations. Default to 1.

### Note

- Negative generation numbers are allowed for parameters `begin`, `end` and `at`. They are interpreted as `endGen + gen + 1`. For example, `begin = -2` in `simu.evolve(..., end=20)` starts at generation 19.
- `REP_ALL`, `REP_LAST`, `GRP_ALL` are special constant that can only be used in the constructor of an operator. That is to say, explicit test of `rep() == REP_LAST` will not work.

### Member Functions

**x.apply(pop)** Apply to one population. It does not check if the operator is activated.

**x.clone()** Deep copy of an operator

**x.diploidOnly()** Determine if the operator can be applied only for diploid population

**x.haploidOnly()** Determine if the operator can be applied only for haploid population

**x.infoField(idx)** Get the information field specified by user (or by default)

**x.infoSize()** Get the length of information fields for this operator

## 2.7 Simulator

### 2.7.1 Class `simulator`

Simulator manages several replicates of a population, evolve them using given mating scheme and operators

#### Details

Simulators combine three important components of `simuPOP`: population, mating scheme and operator together. A simulator is created with an instance of `population`, a replicate number `rep` and a mating scheme. It makes `rep` number of replicates of this population and control the evolutionary process of them.

The most important function of a simulator is `evolve()`. It accepts an array of operators as its parameters, among which, `preOps` and `postOps` will be applied to the populations at the beginning and the end of evolution, respectively, whereas `ops` will be applied at every generation.

A simulator separates operators into *pre-*, *during-*, and *post-mating* operators. During evolution, a simulator first apply all pre-mating operators and then call the `mate()` function of the given mating scheme, which will call during-mating operators during the birth of each offspring. After mating is completed, post-mating operators are applied to the offspring in the order at which they appear in the operator list.

Simulators can evolve a given number of generations (the `end` parameter of `evolve`), or evolve indefinitely until a certain type of operators called terminator terminates it. In this case, one or more terminators will check the status of evolution and determine if the simulation should be stopped. An obvious example of such a terminator is a fixation-checker.

A simulator can be saved to a file in the format of `'txt'`, `'bin'`, or `'xml'`. This allows you to stop a simulator and resume it at another time or on another machine.

#### Initialization

Create a simulator

```
simulator(pop, matingScheme, stopIfOneRepStops=False,
          applyOpToStoppedReps=False, rep=1, grp=[])
```

**applyOpToStoppedReps** If set, the simulator will continue to apply operators to all stopped replicates until all replicates are marked 'stopped'.

**grp** Group number for each replicate. Operators can be applied to a group of replicates using its `grp` parameter.

**matingScheme** A mating scheme

**population** A population created by `population()` function. This population will be copied `rep` times to the simulator. Its content will not be changed.

**rep** Number of replicates. Default to 1.

**stopIfOneRepStops** If set, the simulator will stop evolution if one replicate stops.

#### Member Functions

**`x.addInfoField(field, init=0)`** Add an information field to all replicates

Add an information field to all replicate, and to the simulator itself. This is important because all populations must have the same genotypic information as the simulator. Adding an information field to one or more of the replicates will compromise the integrity of the simulator.

**field** information field to be added



**x.addInfoFields(fields, init=0)** Add information fields to all replicates

Add given information fields to all replicate, and to the simulator itself.

**x.clone()** Deep copy of a simulator

**x.evolve(ops, preOps=[], postOps=[], end=-1, gen=-1, dryrun=False)** Evolve all replicates of the population, subject to operators

Evolve to the `end` generation unless `end=-1`. An operator ( terminator) may stop the evolution earlier.

`ops` will be applied to each replicate of the population in the order of:

- all pre-mating operators
- during-mating operators called by the mating scheme at the birth of each offspring
- all post-mating operators If any pre- or post-mating operator fails to apply, that replicate will be stopped. The behavior of the simulator will be determined by flags `applyOpToStoppedReps` and `stopIfOneRepStopss`.

**dryrun** dryrun mode. Default to `False`.

**gen** generations to evolve. Default to `-1`. In this case, there is no ending generation and a simulator will only be ended by a terminator. Note that `simu.gen()` refers to the beginning of a generation, and starts at 0.

**ops** operators that will be applied at each generation, if they are active at that generation. (Determined by the `begin`, `end`, `step` and `at` parameters of the operator.)

**postOps** operators that will be applied after evolution. `evolve()` function will *not* check if they are active.

**preOps** operators that will be applied before evolution. `evolve()` function will *not* check if they are active.

**Note:** When `gen = -1`, you can not specify negative generation parameters to operators. How would an operator know which generation is the -1 generation if no ending generation is given?

**x.gen()** Return the current generation number

**x.getPopulation(rep, destructive=False)** Return a copy of population `rep`

By default return a cloned copy of population `rep` of the simulator. If `destructive==True`, the population is extracted from the simulator, leaving a defunct simulator.

**destructive** if true, destroy the copy of population within this simulator. Default to false. `getPopulation(rep, true)` is a more efficient way to get hold of a population when the simulator will no longer be used.

**rep** the index number of the replicate which will be obtained

**x.group()** Return group indexes

**x.numRep()** Return the number of replicates

**x.population(rep)** Return a reference to the `rep` replicate of this simulator.

**rep** the index number of replicate which will be accessed

**Note:** The returned reference is temporary in the sense that the referred population will be invalid after another round of evolution. If you would like to get a persistent population, please use `getPopulation(rep)`.

**x.saveSimulator(filename, format="auto", compress=True)** Save simulator in `'txt'`, `'bin'` or `'xml'` format

**compress** whether or not compress the file in `'gzip'` format

**filename** filename to save the simulator. Default to `simu`.

**format** format to save. Default to `auto`. I.e., determine the format by file extensions.

**x.setAncestralDepth(depth)** Set ancestral depth of all replicates

**x.setGen(gen)** Set the current generation. Usually used to reset a simulator.  
     **gen** new generation index number

**x.setGroup(grp)** Set groups for replicates

**x.setMatingScheme(matingScheme)** Set a new mating scheme

**x.step(ops=[], preOps=[], postOps=[], steps=1, dryrun=False)** Evolve *steps* generation

**x.vars(rep, subPop=-1)** Return the local namespace of *populationrep*, equivalent to `x.population(rep).vars(subPop)`.

## 2.7.2 Generation number

Several aspects of the generation number may cause confusion:

- generation starts from zero
- a generation number presents a 'to-be-evolved' generation
- the ending generation specified in `evolve()` will be executed

That is to say, a new simulator will have generation 0 (at the beginning of generation 0). If you do `evolve(..., end=0)`, `evolve` will evolve one generation and stop at the beginning of generation 1.

It may sound strange that

```
evolve(end=2)
```

evolve the population 3 generations. Generation 0, generation 1, and generation 2. When you use `start=0, step=5, end=10` for your operator, it will be applied at generations 0, 5, 10 etc. At the end of the simulation, current generation number is 3! (If you are familiar with C, this is like a `for` loop index). This is why you should test if a simulation is finished correctly by

```
if(simu.gen() == endGen+1)
```

instead of `simu.gen() == endGen`. (`endGen` is the value for parameter `end`).

## 2.7.3 Operator calling sequence

In a simulation, operators are applied at different stages, pre-, during-, and post-mating (controlled by `stage` parameter), at specified generations (controlled by `begin`, `end`, `step`, `at` parameters), and to specified replicates (controlled by `rep`, `grp` parameters). The order of applying operators usually does not matter but errors may occur if you are not careful. For example, `stat(...)` calculates the statistics of the current population. It is a pre-mating operator so you should set `stage=PostMating` and put it after all operators if you would like to measure a post-mating population. It also should be put before any operator (such as an terminator) that uses the shared variable set by `stat(...)`.

If you are not sure about the calling sequence of operators, you can set the `dryrun` parameter of `evolve()` function to `True`. `evolve` will then print out the order of operators to apply. Consider that operators can be `PreMating`, `PostMating`, `PrePostMating`, `DuringMating` and the default value (parameter `stage`) may not be what you expect. Having a look at the calling sequence before the real evolution is always a good idea.

## 2.7.4 Save and Load

Using function `saveSimulator`, we can save a simulator to a file in the format of 'txt', 'bin', or 'xml'. However, a mating scheme can not be saved and has to be re-specified in `LoadSimulator()`.

Example 2.8: Save and load a simulator

```
>>> simu.saveSimulator("s.txt")
>>> simu.saveSimulator("s.xml", format="xml")
>>> simu.saveSimulator("s.bin", format="bin")
>>> simu1 = LoadSimulator("s.txt", randomMating())
>>> simu2 = LoadSimulator("s.xml", randomMating(), format="xml")
>>> simu3 = LoadSimulator("s.bin", randomMating(), format="bin")
>>>
```

## 2.8 Population variables

Populations are associated with Python variables. These variables are usually set by various operators but you can also set them manually. For example, `stat` operator calculates many population statistics and store the results in a population's local namespace.

### 2.8.1 `vars()` and `dvars()` functions

Conceptually, population variables are organized as follows (looking from a simulator's point of view):

<code>simu.vars(0)</code>	<code>simu.vars(1) ...</code>	<code>// replicate</code>
<code>popSize</code>	<code>popSize</code>	<code>// local namespace</code>
<code>alleleFreq[0]</code>	<code>alleleFreq[0]</code>	<code>// allele frequency at locus 1</code>
<code>alleleFreq[1]</code>	<code>alleleFreq[1]</code>	<code>// at locus 2</code>
<code>...</code>	<code>...</code>	
<code>subPop[0]</code>	<code>subPop[0]</code>	<code>// subpop namespace</code>
<code>popSize</code>	<code>popSize</code>	<code>// subpopulation 1 size</code>
<code>alleleFreq[0]</code>	<code>alleleFreq[0]</code>	<code>// allele frequency at locus 1</code>
<code>...</code>	<code>...</code>	
<code>subPop[1]</code>	<code>subPop[1]</code>	<code>// variables for subpop 2</code>
<code>...</code>	<code>...</code>	

You can refer to these variables using `population::vars()` or `population::dvars()` function. The returned values of `vars()` and `dvars()` reflect the same dictionary, but `dvars()` uses a little Python magic so that you can use attribute syntax to access dictionary keys. Because `a.alleleFreq[0]` is easier to read than `a['alleleFreq'][0]`, `dvars()` is more frequently used.

There are several ways to use these two functions

- `pop.vars()`, `pop.dvars()` return the variables of population `pop`
- `pop.vars(subPop)`, `pop.dvars(subPop)` returns dictionary `pop.vars()['subPop'][subPop]`
- `simu.vars(rep)`, `simu.dvars(rep)` return the variables of the `rep`'th population of simulator `simu`, i.e. `simu.population(rep).vars()`.
- `simu.vars(rep, subPop)`, `simu.dvars(rep, subPop)` returns dictionary  
`simu.vars(rep)['subPop'][subPop]`

Direct access to variables `pop.vars()['subPop'][subPop]` is provided because statistics calculator `stat`, by default, calculates the same set of statistics for all subpopulations (and the whole population).

To have a look at all variables defined in this dictionary, you can use function `ListVars` defined in `simuUtil.py`. With `wxPython` installed, this function opens a nice window with a tree representing the variables. Without `wxPython` (or use parameter `useWxPython=False`), variables are displayed in an indented form. Several parameters can be used to limit your display. They are

- `level`: the level of the tree, further nested variables will not be displayed
- `name`: the name of the variable to display
- `subPop`: whether or not display variables for each subpopulation.

#### Example 2.9: Population variables

```
>>> from simuUtil import ListVars
>>> pop = population(size=[1000, 2000], loci=[1])
>>> InitByFreq(pop, [0.2, 0.8])
>>> ListVars(pop.vars(), useWxPython=False)
  rep : -1
  grp : -1
>>> Stat(pop, popSize=1, alleleFreq=[0])
>>> # subPop is True by default, use name to limit the variables to display
>>> ListVars(pop.vars(), useWxPython=False, subPop=False, name='alleleFreq')
  alleleFreq :
    [0]
      [0]      0.1975
      [1]      0.8025
>>> # print number of allele 1 at locus 0
>>> print pop.vars()['alleleNum'][0][1]
4815
>>> print pop.dvars().alleleNum[0][1]
4815
>>> print pop.dvars().alleleFreq[0]
[0.19750000000000001, 0.8024999999999999]
>>> print pop.dvars(1).alleleNum[0][1]
3193
>>>
```

## 2.8.2 Local namespace, `pyEval` and `pyExec` operators

Population variables is a Python dictionary, and furthermore a *Local namespace*, which means that you can use dictionary items as variables during evaluation. To evaluate in a population's local namespace, you can use function `population::evaluate()` or `population::execute()`. For example:

#### Example 2.10: Local namespaces of populations

```
>>> pop = population(size=[1000, 2000], loci=[1])
>>> InitByFreq(pop, [0.2, 0.8])
>>> Stat(pop, popSize=1, alleleFreq=[0])
>>> print pop.evaluate('alleleNum[0][0] + alleleNum[0][1]')
6000
>>> pop.execute('newPopSize=int(popSize*1.5)')
>>> ListVars(pop.vars(), level=1, useWxPython=False)
```

```

newPopSize :    4500
grp :    -1
rep :    -1
popSize :        3000
numSubPop :      2
alleleNum :
    list of length 1
virtualPopSize :
    list of length 2
subPopSize :
    list of length 2
alleleFreq :
    list of length 1
subPop
    list of length 2
>>> # this variable is 'local' to the population and is
>>> # not available in the main namespace
>>> newPopSize
Traceback (most recent call last):
  File "refManual.py", line 1, in <module>
    #
NameError: name 'newPopSize' is not defined
>>> #
>>> simu = simulator(population(10),noMating(), rep=2)
>>> # evaluate an expression in different areas
>>> print simu.vars(1)
{'rep': 1, 'gen': 0, 'grp': 1}
>>> print simu.population(0).evaluate("grp*2")
0
>>> print simu.population(1).evaluate("grp*2")
2
>>> # a statement (no return value)
>>> simu.population(0).execute("myRep=2+rep*rep")
>>> simu.population(1).execute("myRep=2*rep")
>>> print simu.vars(0)
{'rep': 0, 'myRep': 2, 'gen': 0, 'grp': 0}
>>>

```

These two functions are rarely used, because

```
pop.evaluate('alleleNum[0][1] + 1')
```

is equivalent to

```
pop.dvar().alleleNum[0][1] + 1
```

Operators `pyEval`/`pyExec` are more useful in that they can be applied to different populations during evolution, and report statistics calculated by operator `stat` dynamically. The difference between these two operators are that `pyEval` evaluates a Python expression and returns its value, while `pyExec` executes a list of statements in the form of a multi-line string, and does not return any value.

Example 2.11: Use of operators `pyEval` and `pyExec`

```

>>> simu = simulator(population(100, loci=[1]),
...     randomMating(), rep=2)
>>> simu.evolve(

```

```

...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [ stat(alleleFreq=[0]),
...             pyExec('myNum = alleleNum[0][0] * 2'),
...             pyEval(r'"gen %d, rep %d, num %d, myNum %d\n"' \
...                   ' % (gen, rep, alleleNum[0][0], myNum)')]
...     ],
...     gen=3
... )
gen 0, rep 0, num 44, myNum 88
gen 0, rep 1, num 31, myNum 62
gen 1, rep 0, num 49, myNum 98
gen 1, rep 1, num 43, myNum 86
gen 2, rep 0, num 49, myNum 98
gen 2, rep 1, num 45, myNum 90
True
>>>

```

## 2.9 Information fields

An individuals have genotype, sex and affection status information, but other information may be needed. For example, one or more trait values may be needed to calculate quantitative traits, and one may want to keep track of all offspring of a parent. Because the need for information fields varies from simulation to simulation, simuPOP does not fix the amount of information fields, and allow users to specify these fields during the construction of populations, or add them when you need them.

Operators may require certain information fields to work properly. For example, all selectors require field `fitness` to store evaluated fitness values for each individual. `parentTagger` needs `father_idx` and `mother_idx` to store indices of the parents of each individual in the parental generation. These information fields can be added by the `infoFields` parameter of the population constructor or be added later using relevant function. If a required information field is unavailable, an error message will appear and tell you which field is needed. Some operators allow you to specify which information field(s) to use. For example, quantitative trait operator can work on specified fields so an individual can have several quantitative traits.

The information fields is usually set during population creation, using the `infoFields` option of population constructor. It can also be set or added by functions

- `pop.setInfoFields(fields, init)` set information fields of a population, removing all previous ones
- `pop.addInfoField(field, init)` add an information field to a population
- `pop.addInfoFields(fields, init)` add information fields to a population
- `simu.addInfoField(field, init)` add an information field to all populations in a simulator
- `simu.addInfoFields(fields, init)` add information fields to all populations in a simulator

When adding information fields to a simulator, information fields are added to all populations of the simulator. Note that it is illegal to add information field (or in a broader sense changing genotypic structure) to part of the populations of a simulator, because all populations in a simulator should have the same genotypic structure.

One can read/write information fields at individual level:

- `ind.info(idx)`, `ind.info(name)` return individual information field by index or name

- `ind.setInfo(value, idx), ind.setInfo(value, name)` set individual information field by index or name
- `ind.arrInfo()` returns a array of all information fields of an individual

or at the population level

- `pop.indInfo(idx), pop.indInfo(name)` return an information field (referred by index or name) of all individuals
- `pop.indInfo(idx, subPop), pop.indInfo(name, subPop)` return an information field (referred by index or name) of all individuals in a subpopulation `subPop`.
- `pop.setIndInfo(values, idx), pop.setIndInfo(values, name)` set information fields of all individuals with values in an array.

Both `idx` or `name` can be used in these functions. `name` is easier to use but `idx`, which can be obtained by `idx=pop.infoIdx(name)`, is faster.

#### Example 2.12: Use regular information field function

```
>>> pop = population(10, infoFields=['a', 'b'])
>>> aIdx = pop.infoIdx('a')
>>> bIdx = pop.infoIdx('b')
>>> for ind in pop.individuals():
...     a = ind.info(aIdx)
...     ind.setInfo(a+1, bIdx)
...
>>> print pop.indInfo(bIdx)
(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
>>>
```

Information fields can also be used to track individual information fields during evolution, using Python operators or operators `infoEval` and `infoExec`. The latter two operators can evaluate Python expressions and statements with variables being the information fields of individuals. Changes to these variables will change the corresponding information fields of an individual. For example, assuming that population `pop` has information field `a`, the following function (function form of operator `infoExec`) will increase the information field `a` of every individual in the population by 1.

```
InfoExec(pop, 'a += 1')
```

These statements are usually used to change the values of an information field, or derive an information field from other ones. However, variables from a population's local namespace can be made available, using option `usePopVars=True`.

#### Example 2.13: Use `infoExec` and `infoEval` operators

```
>>> pop = population(5, infoFields=['a', 'b'])
>>> InfoExec(pop, 'import random\na=random.randint(2,10)')
>>> InfoExec(pop, 'b=a+a*2')
>>> InfoEval(pop, r"('%0f, %0f)' % (a, b)")
(6, 18) (4, 12) (10, 30) (5, 15) (2, 6) >>>
>>> # this is wrong because 'c' is not available
```

```

>>> InfoExec(pop, 'b=c+a')
Traceback (most recent call last):
  File "<embed>", line 1, in <module>
NameError: name 'c' is not defined
Traceback (most recent call last):
  File "refManual.py", line 1, in <module>
    #
  File "/usr/lib/python2.5/site-packages/simuPOP_la.py", line 11749, in InfoExec
    infoExec(*args, **kwargs).apply(pop)
SystemError: Evaluation of statements failed
>>> # but we can also make use of population variables.
>>> pop.vars()['c'] = 6
>>> InfoExec(pop, 'b=c+a', usePopVars=True)
>>> print pop.indInfo('b')
(12.0, 10.0, 16.0, 11.0, 8.0)
>>>

```

## 2.10 Pedigree

A pedigree records the parent(s) of each individual during evolution. It can be created manually or using tagging operators `parentTagger` (tagging one parent) and `parentsTagger` (tagging both parents). The pedigree can be analyzed to study various properties of the evolutionary process, manipulated (e.g. removing individuals without offspring), and used to re-realize the evolutionary process using `pedigreeMating`.

A pedigree file has the following format:

```

p1 p2 p3 p4 ..... # sp1 sp2 sp3
p1 p2 p3 p4 ..... # sp1 sp2 sp3
...

```

Numbers before # of each line of a pedigree file are the parent(s) of individuals, starting from generation 0. If only one parent is used to produce offspring (e.g. using the `selfMating` mating scheme), `parentTagger(output, outputExpr)` records the index of the parent of each individual (p...) in the parental generation. Otherwise, `parentsTagger(output, outputExpr)` records the indexes of both parents.

The generation number and the size of subpopulations are listed after the # character. The sum of subpopulation sizes should match the individuals listed before #.

A number of auxiliary information pedigrees can be loaded after a pedigree is created. These information pedigree files does not have subpopulation and generation information (does not have character # and numbers after it). If there are  $n$  individuals at a generation, the corresponding line in an information pedigree file should have  $m * n$  numbers where  $m$  is the number of properties for each individual. Information pedigrees can be created by other tagging operators such as `pyTagger(output, outputExpr)`.

These auxiliary information will be attached to individuals in a pedigree. They will be removed if an individual is removed from the pedigree.

### 2.10.1 Class pedigree

A pedigree manipulation class.

#### Details



A pedigree has all the pedigree information that is needed to look at parent offspring relationship in a multi-generation population. Conceptually, there are  $n$  generations with the latest generation being generation 0. The number of generations (c.f. `gen()`) is the number of parental generations plus 1. Therefore, each individual can be identified by (gen, idx). Each individual can have a few properties 1. mother (c.f. `mother()`) 2. father (c.f. `father()`), optional because a pedigree can have only one sex 3. subpopulation (if subpopulation structure is given) 4. sex (c.f. `info('sex')`) 5. affection (c.f. `info('affection')`) 6. arbitrary information fields (c.f. `info()`)

## Initialization

Create a pedigree. If a filename `pedfile` is given, the pedigree will be loaded from this file.

```
pedigree(numParents=2, pedfile=string)
```

## Member Functions

**x.addGen(subPopSize)** Add a generation to the existing pedigree, with given subpopulation sizes `subPopSize`. All parental indexes and information will be set to zero for the new generation.

**x.addInfo(name, init=0)** Add an information field to the pedigree, with given initial value

**x.clone()** Make a copy of this pedigree.

**x.father(gen, idx)** Return the index of the father of individual `idx` at generation `gen` The returned index is the absolute index of father in the parental generation.

**x.father(gen, subPop, idx)** Return the index of the father of individual `idx` of subpopulation `subPop` at generation `gen` The returned index is the absolute index of father in the parental generation.

**x.gen()** Return the number of generations of this pedigree.

**x.info(gen, idx, name)** Return information `name` of individual `idx` at generation `gen`.

**x.info(gen, name)** Return information `name` of all individuals at generation `gen`.

**x.info(gen, subPop, idx, name)** Return information `name` of individual `idx` of subpopulation `subPop` at generation `gen`.

**x.load(filename)** Load pedigree from a file, the file is usually saved by `parentTagger` or `parentsTagger`. The format is described in the `simuPOP` reference manual

**x.loadInfo(filename, name)** Load information `name` from a information pedigree file and add to this pedigree Information `name` should not have existed in the pedigree. The information pedigree file `filename` is usually produced by taggers such as `sexTagger`, `affectionTagger`, `pyTagger` and `infoTagger`.

**x.loadInfo(filename, names)** Load information `names` from a information pedigree file and add to this pedigree Information `names` in `names` should not have existed in the pedigree. pedigree file `filename` is usually produced by taggers such as `sexTagger`, `affectionTagger`, `pyTagger` and `infoTagger`.

**x.markUnrelated()** Mark individuals that are unrelated to the visible individuals at the last generation (not marked by `selectIndividuals`) from the pedigree.

**x.mother(gen, idx)** Return the index of the mother of individual `idx` at generation `gen` The returned index is the absolute index of mother in the parental generation.

**x.mother(gen, subPop, idx)** Return the index of the mother of individual `idx` of subpopulation `subPop` at generation `gen` The returned index is the absolute index of mother in the parental generation.

**x.numParents()** Return the number of parents for each individual

**x.popSize(gen)** Population size at generation `gen`

**x.removeUnrelated(adjust\_index=True)** Remove individuals that are unrelated to the last generation from the pedigree indexes will be adjusted. WARNING: if adjust\_index=false, an invalid pedigree will be generated

**x.save(filename)** Write the pedigree to a file.

**x.saveInfo(filename, name)** Save auxiliary information name to an information pedigree file.

**x.saveInfo(filename, names)** Save auxiliary information names to an information pedigree file

**x.selectIndividuals(inds)** Mark individuals inds as 'unrelated' in the last generation removeUnrelated function will remove these individuals from the pdeigree.

**x.setFather(parent, gen, idx)** Set the index of the father of individualidx at generation gen.

**x.setFather(parent, gen, subPop, idx)** Set the index of the father of individualidx of subpopulation subPop at generation gen.

**x.setInfo(info, gen, idx, name)** Set information name of individualidx at generation gen.

**x.setInfo(info, gen, subPop, idx, name)** Set information name of individualidx of subpopulation subPop at generation gen.

**x.setMother(parent, gen, idx)** Set the index of the mother of individualidx at generation gen.

**x.setMother(parent, gen, subPop, idx)** Set the index of the mother of individualidx of subpopulation subPop at generation gen.

**x.subPopSize(gen, subPop)** Return the subpopulation size of subpopulation subPop of generation gen.

**x.subPopSizes(gen)** Return the subpopulation sizes of generation gen.

# Operator References

This chapter will list all functions, types and operators by category. The reference for class `baseOperator` is in section 2.6.

## 3.1 Python operators

A Python operator that works directly on `simuPOP` population or individuals.

### 3.1.1 Class `pyOperator`

A python operator that directly operate a population.

#### Details

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`).

#### Initialization

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

```
pyOperator(func, param=None, stage=PostMating, formOffGenotype=False,
passOffspringOnly=False, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

**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.

**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.

**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` or `outputExpr` is not supported. That is to say, you have to open/close/append to files explicitly in the Python function.
- 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.

## Member Functions

**x.apply(pop)** Apply the `pyOperator` operator to one population

**x.clone()** Deep copy of a `pyOperator` operator

A Python operator accepts a function and an optional parameter. When `pyOperator` is called, it will simply pass the accepted population (or parents and offspring in the case of `stage=DuringMating`) to the function. To use this operator, in case of `stage=PostMating`, you will need to

- define a function that handle a population as you wish.

```
def myOperator(pop, para):  
    'do whatever you want'  
    return True
```

If you return `False`, this operator will work like a terminator.

- use `pyOperator` in the form of

```
pyOperator(mfunc=pyOperator, param=para)
```

all parameters of an operator are supported except for `output` and `outputExpr` which are ignored for now.

This operator allows implementation of arbitrarily complicated operators,. To use this operator, you will have to know how to use population-related functions. The following example shows how to implement a dynamic mutator which mutate loci according to their allele frequencies.

### Example 3.1: Define a python operator

```
>>> def dynaMutator(pop, param):  
...     ''' this mutator mutate common loci with low mutation rate  
...     and rare loci with high mutation rate, as an attempt to
```

```

...     bring allele frequency of these loci at an equal level.'''
...     # unpack parameter
...     (cutoff, mu1, mu2) = param;
...     Stat(pop, alleleFreq=range( pop.totNumLoci() ) )
...     for i in range( pop.totNumLoci() ):
...         # 1-freq of wild type = total disease allele frequency
...         if 1-pop.dvars().alleleFreq[i][1] < cutoff:
...             KamMutate(pop, maxAllele=2, rate=mu1, loci=[i])
...         else:
...             KamMutate(pop, maxAllele=2, rate=mu2, loci=[i])
...     return True
>>>

```

### Example 3.2: Use of python operator

```

>>> pop = population(size=10000, ploidy=2, loci=[2, 3])
>>>
>>> simu = simulator(pop, randomMating())
>>>
>>> simu.evolve(
...     preOps = [
...         initByFreq( [.6, .4], loci=[0,2,4]),
...         initByFreq( [.8, .2], loci=[1,3]) ],
...     ops = [
...         pyOperator( func=dynaMutator, param=(.5, .1, 0) ),
...         stat(alleleFreq=range(5)),
...         pyEval(r' "%f\t%f\n"%(alleleFreq[0][1],alleleFreq[1][1])', step=10)
...     ],
...     end = 30
... )
Parameter end is obsolete in simulator::evolve(), please use gen instead
0.398800      0.206450
0.386350      0.201850
0.374500      0.221650
0.359600      0.210800
True
>>>

```

`pyOperator` can also be a during-mating operator. You will need to define a function

```
def Func(pop, off, dad, mom, para)
```

or

```
def shortFunc(off, para)
```

where `para` can be ignored. To use this operator, you can do

```
pyOperator(stage=DuringMating, func=Func, param=someparam, formOffGenotype=True)
```

or

```
pyOperator(stage=DuringMating, func=shortFunc, param=someparam,
formOffGenotype=False, passOffspringOnly=True)
```

If your during-mating `pyOperator` returns `False`, the individual will be discarded. Therefore, you can write a filter in this way. However, since the Python function will be called for each mating event, the cost of using such an operator is high, especially when population size is large.

An example of during-mating `pyOperator` can be found in `scripts/demoPyOperator.py`.

### 3.1.2 Class `pyIndOperator`

Individual operator

#### Details

This operator is similar to a `pyOperator` but works at the individual level. It expects a function that accepts an individual, optional genotype at certain loci, and an optional parameter. When it is applied, it passes each individual to this function. When `infoFields` is given, this function should return an array to fill these `infoFields`. Otherwise, `True` or `False` is expected. More specifically, `func` can be

- `func(ind)` when neither `loci` nor `param` is given.
- `func(ind, genotype)` when `loci` is given.
- `func(ind, param)` when `param` is given.
- `func(ind, genotype, param)` when both `loci` and `param` are given.

#### Initialization

A Pre- or PostMating Python operator that apply a function to each individual

```
pyIndOperator(func, loci=[], param=None, stage=PostMating,
formOffGenotype=False, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

**func** A Python function that accepts an individual and optional genotype and parameters.

**infoFields** If given, `func` is expected to return an array of the same length and fill these `infoFields` of an individual.

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

#### Member Functions

**`x.apply(pop)`** Apply the `pyIndOperator` operator to one population

**`x.clone()`** Deep copy of a `pyIndOperator` operator

## 3.2 Initialization

### 3.2.1 Class `initializer`

Initialize alleles at the start of a generation

#### Details

Initializers are used to initialize populations before evolution. They are set to be `PreMating` operators by default. `simuPOP` provides three initializers. One assigns alleles by random, one assigns a fixed set of genotypes, and the last one calls a user-defined function.

#### Initialization

Create an initializer. Default to be always active.

```
initializer(subPop=[], indRange=[], loci=[], atPloidy=-1,
stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

**atPloidy** Initialize which copy of chromosomes. Default to all.

**indRange** A [begin, end] pair of the range of absolute indexes of individuals, for example, ([1, 2]); or an array of [begin, end] pairs, such as ([[1, 4], [5, 6]]). This is how you can initialize individuals differently within subpopulations. Note that ranges are in the form of [a,b). I.e., range [4,6] will initialize individual 4, 5, but not 6. As a shortcut for [4,5], you can use [4] to specify one individual.

**loci** A vector of locus indexes at which initialization will be done. If empty, apply to all loci.

**locus** A shortcut to loci

**subPop** An array specifies applicable subpopulations

## Member Functions

**x.clone()** Deep copy of an initializer

### 3.2.2 Class `initSex` (Function form: `InitSex`)

#### Details

An operator to initialize individual sex. For convenience, this operator is included by other initializers such as `initByFreq`, `initByValue`, or `pyInit`.

#### Initialization

Initialize individual sex.

```
initSex(maleFreq=0.5, sex=[], subPop=[], indRange=[], loci=[],
atPloidy=-1, stage=PreMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

**maleFreq** Male frequency. Default to 0.5. Sex will be initialized with this parameter.

**sex** A list of sexes (Male or Female) and will be applied to individuals in turn. If specified, parameter `maleFreq` is ignored.

## Member Functions

**x.apply(pop)** Apply this operator to population `pop`

**x.clone()** Deep copy of an `initSex`

### 3.2.3 Class `initByFreq` (Function form: `InitByFreq`)

Initialize genotypes by given allele frequencies, and sex by male frequency

#### Details

This operator assigns alleles at `loci` with given allele frequencies. By default, all individuals will be assigned with random alleles. If `identicalInds=True`, an individual is assigned with random alleles and is then copied to

all others. If `subPop` or `indRange` is given, multiple arrays of `alleleFreq` can be given to given different frequencies for different subpopulation or individual ranges.

### Initialization

Randomly assign alleles according to given allele frequencies

```
initByFreq(alleleFreq=[], identicalInds=False, subPop=[],  
indRange=[], loci=[], atPloidy=-1, maleFreq=0.5, sex=[],  
stage=PreMating, begin=0, end=1, step=1, at=[], rep=REP_ALL,  
grp=GRP_ALL, infoFields=[])
```

**alleleFreq** An array of allele frequencies. The sum of all frequencies must be 1; or for a matrix of allele frequencies, each row corresponds to a subpopulation or range.

**identicalInds** Whether or not make individual genotypes identical in all subpopulations. If `True`, this operator will randomly generate genotype for an individual and spread it to the whole subpopulation in the given range.

**sex** An array of sex [Male, Female, Male...] for individuals. The length of sex will not be checked. If it is shorter than the number of individuals, sex will be reused from the beginning.

**stage** Default to `PreMating`.

### Member Functions

**x.apply(pop)** Apply this operator to population `pop`

**x.clone()** Deep copy of the operator `initByFreq`

### 3.2.4 Class `initByValue` (Function form: `InitByValue`)

Initialize genotype by value and then copy to all individuals

#### Details

Operator `initByValue` gets one copy of chromosomes or the whole genotype (or of those corresponds to `loci`) of an individual and copy them to all or a subset of individuals. This operator assigns given alleles to specified individuals. Every individual will have the same genotype. The parameter combinations should be

- `value` - `subPop/indRange`: individual in `subPop` or in range(s) will be assigned genotype value;
- `subPop/indRange`: `subPop` or `indRange` should have the same length as `value`. Each item of `value` will be assigned to each `subPop` or `indRange`.

### Initialization

Initialize a population by given alleles

```
initByValue(value=[], loci=[], atPloidy=-1, subPop=[], indRange=[],  
proportions=[], maleFreq=0.5, sex=[], stage=PreMating, begin=0,  
end=1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

**maleFreq** Male frequency

**proportions** An array of percentages for each item in `value`. If given, assign given genotypes randomly.



**sex** An array of sex [Male, Female, Male...] for individuals. The length of sex will not be checked. If length of sex is shorter than the number of individuals, sex will be reused from the beginning.

**stages** Default to `PreMating`.

**value** An array of genotypes of one individual, having the same length as the length of `loci()` or `loci()*ploidy()` or `pop.genoSize()` (whole genotype) or `totNumLoci()` (one copy of chromosomes). This parameter can also be an array of arrays of genotypes of one individual. If `value` is an array of values, it should have the length one, number of subpopulations, or the length of ranges of proportions.

## Member Functions

**x.apply(pop)** Apply this operator to population `pop`

**x.clone()** Deep copy of the operator `initByValue`

### 3.2.5 Class `spread` (Function form: `Spread`)

Copy the genotype of an individual to all individuals

#### Details

Function `Spread(ind, subPop)` spreads the genotypes of `ind` to all individuals in an array of subpopulations. The default value of `subPop` is the subpopulation where `ind` resides.

#### Initialization

Copy genotypes of `ind` to all individuals in `subPop`

```
spread(ind, subPop=[], stage=PreMating, begin=0, end=1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

## Member Functions

**x.apply(pop)** Apply this operator to population `pop`

**x.clone()** Deep copy of the operator `spread`

### 3.2.6 Class `pyInit` (Function form: `PyInit`)

A python operator that uses a user-defined function to initialize individuals.

#### Details

This is a hybrid initializer. Users of this operator must supply a Python function with parameters `allele`, `ploidy` and `subpopulation indexes (index, ploidy, subPop)`, and return an allele value. This operator will loop through all individuals in each subpopulation and call this function to initialize populations. The arrange of parameters allows different initialization scheme for each subpopulation.

#### Initialization

Initialize populations using given user function

```
pyInit(func, subPop=[], loci=[], atPloidy=-1, indRange=[],
maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

**atPloidy** Initialize which copy of chromosomes. Default to all.

**func** A Python function with parameter (`index`, `ploidy`, `subPop`), where

- `index` is the allele index ranging from 0 to `totNumLoci-1`;
- `ploidy` is the index of the copy of chromosomes;
- `subPop` is the subpopulation index.

The return value of this function should be an integer.

**loci** A vector of locus indexes. If empty, apply to all loci.

**locus** A shortcut to `loci`.

**stage** Default to `PreMating`.

## Member Functions

**x.apply(pop)** Apply this operator to population `pop`

**x.clone()** Deep copy of the operator `pyInit`

## 3.3 Migration

### 3.3.1 Class `migrator`

Migrate individuals from (virtual) subpopulations to other subpopulations

#### Details

Migrator is the only way to mix genotypes of several subpopulations because mating is strictly within subpopulations in `simuPOP`. Migrators are quite flexible in `simuPOP` in the sense that

- migration can happen from and to a subset of subpopulations.
- migration can be done by probability, proportion or by counts. In the case of probability, if the migration rate from subpopulation `a` to `b` is `r`, then everyone in subpopulation `a` will have this probability to migrate to `b`. In the case of proportion, exactly `r*size_of_subPop_a` individuals (chosen by random) will migrate to subpopulation `b`. In the last case, a given number of individuals will migrate.
- new subpopulation can be generated through migration. You simply need to migrate to a subpopulation with a new subpopulation number.

#### Initialization

Create a migrator

```
migrator(rate, mode=MigrByProbability, fromSubPop=[], toSubPop=[],
stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

**fromSubPop** An array of 'from' subpopulations (a number) or virtual subpopulations (a pair of numbers). Default to all subpopulations. For example, if you define a virtual subpopulation by sex, you can use `fromSubPop=[(0,0), 1]` to choose migrants from the first virtual subpopulation of subpopulation 0, and from subpopulation 1. If a single number `sp` is given, it is interpreted as `[sp]`. Note that `fromSubPop=(0, 1)` (two subpopulation) is different from `fromSubPop=[(0,1)]` (a virtual subpopulation).

**mode** One of `MigrByProbability` (default), `MigrByProportion` or `MigrByCounts`

**rate** Migration rate, can be a proportion or counted number. Determined by parameter `mode`. `rate` should be an `m` by `n` matrix. If a number is given, the migration rate will be a `m` by `n` matrix of value `r`

**stage** Default to `PreMating`

**toSubPop** An array of 'to' subpopulations. Default to all subpopulations. If a single subpopulation is specified, `[]` can be ignored.

#### Note

- The overall population size will not be changed. (Mating schemes can do that). If you would like to keep the subpopulation sizes after migration, you can use the `newSubPopSize` or `newSubPopSizeExpr` parameter of a mating scheme.
- `rate` is a matrix with dimensions determined by `fromSubPop` and `toSubPop`. By default, `rate` is a matrix with element  $r(i, j)$ , where  $r(i, j)$  is the migration rate, probability or count from subpopulation `i` to `j`. If `fromSubPop` and/or `toSubPop` are given, migration will only happen between these subpopulations. An extreme case is 'point migration', `rate=[[r]]`, `fromSubPop=a`, `toSubPop=b` which migrate from subpopulation `a` to `b` with given rate `r`.

#### Member Functions

**x.apply(pop)** Apply the migrator

**x.clone()** Deep copy of a migrator

**x.rate()** Return migration rate

**x.setRates(rate, mode)** Set migration rate

Format should be 0-0 0-1 0-2, 1-0 1-1 1-2, 2-0, 2-1, 2-2. For mode `MigrByProbability` or `MigrByProportion`, 0-0,1-1,2-2 will be set automatically regardless of input.

### 3.3.2 Functions (Python) `MigrIslandRates`, `MigrSteppingStoneRates` (`simuUtil.py`)

Migrator is very flexible. It can accept arbitrary migration matrix, from any subset of subpopulations to any (even new) other subset of subpopulations. To facilitate the use of common theoretical migration models, several functions are defined in `simuUtil.py`.

- `MigrIslandRates(r, n)` returns a  $n \times n$  migration matrix

$$\begin{pmatrix} 1-r & \frac{r}{n-1} & \dots & \dots & \frac{r}{n-1} \\ \frac{r}{n-1} & 1-r & \dots & \dots & \frac{r}{n-1} \\ & & \dots & & \\ \frac{r}{n-1} & \dots & \dots & 1-r & \frac{r}{n-1} \\ \frac{r}{n-1} & \dots & \dots & \frac{r}{n-1} & 1-r \end{pmatrix}$$

- `MigrSteppingStoneRates(r, n, circular=False)` returns a  $n \times n$  migration matrix

$$\begin{pmatrix} 1-r & r & & & \\ r/2 & 1-r & r/2 & & \\ & & \dots & & \\ & & r/2 & 1-r & r/2 \\ & & & r & 1-r \end{pmatrix}$$

and if `circular=True`, returns

$$\begin{pmatrix} 1-r & r/2 & & r/2 \\ r/2 & 1-r & r/2 & \\ & & \dots & \\ r/2 & & r/2 & 1-r & r/2 \\ & & & r/2 & 1-r \end{pmatrix}$$

### 3.3.3 Class `pyMigrator`

A more flexible Python migrator

#### Details

This migrator can be used in two ways

- define a function that accepts a generation number and returns a migration rate matrix. This can be used in various migration rate cases.
- define a function that accepts individuals etc, and returns the new subpopulation ID.

More specifically, `func` can be

- `func(ind)` when neither `loci` nor `param` is given.
- `func(ind, genotype)` when `loci` is given.
- `func(ind, param)` when `param` is given.
- `func(ind, genotype, param)` when both `loci` and `param` are given.

#### Initialization

Create a hybrid migrator

```
pyMigrator(rateFunc=None, indFunc=None, mode=MigrByProbability,
fromSubPop=[], toSubPop=[], loci=[], param=None, stage=PreMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

**indFunc** A Python function that accepts an individual, optional genotypes and parameters, then returns a subpopulation ID. This method can be used to separate a population according to individual genotype.

**rateFunc** A Python function that accepts a generation number, current subpopulation sizes, and returns a migration rate matrix. The migrator then migrate like a usual migrator.

**stage** Default to `PreMating`

#### Member Functions

**`x.apply(pop)`** Apply a `pyMigrator`

**`x.clone()`** Deep copy of a `pyMigrator`

### 3.3.4 Class `splitSubPop` (Function form: `SplitSubPop`)

Split a subpopulation

#### Initialization

Split a subpopulation

```
splitSubPop(which=0, sizes=[], proportions=[], subPopID=[],
randomize=True, stage=PreMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

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.

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

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

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

**subPopID** New subpopulation IDs. Otherwise, the operator will automatically set new subpopulation IDs to new subpopulations.

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

#### Member Functions

**x.apply(pop)** Apply a `splitSubPop` operator

**x.clone()** Deep copy of a `splitSubPop` operator

### 3.3.5 Class `mergeSubPops` (Function form: `MergeSubPops`)

Merge subpopulations

#### Details

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

#### Initialization

Merge subpopulations

```
mergeSubPops(subPops=[], removeEmptySubPops=False, stage=PreMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

**subPops** Subpopulations to be merged. Default to all.

#### Member Functions

**x.apply(pop)** Apply a `mergeSubPops` operator

**x.clone()** Deep copy of a `mergeSubPops` operator

### 3.3.6 Class `resizeSubPops` (Function form: `ResizeSubPops`)

Resize subpopulations

#### Details

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.

#### Initialization

Resize subpopulations

```
resizeSubPops(newSizes=[], subPops=[], propagate=True,  
stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,  
grp=GRP_ALL, infoFields=[])
```

**newSizes** Of the specified (or all) subpopulations.

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

**subPops** Subpopulations to be resized. Default to all.

#### Member Functions

**x.apply(pop)** Apply a `resizeSubPops` operator

**x.clone()** Deep copy of a `resizeSubPops` operator

## 3.4 Mutation

### 3.4.1 Class `mutator`

Base class of all mutators.

#### Details

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.

#### Initialization

Create a mutator, do not call this constructor directly

```
mutator(rate=[], loci=[], maxAllele=0, output=">", outputExpr="",  
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,  
grp=GRP_ALL, infoFields=[])
```

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.

**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()`.

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

### Member Functions

**x.apply(pop)** Apply a mutator

**x.clone()** Deep copy of a mutator

**x.maxAllele()** Return maximum allowable allele number

**x.mutate(allele)** Describe how to mutate a single allele

**x.mutationCount(locus)** Return mutation count at locus

**x.mutationCounts()** Return mutation counts

**x.rate()** Return the mutation rate

**x.setMaxAllele(maxAllele)** Set maximum allowable allele

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

## 3.4.2 Class kamMutator (Function form: KamMutate)

K-Allele Model mutator.

### Details

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`.

### Initialization

Create a K-Allele Model mutator

```
kamMutator(rate=[], loci=[], maxAllele=0, output=">", outputExpr="",
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

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

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

**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)`.

### Member Functions

**x.clone()** Deep copy of a `kamMutator`

**x.mutate(allele)** Mutate to a state other than current state with equal probability

### 3.4.3 Class `smmMutator` (Function form: `SmmMutate`)

The stepwise mutation model.

#### Details

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.

#### Initialization

Create a SMM mutator

```
smmMutator(rate=[], loci=[], maxAllele=0, incProb=0.5, output=">",
outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

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.

#### Member Functions

**x.clone()** Deep copy of a `smmMutator`

### 3.4.4 Class `gsmMutator` (Function form: `GsmMutate`)

Generalized stepwise mutation model

#### Details

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.

#### Initialization

Create a `gsmMutator`

```
gsmMutator(rate=[], loci=[], maxAllele=0, incProb=0.5, p=0,
func=None, output=">", outputExpr="", stage=PostMating, begin=0,
end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

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.

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



**incProb** Probability to increase allele state. Default to 0.5.

## Member Functions

**x.clone()** Deep copy of a `gsmMutator`

**x.mutate(allele)** Mutate according to the GSM model

### 3.4.5 Class `pyMutator` (Function form: `PyMutate`)

A hybrid mutator.

#### Details

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.

#### Initialization

Create a `pyMutator`

```
pyMutator(rate=[], loci=[], maxAllele=0, func=None, output=">",
outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

## Member Functions

**x.clone()** Deep copy of a `pyMutator`

**x.mutate(allele)** Mutate according to the mixed model

### 3.4.6 Class `pointMutator` (Function form: `PointMutate`)

Point mutator

#### Details

Mutate specified individuals at specified loci to a specified 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.

#### Initialization

Create a `pointMutator`

```
pointMutator(loci, toAllele, atPloidy=[], inds=[], output=">",
outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

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

**inds** Individuals who will mutate

**toAllele** Allele that will be mutate to

## Member Functions

**x.apply(pop)** Apply a `pointMutator`

**x.clone()** Deep copy of a `pointMutator`

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

**x.mutationCounts()** Return mutation counts

## 3.5 Recombination and gene conversion

### 3.5.1 Class `recombinator`

Recombination and conversion

#### Details

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.

#### Initialization

Recombine chromosomes from parents

```
recombinator(intensity=-1, rate=[], afterLoci=[],
maleIntensity=-1, maleRate=[], maleAfterLoci=[], convProb=0,
convMode=CONVERT_NumMarkers, convParam=1., begin=0, end=-1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

**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.

**convMode** Conversion mode, determines how track length is determined.

- `CONVERT_NumMarkers` Converts a fixed number of markers.
- `CONVERT_GeometricDistribution` An geometric distribution is used to determine how many markers will be converted.

- **CONVERT\_TractLength** Converts a fixed length of tract.
- **CONVERT\_ExponentialDistribution** An exponential distribution with parameter `convLen` will be used to determine track length.

**convParam** Parameter for the conversion process. The exact meaning of this parameter is determined by `convMode`. 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  $\approx$  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.

**convProb** The 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/repared successfully, or a conversion event if the junction is not resolved/repared. 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 0.1/0.90.1 to 15/160.94 of this parameter. When `convProb` is 1, all recombination events will be conversion events.

**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.

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

**maleAfterLoci** If given, males will recombine at different locations.

**maleIntensity** Recombination intensity for male individuals. If given, parameter `intensity` will be considered as female intensity.

**maleRate** Recombination rate for male individuals. If given, parameter `rate` will be considered as female recombination rate.

**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.

## 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.

## Member Functions

**x.clone()** Deep copy of a recombinator

**x.convCount(size)** Return the count of conversion of a certain size (only valid in standard modules)

**x.convCounts()** Return the count of conversions of all sizes (only valid in standard modules)

**x.recCount(locus)** Return recombination count at a locus (only valid in standard modules)

**x.recCounts()** Return recombination counts (only valid in standard modules)

### 3.5.2 Gene conversion

simuPOP uses the Holliday junction model to simulate gene conversion. This model treats recombination and conversion as a unified process. The key features of this model is

- Two (out of four) chromatids pair and a single strand cut is made in each chromatid
- Strand exchange takes place between the chromatids
- Ligation occurs yielding two completely intact DNA molecules
- Branch migration occurs, giving regions of heteroduplex DNA
- Resolution of the Holliday junction gives two DNA molecules with heteroduplex DNA. Depending upon how the holliday junction is resolved, we either observe no exchange of flanking markers, or an exchange of flanking markers. The former forms a conversion event, which can be considered as a double recombination.

Translated to simulation, recombination and conversion are performed in the following steps

1. Users specify the following parameters to a recombinator:
  - (a) recombination points (recombinations are allowed after specified markers) (`loci`),
  - (b) recombination rates (can vary from marker to marker) (`rates`),
  - (c) probability of conversion if a recombination event happens (`convProb`),
  - (d) track length parameters (`convMode` and `convParam`, will discuss later).
2. Starting with two parental chromosomes, randomly choose one of them to copy to an offspring chromosome until a recombination event happens.
3. This recombination event is a conversion event if
  - (a) A random uniform number  $U(0,1)$  is less than the probability of conversion
  - (b) The length of flanking regions does not exceed the end of chromosomeIf a conversion happens, record the end of flanking region as another recombination event.
4. Copy from another copy of parental chromosome (recombination happens), until the recorded second recombination event is reached, or another recombination event happens.
5. Repeat these steps for all chromosomes.

The tract length of a flanking region is determined by parameters `convMode` and `convParam`. `convMode` can be

- `CONVERT_NumMarkers` Convert a fixed number (`convParam`) of markers. This is the default mode with `convParam=1`.
- `CONVERT_TractLength` Convert a fixed length (`convParam`) of chromosome regions. This can be used when markers are not equally spaced on chromosomes.
- `CONVERT_GeometricDistribution` Convert a random number of markers, with a geometric distribution with parameter `convParam`.
- `CONVERT_ExponentialDistribution` Convert a random length of chromosome region, using an exponential distribution with parameter `convParam`.

Note that

- If tract length is determined by length (CONVERT\_TractLength or CONVERT\_ExponentialDistribution), the starting point of the flanking region is uniformly distributed between marker  $i$  and  $i - 1$ , if the recombination happens at marker  $i$ . That is to say, it is possible that no marker is converted with positive tract length.
- A conversion event will act like a recombination event if its flanking region exceeds the end of chromosome, or if another recombination event happens before the end of the flanking region.

Although any parameters can be used in a recombinator, it is worth noting that

- The probability of conversion event among all recombination events is usually expressed as ratio of conversion to recombination events in the literature. This varies greatly from study to study, ranging from 0.1 to 15 (Chen et al, Nature Review Genetics, 2007). This translates to 0.1/0.9~0.1 to 15/16~0.94 of this parameter. When `lc convProb` is 1, all recombination events will be conversion events. The default value is `convProb=0`, meaning no conversion.
- 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). At this marker density, gene conversion can largely be ignored.

## 3.6 Selection

### 3.6.1 Mechanism

It is not very clear that our method agrees with the traditional 'average number of offspring' definition of fitness. (Note that this concept is very difficult to simulate because we do not know who will determine the number of offspring if two parents are involved.) We can, instead, look at the consequence of selection in a simple case (as derived in any population genetics textbook):

At generation  $t$ , genotype  $P_{11}, P_{12}, P_{22}$  has fitness values  $w_{11}, w_{12}, w_{22}$  respectively. In the next generation the proportion of genotype  $P_{11}$  etc., should be

$$\frac{P_{11}w_{11}}{P_{11}w_{11} + P_{12}w_{12} + P_{22}w_{22}}$$

Now, using the 'ability-to-mate' approach, for the sexless case, the proportion of genotype 11 will be the number of 11 individuals times its probability to be chosen:

$$n_{11} \frac{w_{11}}{\sum_{n=1}^N w_n}$$

This is, however, exactly

$$n_{11} \frac{w_{11}}{\sum_{n=1}^N w_n} = n_{11} \frac{w_{11}}{n_{11}w_{11} + n_{12}w_{12} + n_{22}w_{22}} = \frac{P_{11}w_{11}}{P_{11}w_{11} + P_{12}w_{12} + P_{22}w_{22}}$$

The same argument applies to the case of arbitrary number of genotypes and random mating.

The following operators, when applied, will set a variable `fitness` and an indicator so that selector-aware mating scheme can select individuals according to these values. This has two consequences:

- Selector only set information field and mark subpopulations as selection ready. However, how these information are used to select parents can vary from mating scheme to mating scheme. As a matter of fact, some mating schemes do not support selection at all.

- selector has to be `PreMating` operator. This is not a problem when you use the operator form of the selectors since their default stage is `PreMating`. However, if you use the function form of these selectors in a `pyOperator`, make sure to set the stage of `pyOperator` to `PreMating`.

### 3.6.2 Class selector

A base selection operator for all selectors.

#### Details

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  $f_i, i = 1, \dots, N$ , 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.

#### Initialization

Create a selector

```
selector(subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=["fitness"])
```

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

**subPops** Subpopulations that the selector will apply to. Default to all.

#### Member Functions

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

**x.clone()** Deep copy of a selector

### 3.6.3 Class mapSelector (Function form: MapSelector)

Selection according to the genotype at one locus

#### Details

This map selector implements selection at one locus. A user provided dictionary (map) of genotypes will be used in this selector to set each individual's fitness value.

#### Initialization

Create a map selector

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

**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.

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

**locus** The locus index. A shortcut to loci=[locus]

**output** And other parameters please refer to help (baseOperator.\_\_init\_\_)

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

#### Member Functions

**x.clone()** Deep copy of a map selector

The example for class mapSelector is a typical example of heterozygote superiority. When  $w_{11} < w_{12} > w_{22}$ , the genotype frequencies will go to an equilibrium state. Theoretically, if

$$\begin{aligned}s_1 &= w_{12} - w_{11} \\ s_2 &= w_{12} - w_{22}\end{aligned}$$

the stable allele frequency of allele 1 is

$$p = \frac{s_2}{s_1 + s_2}$$

Which is .677 in the example ( $s_1 = .1$ ,  $s_2 = .2$ ).

### 3.6.4 Class maSelector (Function form: MaSelect)

Multiple allele selector (selection according to wildtype or diseased alleles)

#### Details

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.

### Initialization

Create a multiple allele selector

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

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.

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

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

### Note

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

### Member Functions

**x.clone()** Deep copy of a `maSelector`

## 3.6.5 Class `mlSelector` (Function form: `MlSelect`)

Selection according to genotypes at multiple loci in a multiplicative model

### Details

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

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

### Initialization

Create a multiple-locus selector

```
mlSelector(selectors, mode=SEL_Multiplicative, subPops=[],
stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["fitness"])
```

Please refer to `mapSelector` for other parameter descriptions.



**selectors** A list of selectors

## Member Functions

**x.clone()** Deep copy of a `mlSelector`

### 3.6.6 Class `pySelector` (Function form: `PySelect`)

Selection using user provided function

#### Details

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.

#### Initialization

Create a Python hybrid selector

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

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

**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.

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

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

## Member Functions

**x.clone()** Deep copy of a `pySelector`

## 3.7 Penetrance

### 3.7.1 Class `penetrance`

Base class of all penetrance operators.

#### Details

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.

### Initialization

Create a penetrance operator

```
penetrance(ancestralGen=-1, stage=DuringMating, begin=0, end=-1,
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

**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.

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

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

### Member Functions

**x.apply(pop)** Set penetrance to all individuals and record penetrance if requested

**x.clone()** Deep copy of a penetrance operator

## 3.7.2 Class `mapPenetrance` (Function form: `MapPenetrance`)

Penetrance according to the genotype at one locus

### Details

Assign penetrance using a table with keys 'X-Y' where X and Y are allele numbers.

### Initialization

Create a map penetrance operator

```
mapPenetrance(loci, penet, phase=False, ancestralGen=-1,
stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

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

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

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

**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`.

## Member Functions

**x.clone()** Deep copy of a map penetrance operator

### 3.7.3 Class `maPenetrance` (Function form: `MaPenetrance`)

Multiple allele penetrance operator

#### Details

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.

#### Initialization

Create a multiple allele penetrance operator ( penetrance according to diseased or wildtype alleles)

```
maPenetrance(loci, penet, wildtype, ancestralGen=-1,
stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

**loci** The locus indexes. The genotypes of these loci will be examined.

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

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

**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.

## Member Functions

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

### 3.7.4 Class `mlPenetrance` (Function form: `MlPenetrance`)

Penetrance according to the genotype according to a multiple loci multiplicative model

#### Details

This is the 'multiple-locus' penetrance 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.

### Initialization

Create a multiple locus penetrance operator

```
mlPenetrance(peneOps, mode=PEN_Multiplicative, ancestralGen=-1,
stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

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

**peneOps** A list of penetrance operators

### Member Functions

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

## 3.7.5 Class `pyPenetrance` (Function form: `PyPenetrance`)

Assign penetrance values by calling a user provided function

### Details

For each individual, the penetrance is determined by a user-defined penetrance function `func`. This function takes genotypes 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.

### Initialization

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

```
pyPenetrance(loci, func, ancestralGen=-1, stage=DuringMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

**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.

**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.

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

## Member Functions

**x.clone()** Deep copy of a Python penetrance operator

## 3.8 Quantitative Trait

### 3.8.1 Class `quanTrait`

Base class of quantitative trait

#### Details

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.

#### Initialization

Create a quantitative trait operator

```
quanTrait(ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=["qtrait"])
```

## Member Functions

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

**x.clone()** Deep copy of a quantitative trait operator

### 3.8.2 Class `mapQuanTrait` (Function form: `MapQuanTrait`)

Quantitative trait according to genotype at one locus

#### Details

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  $N(0, \sigma^2)$ . This random part is usually considered as the environmental factor of the trait.

#### Initialization

Create a map quantitative trait operator

```
mapQuanTrait(loci, qtrait, sigma=0, phase=False, ancestralGen=-1,
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["qtrait"])
```

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

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

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

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

**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  $N(\text{mean}, \sigma^2)$ . For multiple loci, the form is 'a-b1c-d1e-f' etc.

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

## Member Functions

**x.clone()** Deep copy of a map quantitative trait operator

### 3.8.3 Class `maQuanTrait` (Function form: `MaQuanTrait`)

Multiple allele quantitative trait (quantitative trait according to disease or wildtype alleles)

#### Details

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  $N(0, \sigma^2)$  will be added to the returned trait value.

#### Initialization

Create a multiple allele quantitative trait operator

```
maQuanTrait(loci, qtrait, wildtype, sigma=[], ancestralGen=-1,
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["qtrait"])
```

Please refer to `quanTrait` for other parameter descriptions.

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

**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]`.

## Member Functions

**x.clone()** Deep copy of a multiple allele quantitative trait

### 3.8.4 Class `mlQuanTrait` (Function form: `MlQuanTrait`)

Quantitative trait according to genotypes from a multiple loci multiplicative model

#### Details

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

- **QT\_Multiplicative**: the mean of the quantitative trait is calculated as  $f = \prod f_i$ .
- **QT\_Additive**: the mean of the quantitative trait is calculated as  $f = \sum f_i$ .

Note that all  $\sigma_i$  (for  $f_i$ ) and  $\sigma$  (for  $f$ ) will be considered. I.e, the trait value should be

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

for QT\_Additive case. If this is not desired, you can set some of the  $\sigma$  to zero.

### Initialization

Create a multiple locus quantitative trait operator

```
mlQuanTrait(qtraits, mode=QT_Multiplicative, sigma=0,
ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=["qtrait"])
```

Please refer to `quanTrait` for other parameter descriptions.

**mode** Can be one of `QT_Multiplicative` and `QT_Additive`

**qtraits** A list of quantitative traits

### Member Functions

**x.clone()** Deep copy of a multiple loci quantitative trait operator

## 3.8.5 Class `pyQuanTrait` (Function form: `PyQuanTrait`)

Quantitative trait using a user provided function

### Details

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

### Initialization

Create a Python quantitative trait operator

```
pyQuanTrait(loci, func, ancestralGen=-1, stage=PostMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=["qtrait"])
```

Please refer to `quanTrait` for other parameter descriptions.

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

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

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

### Member Functions

**x.clone()** Deep copy of a Python quantitative trait operator

## 3.9 Ascertainment

### 3.9.1 Class `sample`

Base class of other sample operator

#### Details

Ascertainment/sampling refers to the ways of selecting individuals from a population. In `simuPOP`, ascertainment operators create sample populations that can be accessed from the population's local namespace. All the ascertainment operators work like this except for `pySubset` which shrink the population itself.

Individuals in sampled populations may or may not keep their original order but their indexes in the whole population are stored in an information field `oldindex`. This is to say, you can use `ind.info('oldindex')` to check the original position of an individual.

Two forms of sample size specification are supported: with or without subpopulation structure. For example, the `size` parameter of `randomSample` can be a number or an array (which has the length of the number of subpopulations). If a number is given, a sample will be drawn from the whole population, regardless of the population structure. If an array is given, individuals will be drawn from each subpopulation `sp` according to `size[sp]`.

An important special case of sample size specification occurs when `size=[]` (default). In this case, usually all qualified individuals will be returned.

The function forms of these operators are a little different from others. They do return a value: an array of samples.

#### Initialization

Draw a sample

```
sample(name="sample", nameExpr="", times=1, saveAs="", saveAsExpr="",
format="auto", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Please refer to `baseOperator.__init__` for other parameter descriptions.

**format** Format to save the samples

**name** Name of the sample in the local namespace. This variable is an array of populations of size `times`. Default to `sample`.

**nameExpr** Expression version of parameter `name`. If both `name` and `nameExpr` are empty, sample populations will not be saved in the population's local namespace. This expression will be evaluated dynamically in population's local namespace.

**saveAs** Filename to save the samples

**saveAsExpr** Expression version of parameter `saveAs`. It will be evaluated dynamically in population's local namespace.

**times** How many times to sample from the population. This is usually 1, but we may want to take several random samples.

#### Member Functions

**`x.apply(pop)`** Apply the `sample` operator

**`x.clone()`** Deep copy of a `sample` operator

**`x.samples(pop)`** Return the samples



### 3.9.2 Class `pySubset` (Function form: `PySubset`)

Shrink population

#### Details

This operator shrinks a population according to a given array or the `subPopID()` value of each individual. Individuals with negative subpopulation IDs will be removed.

#### Initialization

Create a `pySubset` operator

```
pySubset(keep=[], stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

**keep** An array of individual subpopulation IDs

#### Member Functions

**`x.apply(pop)`** Apply the `pySubset` operator

**`x.clone()`** Deep copy of a `pySubset` operator

### 3.9.3 Class `pySample` (Function form: `PySample`)

Python sampler.

#### Details

A Python sampler that generate a sample with given individuals. This sampler accepts a Python array with elements that will be assigned to individuals as their subpopulation IDs. Individuals with positive subpopulation IDs will then be picked out and form a sample.

#### Initialization

Create a Python sampler

```
pySample(keep, keepAncestralPops=-1, name="sample", nameExpr="",
times=1, saveAs="", saveAsExpr="", format="auto", stage=PostMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

Please refer to class `sample` for other parameter descriptions.

**keep** Subpopulation IDs of all individuals

**keepAncestralPop** The number of ancestral populations that will be kept. If `-1` is given, keep all ancestral populations (default). If `0` is given, no ancestral population will be kept.

#### Member Functions

**`x.clone()`** Deep copy of a Python sampler

**`x.drawsample(pop)`** Draw a Python sample

### 3.9.4 Class `randomSample` (Function form: `RandomSample`)

Randomly draw a sample from a population

#### Details

This operator will randomly choose `size` individuals (or `size[i]` individuals from subpopulation `i`) and return a new population. The function form of this operator returns the samples directly. This operator keeps samples in an array `name` in the local namespace. You may access them through `dvars()` or `vars()` functions.

The original subpopulation structure or boundary is kept in the samples.

#### Initialization

Draw a random sample, regardless of the affectedness status

```
randomSample(size=[], name="sample", nameExpr="", times=1, saveAs="",
saveAsExpr="", format="auto", stage=PostMating, begin=0, end=-1,
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Please refer to class `sample` for other parameter descriptions.

**size** Size of the sample. It can be either a number which represents the overall sample size, regardless of the population structure; or an array which represents the number of individuals drawn from each subpopulation.

#### Note

Ancestral populations will not be copied to the samples.

#### Member Functions

**`x.clone()`** Deep copy of a `randomSample` operator

### 3.9.5 Class `caseControlSample` (Function form: `CaseControlSample`)

Draw a case-control sample from a population

#### Details

This operator will randomly choose `cases` affected individuals and `controls` unaffected individuals as a sample. The affectedness status is usually set by penetrance functions or operators. The sample populations will have two subpopulations: cases and controls.

You may specify the number of cases and the number of controls from each subpopulation using the array form of the parameters. The sample population will still have only two subpoulations (cases and controls) though.

A special case of this sampling scheme occurs when one of or both `cases` and `controls` are omitted (zeros). In this case, all cases and/or controls are chosen. If both parameters are omitted, the sample is effectively the same population with affected and unaffected individuals separated into two subpopulations.

#### Initialization

Draw cases and controls as a sample

```
caseControlSample(cases=[], controls=[], spSample=False,
name="sample", nameExpr="", times=1, saveAs="", saveAsExpr="",
format="auto", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Please refer to class `sample` for other parameter descriptions.

**cases** The number of cases, or an array of the numbers of cases from each subpopulation

**controls** The number of controls, or an array of the numbers of controls from each subpopulation

## Member Functions

**x.clone()** Deep copy of a `caseControlSample` operator

### 3.9.6 Class `affectedSibpairSample` (Function form: `AffectedSibpairSample`)

Draw an affected sibling pair sample

#### Details

Special preparation for the population is needed in order to use this operator. Obviously, to obtain affected sibling pairs, we need to know the parents and the affectedness status of each individual. Furthermore, to get parental genotypes, the population should have `ancestralDepth` at least 1. The most important problem, however, comes from the mating scheme we are using.

`randomMating()` is usually used for diploid populations. The *real random* mating requires that a mating will generate only one offspring. Since parents are chosen with replacement, a parent can have multiple offspring with different parents. On the other hand, it is very unlikely that two offspring will have the same parents. The probability of having a sibling for an offspring is  $\frac{1}{N^2}$  (if do not consider selection). Therefore, we will have to allow multiple offspring per mating at the cost of small effective population size.

#### Initialization

Draw an affected sibling pair sample

```
affectedSibpairSample(size=[], chooseUnaffected=False,
countOnly=False, name="sample", nameExpr="", times=1, saveAs="",
saveAsExpr="", format="auto", stage=PostMating, begin=0, end=-1,
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=["father_idx",
"mother_idx"])
```

Please refer to class `sample` for other parameter descriptions.

**chooseUnaffected** Instead of affected sibpairs, choose unaffected families.

**countOnly** Set variables about the number of affected sibpairs, do not actually draw the sample

**size** The number of affected sibling pairs to be sampled. Can be a number or an array. If a number is given, it is the total number of sibpairs, ignoring the population structure. Otherwise, specified numbers of sibpairs are sampled from subpopulations. If `size` is unspecified, this operator will return all affected sibpairs.

## Member Functions

**x.clone()** Deep copy of a `affectedSibpairSample` operator

**x.drawsample(pop)** Draw a sample

**x.prepareSample(pop)** Preparation before drawing a sample

### 3.9.7 Class `largePedigreeSample`

Draw a large pedigree sample

#### Initialization

Draw a large pedigree sample

```
largePedigreeSample(size=[], minTotalSize=0, maxOffspring=5,  
minPedSize=5, minAffected=0, countOnly=False, name="sample",  
nameExpr="", times=1, saveAs="", saveAsExpr="", format="auto",  
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,  
grp=GRP_ALL, infoFields=["father_idx", "mother_idx"])
```

Please refer to class `sample` for other parameter descriptions.

**countOnly** Set variables about the number of affected sibpairs, do not actually draw the sample.

**maxOffspring** The maximum number of offspring a parent may have

**minAffected** The minimal number of affected individuals in each pedigree. Default to 0.

**minPedSize** The minimal pedigree size. Default to 5.

**minTotalSize** The minimum number of individuals in the sample

#### Member Functions

**x.clone()** Deep copy of a `largePedigreeSample` operator

**x.drawsample(pop)** Draw a large pedigree sample

**x.prepareSample(pop)** Preparation before drawing a sample

### 3.9.8 Class `nuclearFamilySample`

Draw a nuclear family sample

#### Initialization

Draw a nuclear family sample

```
nuclearFamilySample(size=[], minTotalSize=0, maxOffspring=5,  
minPedSize=5, minAffected=0, countOnly=False, name="sample",  
nameExpr="", times=1, saveAs="", saveAsExpr="", format="auto",  
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,  
grp=GRP_ALL, infoFields=["father_idx", "mother_idx"])
```

Please refer to class `sample` for parameter descriptions.

#### Member Functions

**x.clone()** Deep copy of a `nuclearFamilySample` operator

**x.drawsample(pop)** Draw a nuclear family sample

**x.prepareSample(pop)** Preparation before drawing a sample

## 3.10 Statistics Calculation

### 3.10.1 Class `stator`

Base class of all the statistics calculator

#### Details

Operator `stator` calculates various basic statistics for the population and set variables in the local namespace. Other operators or functions can refer to the results from the namespace after `stat` is applied.

#### Initialization

Create a stator

```
stator(output="", outputExpr="", stage=PostMating, begin=0, end=-1,
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

#### Member Functions

**`x.clone()`** Deep copy of a stator

### 3.10.2 Class `stat` (Function form: `Stat`)

Calculate statistics

#### Details

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 disequilibrium is requested.

#### Initialization

Create an `stat` operator

```
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="", outputExpr="", stage=PostMating, begin=0, end=-1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

**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);
- $\theta$  (  $F_{ST}$  ) 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 chosen from the following options: `Fst`, `Fis`, `Fit`, `AvgFst`, `AvgFis`, and `AvgFit`.

**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,

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

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

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

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

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']['allele1-allele2']`
- `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.

**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 chosen from the following options: `numOfAlleles`, `alleleNum`, and `alleleFreq`.

**association** Association measures

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

**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 chosen from the following options: `subpop` and `midValues`.

**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`.

**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 sequences on loci haplo.
- `haploFreq[haplo], subPop[sp]['haploFreq'][haplo]`, the frequency of allele sequences on loci haplo.

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

**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 calculated 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]`.

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

**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 chosen 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 chosen from the following options: `numOfAffected`, `propOfAffected`, `numOfUnaffected`, `propOfUnaffected`.

**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 chosen from the following options: `numOfMale`, `propOfMale`, `numOfFemale`, and `propOfFemale`.

**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.

**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

**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. `relatedness[grp1][grp2]` is the relatedness value between `grp1` and `grp2`. There is no subpopulation level relatedness value.

**rel\_param** A dictionary of parameters of relatedness statistics. Can be one or more items chosen from the following options: `Fst`, `Fis`, `Fit`, `AvgFst`, `AvgFis`, and `AvgFit`.

## Member Functions

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

**x.clone()** Deep copy of a `stat` operator

## 3.11 Expression and Statements

### 3.11.1 Class `dumper`

Dump the content of a population.

#### Initialization

Dump a population

```
dumper(alleleOnly=False, infoOnly=False, ancestralPops=False,
dispWidth=1, max=100, chrom=[], loci=[], subPop=[], indRange=[],
output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

**alleleOnly** Only display allele

**ancestralPops** Whether or not display ancestral populations. Default to `False`.

**chrom** Chromosome(s) to display

**dispWidth** Number of characters to display an allele. Default to 1.

**indRange** Range(s) of individuals to display

**infoOnly** Only display genotypic information

**loci** Loci to display

**max** The maximum number of individuals to display. Default to 100. This is to avoid careless dump of huge populations.

**output** Output file. Default to the standard output.

**outputExpr** And other parameters: refer to `help(baseOperator.__init__)`

**subPop** Only display subpopulation(s)

## Member Functions

**x.alleleOnly()** Only show alleles (not structure, gene information?)

**x.apply(pop)** Apply to one population. It does not check if the operator is activated.

**x.clone()** Deep copy of an operator

**x.infoOnly()** Only show info

**x.setAlleleOnly(alleleOnly)** `SimuPOP::dumper::setAlleleOnly`

**x.setInfoOnly(infoOnly)** `SimuPOP::dumper::setInfoOnly`

### 3.11.2 Class `savePopulation`

Save population to a file

#### Initialization

Save population

```
savePopulation(output="", outputExpr="", format="bin", compress=True,
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

## Member Functions

**x.apply(pop)** Apply to one population. It does not check if the operator is activated.

**x.clone()** Deep copy of an operator

### 3.11.3 Class `pyOutput`

Output a given string.

#### Details

A common usage is to output a new line for the last replicate.

#### Initialization

Create a `pyOutput` operator that outputs a given string.

```
pyOutput(str="", output=">", outputExpr="", stage=PostMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

**str** String to be outputted

## Member Functions

**x.apply(pop)** Simply output some info  
**x.clone()** Deep copy of an operator  
**x.setString(str)** Set output string.

### 3.11.4 Class `pyEval` (Function form: `PyEval`)

Evaluate an expression

#### Details

Python expressions/statements will be executed when `pyEval` is applied to a population by using parameters `expr/stmts`. Statements can also be executed when `pyEval` is created and destroyed or before `expr` is executed. The corresponding parameters are `preStmts`, `postStmts` and `stmts`. For example, operator `varPlotter` uses this feature to initialize R plots and save plots to a file when finished.

#### Initialization

Evaluate expressions/statments in the local namespace of a replicate

```
pyEval(expr="", stmts="", preStmts="", postStmts="", exposePop=False,
name="", output=">", outputExpr="", stage=PostMating, begin=0,
end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

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

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

**name** Used to let pure Python operator to identify themselves

**output** Default to `>`. I.e., output to standard output.

**postStmts** The statement that will be executed when the operator is destroyed

**preStmts** The statement that will be executed when the operator is constructed

**stmts** The statement that will be executed before the expression

#### Member Functions

**x.apply(pop)** Apply the `pyEval` operator

**x.clone()** Deep copy of a `pyEval` operator

**x.name()** Return the name of an expression

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

### 3.11.5 Class `pyExec` (Function form: `PyExec`)

Execute a Python statement

#### Details

This operator takes a list of statements and executes them. No value will be returned or outputted.

#### Initialization

Evaluate statments in the local replicate namespace, no return value

```
pyExec(stmts="", preStmts="", postStmts="", exposePop=False, name="",
output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Please refer to class `pyEval` for parameter descriptions.

## Member Functions

**x.clone()** Deep copy of a `pyExec` operator

### 3.11.6 Class `infoEval` (Function form: `infoEval`)

#### Details

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. Its 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.

#### Initialization

Evaluate Python statements with variables being an individual's information fields

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

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 available 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 exception will be raised 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.

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

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

**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.

**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`

## Member Functions

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

**x.clone()** Deep copy of a `infoEval` operator

**x.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`.

### 3.11.7 Class `infoExec` (Function form: `infoExec`)

Execute a Python statement for each individual, using information fields

#### Details

This operator takes a list of statements and executes them. No value will be returned or outputted.

#### Initialization

Evaluate statments in the a namespace consists of individual information fields, optionally with variable in population's local namespace

```
infoExec(stmts="", subPops=[], usePopVars=False, exposePop=False,
name="", output=">", outputExpr="", stage=PostMating, begin=0,
end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Please refer to class `infoEval` for parameter descriptions.

#### Member Functions

**x.clone()** Deep copy of a `infoExec` operator

## 3.12 Tagging (used for pedigree tracking)

### 3.12.1 Class `tager`

Base class of tagging individuals

#### Details

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.

#### Initialization

Create a `tager`, default to be always active but no output

```
tager(output="", outputExpr="", begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

#### Member Functions

**x.clone()** Deep copy of a  
`tager`

### 3.12.2 Class `inheritTagger`

Inherit tag from parents

#### Details

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.

#### Initialization

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

```
inheritTagger(mode=TAG_Paternal, begin=0, end=-1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, output="", outputExpr="",
infoFields=["paternal_tag", "maternal_tag"])
```

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

#### Member Functions

**x.clone()** Deep copy of a `inheritTagger`

### 3.12.3 Class `parentTagger`

Tagging according to parental indexes

#### Details

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`.

#### Initialization

Create a `parentTagger`

```
parentTagger(begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, output="", outputExpr="", infoFields=["parent_idx"])
```

#### Member Functions

**x.apply(pop)** At the end of a generation, write population structure information to a file with a newline.

**x.clone()** Deep copy of a `parentTagger`

### 3.12.4 Class `parentsTagger`

Tagging according to parents' indexes

#### Details

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.

### Initialization

Create a `parentsTagger`

```
parentsTagger(begin=0, end=-1, step=1, at=[], rep=REP_ALL,
  grp=GRP_ALL, output="", outputExpr="", infoFields=["father_idx",
  "mother_idx"])
```

### Member Functions

**x.apply(pop)** At the end of a generation, write population structure information to a file with a newline.

**x.clone()** Deep copy of a `parentsTagger`

## 3.12.5 Class `sexTagger`

Tagging sex status.

### Details

This is a simple post-mating tagger that write sex status to a file. By default, 1 for Male, 2 for Female.

### Initialization

`SimuPOP::sexTagger::sexTagger`

```
sexTagger(code=[], begin=0, end=-1, step=1, at=[], rep=REP_ALL,
  grp=GRP_ALL, stage=PostMating, output=">", outputExpr="",
  infoFields=[])
```

**code** Code for Male and Female, default to 1 and 2, respectively. This is used by Linkage format.

## 3.12.6 Class `affectionTagger`

Tagging affection status.

### Details

This is a simple post-mating tagger that write affection status to a file. By default, 1 for unaffected, 2 for affected.

### Initialization

`SimuPOP::affectionTagger::affectionTagger`

```
affectionTagger(code=[], begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, stage=PostMating, output=">", outputExpr="",
infoFields=[])
```

**code** Code for Male and Female, default to 1 and 2, respectively. This is used by Linkage format.

### 3.12.7 Class `infoTagger`

Tagging information fields.

#### Details

This is a simple post-mating tagger that write given information fields to a file (or standard output).

#### Initialization

SimuPOP::infoTagger::infoTagger

```
infoTagger(begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
stage=PostMating, output=">", outputExpr="", infoFields=[])
```

### 3.12.8 Class `pyTagger`

Python tagger.

#### Details

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.

#### Initialization

Creates a `pyTagger` that works on specified information fields

```
pyTagger(func=None, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, output="", outputExpr="", infoFields=[])
```

**func** A Python 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.

**infoFields** Information fields. The user should guarantee the existence of these fields.

#### Member Functions

**x.clone()** Deep copy of a `pyTagger`

## 3.13 Terminator

### 3.13.1 Class `terminator`

Base class of all terminators.



## Details

Terminators are used to see if an evolution is running as expected, and terminate the evolution if a certain condition fails.

## Initialization

Create a terminator

```
terminator(message="", output=">", outputExpr="", stage=PostMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

**message** A message that will be displayed when the evolution is terminated.

## Member Functions

**x.clone()** Deep copy of a terminator

### 3.13.2 Class `terminateIf`

Terminate according to a condition

## Details

This operator terminates the evolution under certain conditions. For example, `terminateIf(condition='alleleFreq[0][1]<0.05', begin=100)` terminates the evolution if the allele frequency of allele 1 at locus 0 is less than 0.05. Of course, to make this operator work, you will need to use a `stat` operator before it so that variable `alleleFreq` exists in the local namespace.

When the value of condition is `True`, a shared variable `var="terminate"` will be set to the current generation.

## Initialization

Create a `terminateIf` terminator

```
terminateIf(condition="", message="", var="terminate", output="",
outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

## Member Functions

**x.apply(pop)** Apply the `terminateIf` terminator

**x.clone()** Deep copy of a `terminateIf` terminator

### 3.13.3 Class `continueIf`

Terminate according to a condition failure

## Details

The same as `terminateIf` but continue if the condition is `True`.

## Initialization

Create a `continueIf` terminator

```
continueIf(condition="", message="", var="terminate", output="",
outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

## Member Functions

**x.apply(pop)** Apply this operator

**x.clone()** Deep copy of a `continueIf` terminator

## 3.14 Conditional operator

### 3.14.1 Class `ifElse`

Conditional operator

#### Details

This operator accepts

- an expression that will be evaluated when this operator is applied.
- an operator that will be applied if the expression is `True` (default to null).
- an operator that will be applied if the expression is `False` (default to null).

When this operator is applied to a population, it will evaluate the expression and depending on its value, apply the supplied operator. Note that the `begin`, `end`, `step`, and `at` parameters of `ifOp` and `elseOp` will be ignored. For example, you can mimic the `at` parameter of an operator by `ifElse('rep in [2,5,9]' operator)`. The real use of this mechanism is to monitor the population statistics and act accordingly.

#### Initialization

Create a conditional operator

```
ifElse(cond, ifOp=None, elseOp=None, output=">", outputExpr="",
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

**cond** Expression that will be treated as a boolean variable

**elseOp** An operator that will be applied when `cond` is `False`

**ifOp** An operator that will be applied when `cond` is `True`

## Member Functions

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

**x.clone()** Deep copy of an `ifElse` operator

## 3.15 Debug-related operators/functions

Standard `simuPOP` module can print out lots of debug information upon request. These are mostly used for internal debugging purposes but you can also use them when error happens. For example, the following code will crash `simuPOP`:

```
>>> population(1).individual(0).arrAllele()
```

It is not clear why this simple line will cause us trouble, instead of outputting the genotype of the only individual of this population. However, the reason is clear if you turn on debug information:

Example 3.3: Turn on/off debug information

```
>>> TurnOnDebug(DBG_ALL)
Debug code DBG_ALL is turned on. cf. listDebugCode(), turnOffDebug()
>>> population(1).individual(0).arrAlleles()
Constructor of Population is called
Population size 1
Destructor of Population is called
Segmentation fault (core dumped)
```

`population(1)` creates a temporary object that is destroyed right after the execution of the input. When Python tries to display the genotype, it will refer to an invalid location. The right way to do this is to create a persistent population object:

```
>>> pop = population(1)
>>> pop.individual(0).arrAllele()
```

If the output is overwhelming after you turn on all debug information, you can turn on certain part of the information by using the following functions:

- `ListDebugCode()` list all debug code.
- `turnOnDebug()`, `TurnOnDebug(code)` turn on debug codes.
- `turnOffDebug()`, `TurnOffDebug(code)` turn off debug codes.

`turnOnDebug()` and `turnOffDebug()` are operators and accept all operator parameters `begin`, `step` etc. Usually, you can use `turnOnDebug` to output more information about a potential bug before `simuPOP` starts to misbehave.

Another useful debug code is `DBG_PROFILE`. When turned on, it will display running time of each operator. This will give you a good sense of which operator runs slowly (or simply the order of operator execution if you are not sure). If most of the execution time is spent on a pure-Python operator, you may want to rewrite it in C++. Note that `DBG_PROFILE` is suitable for measuring individual operator performance. If you would like to measure the execution time of all operators in several generations, `ticToc` operator is better.

### 3.15.1 Class `turnOnDebug` (Function form: `TurnOnDebug`)

Set debug on

#### Details

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 `TurnOnDebug` or `TurnOnDebugByName` function.
- use this `turnOnDebug` operator

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

### Initialization

Create a `turnOnDebug` operator

```
turnOnDebug(code, stage=PreMating, begin=0, end=-1, step=1, at=[],
            rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

### Member Functions

**x.apply(pop)** Apply the `turnOnDebug` operator to one population

**x.clone()** Deep copy of a `turnOnDebug` operator

## 3.15.2 Class `turnOffDebug` (Function form: `TurnOffDebug`)

Set debug off

### Details

Turn off debug.

### Initialization

Create a `turnOffDebug` operator

```
turnOffDebug(code, stage=PreMating, begin=0, end=-1, step=1, at=[],
            rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

### Member Functions

**x.apply(pop)** Apply the `turnOffDebug` operator to one population

**x.clone()** Deep copy of a `turnOffDebug` operator

## 3.16 Miscellaneous

### 3.16.1 Class `noneOp`

None operator

### Details

This operator does nothing.

### Initialization

Create a `none` operator

```
noneOp(output=">", outputExpr="", stage=PostMating, begin=0, end=0,
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

## Member Functions

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

**x.clone()** Deep copy of a noneOp operator

### 3.16.2 Class pause

Pause a simulator

#### Details

This operator pauses the evolution of a simulator at given generations or at a key stroke, using `stopOnKeyStroke=True` option. Users can use 'q' to stop an evolution. When a simulator is stopped, press any other key to resume the simulation or escape to a Python shell to examine the status of the simulation by pressing 's'.

There are two ways to use this operator, the first one is to pause the simulation at specified generations, using the usual operator parameters such as `at`. Another way is to pause a simulation with any key stroke, using the `stopOnKeyStroke` parameter. This feature is useful for a presentation or an interactive simulation. When 's' is pressed, this operator expose the current population to the main Python dictionary as variable `pop` and enter an interactive Python session. The way current population is exposed can be controlled by parameter `exposePop` and `popName`. This feature is useful when you want to examine the properties of a population during evolution.

#### Initialization

Stop a simulation. Press 'q' to exit or any other key to continue.

```
pause(prompt=True, stopOnKeyStroke=False, exposePop=True,
popName="pop", output=">", outputExpr="", stage=PostMating, begin=0,
end=-1, step=1, at=[], rep=REP_LAST, grp=GRP_ALL, infoFields=[])
```

**exposePop** Whether or not expose pop to user namespace, only useful when user choose 's' at pause. Default to True.

**popName** By which name the population is exposed. Default to pop.

**prompt** If True (default), print prompt message.

**stopOnKeyStroke** If True, stop only when a key was pressed.

## Member Functions

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

**x.clone()** Deep copy of a pause operator

### 3.16.3 Class ticToc (Function form: TicToc)

Timer operator

#### Details

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.

### Initialization

Create a timer

```
ticToc(output=">", outputExpr="", stage=PreMating, begin=0, end=-1,  
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

### Member Functions

**x.apply(pop)** Apply the `ticToc` operator to one population

**x.clone()** Deep copy of a `ticToc` operator

## 3.16.4 Class `setAncestralDepth`

Set ancestral depth

### Details

This operator set the number of ancestral generations to keep in a population. It is usually called like `setAncestral(at=[-2])` to start recording ancestral generations to a population at the end of the evolution. This is useful when constructing pedigree trees from a population.

### Initialization

Create a `setAncestralDepth` operator

```
setAncestralDepth(depth, output=">", outputExpr="", stage=PreMating,  
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,  
infoFields=[])
```

### Member Functions

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

**x.clone()** Deep copy of a `setAncestralDepth` operator

# Global and Python Utility functions

## 4.1 Global functions

**AlleleType()**

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

**Limits()**

Print out system limits

**ListAllRNG()**

List the names of all available random number generators

**ListDebugCode()**

List all debug codes

**LoadPopulation(file, format="auto")**

Load a population from a file. The file format is by default determined by file extension (`format="auto"`). Otherwise, `format` can be one of `txt`, `bin`, or `xml`.

**LoadSimulator(file, mate, format="auto")**

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  $1, 2^8 - 1, 2^{16} - 1$  for binary, short, or long allele modules, respectively

**MergePopulations(pops, newSubPopSizes=[], keepAncestralPops=-1)**

Merge several populations with the same genotypic structure and create a new population

**MergePopulationsByLoci(pops, newNumLoci=[], newLociPos=[], byChromosome=False)**

Merge several populations of the same size by loci and create a new population

**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.

### **TurnOffDebug(code=DBG\_ALL)**

Turn off debug information. Default to turn off all debug codes. Only available in non-optimized modules.

### **TurnOnDebug(code=DBG\_ALL)**

Set debug codes. Default to turn on all debug codes. Only available in non-optimized modules.

### **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

## 4.2 Utility Modules

Several utility modules are distributed with simuPOP. They provide important functions and extensions to simuPOP and serve as good examples on how simuPOP can be used.

Compared to simuPOP kernel functions, these utility functions are less tested, and are subject to more frequent changes. Please report to simuPOP mailing list if any function stops working.

### 4.2.1 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-optimizedmodules.

**-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. Bydefault, wxPython is used whenever possible.



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

Because these options are reserved, you can not use them in your simuPOP script.

## Module Functions

**getParam (options=[], doc="", details="", noDialog=False, UnprocessedArgs=True, verbose=False, nCol=1)**

Get parameters from either

- a Tcl/Tk based, or wxPython based parameter dialog (wxPython is used if it is available)
- command line argument
- configuration file specified by `-c file (-config file)`, or
- prompt for user input

The option description list consists of dictionaries with some predefined keys. Each dictionary defines an option. Each option description item can have the following keys:

**arg** short command line option name. 'h' checks the presence of argument -h. If an argument is expected, add a comma to the option name. For example, 'p:' matches command line option -p=100 or -p 100.

**longarg** long command line option name. 'help' checks the presence of argument '-help'. 'mu=' matches command line option -mu=0.001 or -mu 0.001.

**label** The label of the input field in a parameter dialog, and as the prompt for user input.

**default** default value for this parameter. It is used to as the default value in the parameter dialog, and as the option value when a user presses 'Enter' directly during interactive parameter input.

**useDefault** use default value without asking, if the value can not be determined from GUI, command line option or config file. This is useful for options that rarely need to be changed. Setting them to useDefault allows shorter command lines, and easy user input.

**description** a long description of this parameter, will be put into the usage information, which will be displayed with (-h, -help command line option, or help button in parameter dialog).

**allowedTypes** acceptable types of this option. If allowedTypes is types.ListType or types.TupleType and the user's input is a scalar, the input will be converted to a list automatically. If the conversion can not be done, this option will not be accepted.

**validate** a function to validate the parameter. You can define your own functions or use the ones defined in this module.

**chooseOneOf** if specified, simuOpt will choose one from a list of values using a listbox (Tk) or a combo box (wxPython).

**chooseFrom** if specified, simuOpt will choose one or more items from a list of values using a listbox (tk) or a combo box (wxPython).

**separator** if specified, a blue label will be used to separate groups of parameters.

**jump** it is used to skip some parameters when doing the interactive user input. For example, getParam will skip the rest of the parameters if -h is specified if parameter -h has item 'jump':-1 which means jumping to the end. Another situation of using this value is when you have a hierarchical parameter set. For example, if mutation is on, specify mutation rate, otherwise proceed. The value of this option can be the absolute index or the longarg name of another option.

**jumpIfFalse** The same as jump but jump if current parameter is False.

This function will first check command line argument. If the argument is available, use its value. Otherwise check if a config file is specified. If so, get the value from the config file. If both failed, prompt user to input a value. All input will be checked against types, if exists, an array of allowed types. Parameters of this function are:

**options** a list of option description dictionaries  
**doc** short description put to the top of parameter dialog  
**details** module help. Usually set to `__doc__`.  
**noDialog** do not use a parameter dialog, used in batch mode. Default to False.  
**checkUnprocessedArgs** obsolete because unused args are always checked.  
**verbose** whether or not print detailed info  
**nCol** number of columns in the parameter dialog.

**prettyOutput (value, quoted=False, outer=True)**

Return a value in good format, the main purpose is to avoid [0.90000001, 0.2].

**printConfig (opt, param, out=<open file '<stdout>', mode 'w' at 0xb7dbd068>)**

Print configuration.

**opt** option description list

**param** parameters returned from `getParam()`

**out** output

**requireRevision (rev)**

Compare the revision of this simuPOP module with given revision. Raise an exception if current module is out of date.

**saveConfig (opt, file, param)**

Write a configuration file. This file can be later read with command line option `-c` or `-config`.

**opt** the option description list

**file** output file

**param** parameters returned from `getParam`

**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, suppress banner information when simuPOP is loaded.

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

**usage (options, before="")**

Print usage information from the option description list. Used with `-h` (or `-help`) option, and in the parameter input dialog.

**options** option description list.

**before** optional information

**valueAnd (t1, t2)**

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

**valueBetween (a, b)**

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

**valueEqual (a)**

Return a function that returns true if passed option equals a

**valueGE (a)**

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

**valueGT (a)**

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

**valueIsList ()**

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

**valueIsNum ()**

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

**valueLE (a)**

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

**valueLT (a)**

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

**valueListOf (t)**

Return a function that returns true if passed option val is a list of type t. If t is a function (validator), check if all v in val pass t(v)

**valueNot (t)**

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

**valueNotEqual (a)**

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

**valueOneOf (t)**

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

**valueOr (t1, t2)**

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

**valueTrueFalse ()**

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

**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

## 4.2.2 Module `simuUtil`

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

### Module Functions

#### **CaseControl\_ChiSq (pop, sampleSize, penetrance=None)**

Draw affected sibpair sample from pop, run TDT using GENEHUNTER

**pop** simuPOP population. It can be a string if path to a file is given. This population must 1. have at least one ancestral generation (parental generation) 2. have a variable DSL (`pop.dvars().DSL`) indicating the Disease susceptibility loci. These DSL will be removed from the samples. 3. has only binary alleles

**pene** penetrance function, if not given (None), existing affectionstatus will be used.

**sampleSize** total sample size N. N/4 is the number of families to ascertain.

**keep\_temp** if True, do not remove sample data. Default to False.

#### **ChiSq\_test (pop)**

perform case control test

**pop** loaded population, or population file in simuPOP format. This function assumes that pop has two subpopulations, cases and controls, and have 0 as wildtype and 1 as disease allele. pop can also be an loaded population object.

**Return value** A list of p-value at each locus.

**Note** this function requires rpy module.

#### **ConstSize (size, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0)**

The population size is constant, but will split into numSubPop subpopulations at generation split

#### **ExponentialExpansion (initSize, endSize, end, burnin=0, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0)**

Exponentially expand population size from intiSize to endSize after burnin, split the population at generation split.

#### **InstantExpansion (initSize, endSize, end, burnin=0, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0)**

Instantaneously expand population size from intiSize to endSize after burnin, split the population at generation split.

#### **LOD\_gh (file, gh='gh')**

Analyze data using the linkage method of genehunter. Note that this function may not work under platforms other than linux, and may not work with your version of genehunter. As a matter of fact, it is almost unrelated to simuPOP and is provided only as an example how to use python to analyze data.

##### **Parameters**

**file** file to analyze. This function will look for file.dat and file.prein linkage format.

**loci** a list of loci at which p-value will be returned. If the list is empty, all p-values are returned.

**gh** name (or full path) of genehunter executable. Default to 'gh'

**Return value** A list (for each chromosome) of list (for each locus) of p-values.

#### **LOD\_merlin (file, merlin='merlin')**

run multi-point non-parametric linkage analysis using merlin

**LargePeds\_Reg\_merlin** (pop, sampleSize, qtrait=None, infoField='qtrait', merlin='merlin-regress', keep\_temp=False)

Draw affected sibpair sample from pop, run TDT using GENEHUNTER

**pop** simuPOP population. It can be a string if path to a file is given. This population must 1. have at least one ancestral generation (parental generation) 2. have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples. 3. has only binary alleles

**qtrait** a function to calculate quantitative trait

**infoField** information field to store quantitative trait. Default to 'qtrait'

**sampleSize** total sample size N. N/4 is the number of families to ascertain.

**merlin** executable name of merlin, full path name can be given.

**keep\_temp** if True, do not remove sample data. Default to False.

**LargePeds\_VC\_merlin** (pop, sampleSize, qtrait=None, infoField='qtrait', merlin='merlin', keep\_temp=False)

Draw affected sibpair sample from pop, run TDT using GENEHUNTER

**pop** simuPOP population. It can be a string if path to a file is given. This population must 1. have at least one ancestral generation (parental generation) 2. have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples. 3. has only binary alleles

**qtrait** a function to calculate quantitative trait

**infoField** information field to store quantitative trait. Default to 'qtrait'

**sampleSize** total sample size N. N/4 is the number of families to ascertain.

**merlin** executable name of merlin, full path name can be given.

**keep\_temp** if True, do not remove sample data. Default to False.

**LinearExpansion** (initSize, endSize, end, burnin=0, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0)

Linearly expand population size from intiSize to endSize after burnin, split the population at generation split.

**ListVars** (var, level=-1, name="", subPop=True, useWxPython=True)

list a variable in tree format, either in text format or in a wxPython window.

**var** any variable to be viewed. Can be a dw object returned by dvars() function

**level** level of display.

**name** only view certain variable

**subPop** whether or not display info in subPop

**useWxPython** if True, use terminal output even if wxPython is available.

**LoadFstat** (file, loci=[])

load population from fstat file 'file' since fstat does not have chromosome structure an additional parameter can be given

**MigrIslandRates** (r, n)

migration rate matrix  $\times m/(n-1) \ m/(n-1) \ \dots \ m/(n-1) \times \dots \dots \ m/(n-1) \ m/(n-1) \times$  where  $x = 1-m$

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

migration rate matrix, circular stepping stone model ( $X=1-m$ )  $\times \begin{matrix} m/2 & m/2 & m/2 & X & m/2 & 0 & 0 & m/2 & \times & m/2 \\ \dots & 0 & \dots & m/2 & 0 & \dots & m/2 & X \end{matrix}$  or non-circular  $\times \begin{matrix} m/2 & m/2 & m/2 & X & m/2 & 0 & 0 & m/2 & X & m/2 \\ \dots & 0 & \dots & m/2 & 0 & \dots & m/2 & X \end{matrix}$

```
QtraitSibs_Reg_merlin (pop, sampleSize, qtrait=None, infoField='qtrait',
    merlin='merlin-regress', keep_temp=False)
```

Draw affected sibpair sample from pop, run TDT using GENEHUNTER

**pop** simuPOP population. It can be a string if path to a file is given. This population must 1. have at least one ancestral generation (parental generation) 2. have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples. 3. has only binary alleles

**qtrait** a function to calculate quantitative trait

**infoField** information field to store quantitative trait. Default to 'qtrait'

**sampleSize** total sample size N. N/4 is the number of families to ascertain.

**merlin** executable name of merlin, full path name can be given.

**keep\_temp** if True, do not remove sample data. Default to False.

```
QtraitSibs_VC_merlin (pop, sampleSize, qtrait=None, infoField='qtrait',
    merlin='merlin', keep_temp=False)
```

Draw affected sibpair sample from pop, run TDT using GENEHUNTER

**pop** simuPOP population. It can be a string if path to a file is given. This population must 1. have at least one ancestral generation (parental generation) 2. have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples. 3. has only binary alleles

**qtrait** a function to calculate quantitative trait

**infoField** information field to store quantitative trait. Default to 'qtrait'

**sampleSize** total sample size N. N/4 is the number of families to ascertain.

**merlin** executable name of merlin, full path name can be given.

**keep\_temp** if True, do not remove sample data. Default to False.

```
Regression_merlin (file, merlin='merlin-regress')
```

run merlin regression method

```
SaveCSV (pop, output="", outputExpr="", fields=['sex', 'affection'], loci=[],
    combine=None, shift=1, **kwargs)
```

save file in CSV format

**fields** information fields, 'sex' and 'affection' are special fields that is treated differently.

**genotype** list of loci to output, default to all.

**combine** how to combine the markers. Default to None. A function can be specified, that takes the form

```
def func(markers) return markers[0]+markers[1]
```

**shift** since alleles in simuPOP is 0-based, shift=1 is usually needed to output alleles starting from allele 1. This parameter is ignored if combine is used.

```
SaveFstat (pop, output="", outputExpr="", maxAllele=0, loci=[], shift=1,
    combine=None)
```

# save file in FSTAT format

```
SaveLinkage (pop, output="", outputExpr="", loci=[], shift=1, combine=None,
    fields=[], recombination=1.0000000000000001e-05, penetrance=[0, 0.25,
    0.5], affectionCode=['1', '2'], pre=True, daf=0.001)
```

save population in Linkage format. Currently only support affected sibpairs sampled with affectedSibpairSample operator.

**pop** population to be saved. Must have ancestralDepth 1. paired individuals are sibs. Parental population are corresponding parents. If pop is a filename, it will be loaded.

**output** output.dat and output.ped will be the data and pedigree file. You may need to rename them to be analyzed by LINKAGE. This allows saving multiple files.

**outputExpr** expression version of output.

**affectionCode** default to '1'

**pre** True. pedigree format to be fed to makeped. Non-pre format it is likely to be wrong now for non-sibpair families.

**Note** the first child is always the proband.

```
SaveMerlinDatFile (pop, output="", outputExpr="", loci=[], fields=[],
outputAffection=False)
```

Output a .dat file readable by merlin

```
SaveMerlinMapFile (pop, output="", outputExpr="", loci=[])
```

Output a .map file readable by merlin

```
SaveMerlinPedFile (pop, output="", outputExpr="", loci=[], fields=[],
header=False, outputAffection=False, affectionCode=['U', 'A'],
combine=None, shift=1, **kwargs)
```

Output a .ped file readable by merlin

```
SaveQTDT (pop, output="", outputExpr="", loci=[], header=False,
affectionCode=['U', 'A'], fields=[], combine=None, shift=1, **kwargs)
```

save population in Merlin/QTDT format. The population must have pedindex, father\_idx and mother\_idx information fields.

**pop** population to be saved. If pop is a filename, it will be loaded.

**output** base filename.

**outputExpr** expression for base filename, will be evaluated in pop's local namespace.

**affectionCode** code for unaffected and affected. '1', '2' are default, but 'U', and 'A' or others can be specified.

**loci** loci to output

**header** whether or not put head line in the ped file.

**fields** information fields to output

**combine** an optional function to combine two alleles of a diploid individual.

**shift** if combine is not given, output two alleles directly, adding this value (default to 1).

```
SaveSolarFrqFile (pop, output="", outputExpr="", loci=[], calcFreq=True)
```

Output a frequency file, in a format readable by solar

**calcFreq** whether or not calculate allele frequency

```
Sibpair_LOD_gh (pop, sampleSize, penetrance=None, recRate=None, daf=None,
gh='gh', keep_temp=False)
```

Draw affected sibpair sample from pop, run Linkage analysis using GENEHUNTER

**pop** simuPOP population. It can be a string if path to a file is given. This population must 1. have at least one ancestral generation (parental generation) 2. have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples. 3. has only binary alleles

**pene** penetrance function, if not given (None), existing affection status will be used.

**sampleSize** total sample size N. N/4 is the number of families to ascertain.

**recRate** recombination rate, used in the Linkage file. If not given, `pop.dvars().recRate[0]` will be used. If there is no such variable, 0.0001 is used.

**daf** disease allele frequency. This is needed for the linkage format but I am not sure if it is used by TDT.

**gh** executable name of genehunter, full path name can be given.

**keep\_temp** if True, do not remove sample data. Default to False.

**Sibpair\_LOD\_merlin (pop, sampleSize, penetrance=None, merlin='merlin', keep\_temp=False)**

Draw affected sibpair sample from pop, run multi-point linkage analysis using merlin

**pop** simuPOP population. It can be a string if path to a file is given. This population must 1. have at least one ancestral generation (parental generation) 2. have a variable DSL (`pop.dvars().DSL`) indicating the Disease susceptibility loci. These DSL will be removed from the samples. 3. has only binary alleles

**pene** penetrance function, if not given (None), existing affectionstatus will be used.

**sampleSize** total sample size N. N/4 is the number of families to ascertain.

**merlin** executable name of merlin, full path name can be given.

**keep\_temp** if True, do not remove sample data. Default to False.

**Sibpair\_TDT\_gh (pop, sampleSize, penetrance=None, recRate=None, daf=None, gh='gh', keep\_temp=False)**

Draw affected sibpair sample from pop, run TDT using GENEHUNTER

**pop** simuPOP population. It can be a string if path to a file is given. This population must 1. have at least one ancestral generation (parental generation) 2. have a variable DSL (`pop.dvars().DSL`) indicating the Disease susceptibility loci. These DSL will be removed from the samples. 3. has only binary alleles

**pene** penetrance function, if not given (None), existing affectionstatus will be used.

**sampleSize** total sample size N. N/4 is the number of families to ascertain.

**recRate** recombination rate, used in the Linkage file. If not given, `pop.dvars().recRate[0]` will be used. If there is no such variable, 0.0001 is used.

**daf** disease allele frequency. This is needed for the linkage format but I am not sure if it is used by TDT.

**gh** executable name of genehunter, full path name can be given.

**keep\_temp** if True, do not remove sample data. Default to False.

**TDT\_gh (file, gh='gh')**

Analyze data using genehunter/TDT. Note that this function may not work under platforms other than linux, and may not work with your version of genehunter. As a matter of fact, it is almost unrelated to simuPOP and is provided only as an example how to use python to analyze data.

#### Parameters

**file** file to analyze. This function will look for file.dat and file.prein linkage format.

**loci** a list of loci at which p-value will be returned. If the list is empty, all p-values are returned.

**gh** name (or full path) of genehunter executable. Default to 'gh'

**Return value** A list (for each chromosome) of list (for each locus) of p-values.

**VC\_merlin (file, merlin='merlin')**

run variance component method

**file** file.ped, file.dat, file.map and file.mdl are expected. file can contain directory name.



**dataAggregator (self, maxRecord=0, recordSize=0)**

EXPERIMENTAL collect variables so that plotters can plot them all at once You can of course put it in other uses

**Usage** a = dataAggregator( maxRecord=0, recordSize=0)

**maxRecord** if more data is pushed, the old ones are discarded

**recordSize** size of record a.push(gen, data, idx=-1)

**gen** generation number

**data** one record (will set recordSize if the first time), or

**idx** if idx!=-1, set data at idx a.clear() a.range() # return min, max of all data a.data[i] # column i of the data a.gen # a.ready() # if all column has the same length, so data is ready

**Internal data storage** self.gen [ .... ] self.data column1 [ ..... ] column2 [ ..... ] ..... each record is pushed at the end of  
Initialization

**maxRecord** maxRecorddow size. I.e., maximum generations of data to keep

**saveFstat (output="", outputExpr="", \*\*kwargs)**

operator version of the function SaveFstat

**saveLinkage (output="", outputExpr="", \*\*kwargs)**

An operator to save population in linkage format

### 4.2.3 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.

#### Module Functions

**rmatrix (mat)**

Convert a Python 2d list to r matrix format that can be passed to functions like image directly.

**varPlotter (self, 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)**

Plotting with history plot a number in the form of a variable or expression, use

```
>>> varPlotter(var='expr')
```

plot a vector in the same window and there is only one replicate in the simulator, use

```
>>> varPlotter(var='expr', varDim=len)
```

where len is the dimension of your variable or expression. Each line in the figure represents the history of an item in the array. plot a vector in the same window and there are several replicates, use

```
>>> varPlotter(var='expr', varDim=len, numRep=nr, byRep=1)
```

varPlotter will try to use an appropriate layout for your subplots (for example, use 3x4 if numRep=10 ). You can also specify parameter mfrow to change the layout. if you would like to plot each item of your array variables in a subplot, use

```
>>> varPlotter(var='expr', varDim=len, byVal=1)
```

or in case of a single replicate

```
>>> varPlotter(var='expr', varDim=len, byVal=1, numRep=nr)
```

There will be numRep lines in each subplot. Use option `history=False` to plot with history. Parameters `byVal`, `varDim` etc. will be ignored. Other options are

**title**, **xtitle**, **ytitle** title of your figure(s). `title` is default to your expression, `xtitle` is defaulted to generation.

**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.

**update** update figure after update generations. This is used when you do not want to update the figure at every generation.

**saveAs** save figures in files `saveAs#gen.eps`. For example, if `saveAs='demo'`, you will get files `demo1.eps`, `demo2.eps` etc.

**separate** plot data lines in separate panels.

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

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

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

#### 4.2.4 Module `hapMapUtil`

Utility functions to manipulate HapMap data. These functions are provided as examples on how to load and evolve the HapMap dataset. They tend to change frequently so do not call these functions directly. It is recommended that you copy these function to your script when you need to use them.

##### Module Functions

```
evolveHapMap (pop, endingSize, gen, migr=<simuPOP_std.noneOp; proxy  
  of <Swig Object of type 'simuPOP::noneOp *' at 0xa9fdec8> >,  
  expand='exponential', mergeAt=10000, initMultiple=1, recIntensity=0.01,  
  mutRate=9.999999999999995e-08, step=10, keepParents=False,  
  numOffspring=1, recordAncestry=False)
```

Evolve and expand the hapmap population

**gen** total evolution generation

**initMultiple** copy each individual `initMultiple` times, to avoid rapid loss of genotype variation when population size is small.

**endingSize** ending population size

**expand** expanding method, can be linear or exponential

**mergeAt** when to merge population?

**gen** generations to evolve

**migr** a migrator to be used.

**recIntensity** recombination intensity

**mutRate** mutation rate

**step** step at which to display statistics

**keepParents** whether or not keep parental generations

**numOffspring** number of offspring at the last generation

**recordAncestry** whether or not calculate ancestry to an information fieldancestry. Only usable with two hapmap populations.

**getMarkersFromName** (HapMap\_dir, names, chroms=[], hapmap\_pops=[], minDiffAF=0, numMarkers=[])

Get population from marker names. This function returns a tuple with a population with found markers and names of markers that can not be located in the HapMap data. The returned population has three subpopulations, corresponding to CEU, YRI and JPT+CHB HapMap populations.

**HapMap\_dir** where HapMap data in simuPOP format is stored. The file should have been prepared by script-  
s/loadHapMap.py.

**names** names of markers. It can either be a straight list of names, or a dictionary of names categorized by chromosome number.

**chroms** a list of chromosomes to look in. If empty, all 22 autosomes will be tried. Chromosome index starts from 1. (1, ..., 22).

**hapmap\_pops** hapmap populations to load, can be a list of 'CEU', 'YRI' or 'JPT+CHB', or a list of 0, 1, 2. If empty (default), all three populations will be loaded.

**minDiffAF** minimal allele frequency difference between hapmap populations. If three subpopulations are loaded, use the maximal of three pair-wise allele frequency differences for comparison. This option is ignored if hapmap\_pops has length one.

**numMarkers** number of markers to use for each chromosome. Must have the same length as chroms.

**getMarkersFromRange** (HapMap\_dir, hapmap\_pops, chrom, startPos, endPos, maxNum, minAF=0, minDiffAF=0, minDist=0, maxDist=0)

Get a population with markers from given range

**HapMap\_dir** where HapMap data in simuPOP format is stored. The file should have been prepared by script-  
s/loadHapMap.py.

**hapmap\_pops** HapMap populations to load. It can be a list of 'CEU', 'YRI' or 'JPT+CHB', or a list of 0, 1, 2. If empty, all hapmap populations will be loaded.

**chrom** chromosome number (1-based index)

**startPos** starting position (in cM)

**endPos** ending position (in cM). If 0, ignore this parameter.

**maxNum** maximum number of markers to get. If 0, ignore this parameter.

**minAF** minimal minor allele frequency

**minDiffAf** minimal allele frequency between HapMap populations.

**minDist** minimal distance between two adjacent markers, in cM

**maxDist** maximum distance. If exceed, try to pick up a marker ASAP.

**sample1DSL** (pop, DSL, DA, pene, name, sampleSize)

Sample from the final population, using a single locus penetrance model.

**DSL** disease locus

**DA** disease allele

**pene** penetrance

**name** name of directory to save (it must exist)

**sampleSize** sample size, in this case, sampleSize/4 is the number of families

**sample2DSL** (pop, DSL, pene, name, size)

Sample from the final population, using a two locus penetrance model

**DSL** disease loci (two locus)

**pene** penetrance value, assuming a two-locus model

**name** name to save sample

**size** sample size



# INDEX

alleleType, 3  
ascertainment, 94  
  
bin format, 24  
  
carray, 5  
class  
    affectedSibpairSample, 97  
    affectionSplitter, 26  
    affectionTagger, 109  
    alphaMating, 38  
    baseOperator, 52  
    baseRandomMating, 34  
    binomialSelection, 34  
    caseControlSample, 96  
    cloneMating, 33  
    cloneOffspringGenerator, 44  
    combinedSplitter, 28  
    consanguineousMating, 37  
    continueIf, 111  
    dumper, 103  
    GenoStruTrait, 12  
    genotypeSplitter, 28  
    gsmMutator, 78  
    haplodiploidMating, 39  
    haplodiploidOffspringGenerator, 44  
    heteroMating, 40  
    ifElse, 112  
    individual, 29  
    infoEval, 106  
    infoExec, 107  
    infoParentsChooser, 42  
    infoSplitter, 26  
    infoTagger, 110  
    inheritTagger, 108  
    initByFreq, 69  
    initByValue, 70  
    initializer, 68  
    initSex, 69  
    kamMutator, 77  
    largePedigreeSample, 98  
    maPenetrance, 89  
    mapPenetrance, 88  
    mapQuanTrait, 91  
    mapSelector, 85  
    maQuanTrait, 92  
    maSelector, 85  
    mating, 32  
    mendelianOffspringGenerator, 45  
    mergeSubPops, 75  
    migrator, 72  
    mlPenetrance, 89  
    mlQuanTrait, 92  
    mlSelector, 86  
    monogamousMating, 36  
    mutator, 76  
    noMating, 33  
    noneOp, 114  
    nuclearFamilySample, 98  
    parentsTagger, 108  
    parentTagger, 108  
    pause, 115  
    pedigree, 62  
    pedigreeMating, 39  
    pedigreeParentsChooser, 43  
    penetrance, 87  
    pointMutator, 79  
    polygamousMating, 37  
    population, 14  
    proportionSplitter, 27  
    pyEval, 105  
    pyExec, 105  
    pyIndOperator, 68  
    pyInit, 71  
    pyMating, 40  
    pyMigrator, 74  
    pyMutator, 79  
    pyOperator, 65  
    pyOutput, 104  
    pyParentsChooser, 43  
    pyPenetrance, 90  
    pyQuanTrait, 93  
    pySample, 95  
    pySelector, 87

- pySubset, 95
- pyTagger, 110
- quanTrait, 91
- randomMating, 35
- randomParentChooser, 41
- randomParentsChooser, 41
- randomSample, 96
- rangeSplitter, 27
- recombinator, 80
- resizeSubPops, 76
- RNG, 7
- sample, 94
- savePopulation, 104
- selector, 84
- selfingOffspringGenerator, 44
- selfMating, 36
- sequentialParentChooser, 40
- sequentialParentsChooser, 41
- setAncestralDepth, 116
- sexSplitter, 26
- sexTagger, 109
- simulator, 54
- smmMutator, 78
- splitSubPop, 75
- spread, 71
- stat, 99
- stator, 99
- tagger, 107
- terminateIf, 111
- terminator, 110
- ticToc, 115
- turnOffDebug, 114
- turnOnDebug, 113
- constant
  - DuringMating, 56
  - PostMating, 56
  - PreMating, 56
  - PrePostMating, 56
- conversion, 80
- Function
  - migrIslandRates, 73
  - migrStepstoneRates, 73
  - SavePopulation, 23
  - turnOffDebug, 113
  - TurnOnDebug, 113
- function
  - AffectedSibpairSample, 97
  - CaseControlSample, 96
  - GsmMutate, 78
  - InitByFreq, 69
  - InitByValue, 70
  - InitSex, 69
  - KamMutate, 77
  - MaPenetrance, 89
  - MaQuanTrait, 92
  - MaSelect, 85
  - MapPenetrance, 88
  - MapQuanTrait, 91
  - MapSelector, 85
  - MergeSubPops, 75
  - MlPenetrance, 89
  - MlQuanTrait, 92
  - MlSelect, 86
  - PointMutate, 79
  - PyEval, 105
  - PyExec, 105
  - PyInit, 71
  - PyMutate, 79
  - PyPenetrance, 90
  - PyQuanTrait, 93
  - PySample, 95
  - PySelect, 87
  - PySubset, 95
  - RandomSample, 96
  - ResizeSubPops, 76
  - SmmMutate, 78
  - SplitSubPop, 75
  - Spread, 71
  - Stat, 99
  - TicToc, 115
  - TurnOffDebug, 114
  - TurnOnDebug, 113
  - infoEval, 106
  - infoExec, 107
  - LoadPopulation, 23
  - rng, 7
- GenoStruTrait
  - alleleName, 11
  - arrLociPos, 11
  - infoField, 11
  - infoFields, 11
  - locusPos, 11
  - maxAllele, 11
  - numChrom, 11
  - numLoci, 11
  - ploidy, 11
  - ploidyName, 11
  - sexChrom, 11
  - totNumLoci, 11
- genotypic structure, 11
- index
  - absolute, 5
  - relative, 5
- initializer, 68
- ListDebugCode, 3

- listDebugCode, 113
- loadSimulator, 57
- mating scheme, 32
- migrator, 72
- module
  - hapMapUtil, 128
  - simuOpt, 118
  - simuRPy, 127
  - simuUtil, 122
- Mutation, 76
- operator
  - DuringMating, 56
  - PostMating, 56
  - PreMating, 56
  - PrePostMating, 56
  - stat, 57
  - turnOffDebug, 113
  - turnOnDebug, 113
- penetrance, 87
- population, 13
  - individual, 29
  - population, 57
  - vars, 57
- quantitative trait, 91
- recombination, 80
- savePopulation, 23
- selection, 83
- SIMUALLELETYPE, 3
- Simulator, 54
- simulator
  - dryun, 56
- simuOpt, 3
- text format, 24
- xml format, 23