simuPOP Reference Manual

Release 0.8.8 (Rev: 1786)

Bo Peng

December 2004

Last modified
October 27, 2008

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

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

© 2004-2008 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

1 Introduction **1**
- 1.1 Loading simuPOP . . . 2
- 1.2 References and the `clone()` member function . . . 4
- 1.3 Zero-based indexes, ranges, absolute and relative indexes . . . 5
- 1.4 Function form of an operator . . . 5
- 1.5 The `carray` type . . . 6
- 1.6 Random Number Generator . . . 7
- 1.7 Name Conventions . . . 7
- 1.8 Online resources . . . 8

2 simuPOP Components **9**
- 2.1 Genotypic structure . . . 9
  - 2.1.1 Class `GenoStruTrait` . . . 10
- 2.2 Population . . . 11
  - 2.2.1 Class `population` . . . 12
  - 2.2.2 Ancestral populations . . . 21
  - 2.2.3 Save and Load a Population . . . 22
  - 2.2.4 View a population (GUI, wxPython required) . . . 22
- 2.3 Virtual subpopulations . . . 23
  - 2.3.1 Class `sexSplitter` . . . 24
  - 2.3.2 Class `affectionSplitter` . . . 24
  - 2.3.3 Class `infoSplitter` . . . 25
  - 2.3.4 Class `proportionSplitter` . . . 25
  - 2.3.5 Class `rangeSplitter` . . . 25
  - 2.3.6 Class `genotypeSplitter` . . . 25
  - 2.3.7 Class `combinedSplitter` . . . 26
- 2.4 Individuals . . . 26
  - 2.4.1 Access individuals from a population object . . . 26
  - 2.4.2 Access individual genotype . . . 27
  - 2.4.3 Class `individual` . . . 28
- 2.5 Mating Scheme . . . 31
  - 2.5.1 Class `mating` . . . 31
  - 2.5.2 Class `noMating` (Applicable to `all ploidy`) . . . 31
  - 2.5.3 Class `cloneMating` (Applicable to `all ploidy`) . . . 32
  - 2.5.4 Class `binomialSelection` (Applicable to `all ploidy`) . . . 32
  - 2.5.5 Class `baseRandomMating` (Applicable to `diploid only`) . . . 33
  - 2.5.6 Class `randomMating` (Applicable to `diploid only`) . . . 33
  - 2.5.7 Class `selfMating` (Applicable to `diploid only`) . . . 34
  - 2.5.8 Class `monogamousMating` (Applicable to `diploid only`) . . . 34

2.5.9 Class polygamousMating (Applicable to diploid only) . . . 34
2.5.10 Class consanguineousMating (Applicable to diploid only) . . . 35
2.5.11 Class alphaMating (Applicable to diploid only) . . . 35
2.5.12 Class haplodiploidMating (Applicable to haplodiploid only) . . . 36
2.5.13 Class pedigreeMating (Applicable to all ploid) . . . 36
2.5.14 Class pyMating (Applicable to all ploidy) . . . 36
2.5.15 Class heteroMating (Applicable to diploid only) . . . 37
2.5.16 Class sequentialParentChooser (Applicable to all ploidy) . . . 37
2.5.17 Class sequentialParentsChooser (Applicable to all ploidy) . . . 37
2.5.18 Class randomParentChooser (Applicable to all ploidy) . . . 37
2.5.19 Class randomParentsChooser (Applicable to all ploidy) . . . 38
2.5.20 Class infoParentsChooser (Applicable to all ploidy) . . . 38
2.5.21 Class pedigreeParentsChooser (Applicable to all ploidy) . . . 39
2.5.22 Class pyParentsChooser (Applicable to all ploidy) . . . 39
2.5.23 Class cloneOffspringGenerator (Applicable to all ploidy) . . . 39
2.5.24 Class selfingOffspringGenerator (Applicable to diploid only) . . . 39
2.5.25 Class haplodiploidOffspringGenerator (Applicable to haplodiploid only) . . . 40
2.5.26 Class mendelianOffspringGenerator (Applicable to diploid only) . . . 40
2.5.27 Determine the number of offspring during mating . . . 40
2.5.28 Determine offspring sex . . . 41
2.5.29 Determine subpopulation sizes of the next generation . . . 42
2.5.30 Demographic change functions . . . 42
2.5.31 Sex chromosomes . . . 43
2.5.32 Parent choosers and offspring generators . . . 43
2.5.33 Homogeneous and hybrid mating schemes . . . 44
2.5.34 Heterogeneous mating schemes . . . 45
2.6 Operators . . . 47
2.6.1 Class baseOperator . . . 47
2.7 Simulator . . . 48
2.7.1 Class simulator . . . 48
2.7.2 Generation number . . . 50
2.7.3 Operator calling sequence . . . 51
2.7.4 Save and Load . . . 51
2.8 Population variables . . . 51
2.8.1 vars() and dvars() functions . . . 52
2.8.2 Local namespace, pyEval and pyExec operators . . . 53
2.9 Information fields . . . 54
2.10 Pedigree . . . 56
2.10.1 Class pedigree . . . 57

3 Operator References **59**
3.1 Python operators . . . 59
3.1.1 Class pyOperator . . . 59
3.1.2 Class pyIndOperator . . . 61
3.2 Initialization . . . 62
3.2.1 Class initializer . . . 62
3.2.2 Class initSex (Function form: InitSex) . . . 63
3.2.3 Class initByFreq (Function form: InitByFreq) . . . 63
3.2.4 Class initByValue (Function form: InitByValue) . . . 63
3.2.5 Class spread (Function form: Spread) . . . 64
3.2.6 Class pyInit (Function form: PyInit) . . . 64
3.3 Migration . . . 65
3.3.1 Class migrator . . . 65

3.3.2 Functions (Python) MigrIslandRates, MigrSteppingStoneRates (simuUtil.py) 66
3.3.3 Class pyMigrator . . . 66
3.3.4 Class splitSubPop (Function form: SplitSubPop) . . . 67
3.3.5 Class mergeSubPops (Function form: MergeSubPops) . . . 68
3.3.6 Class resizeSubPops (Function form: ResizeSubPops) . . . 68
3.4 Mutation . . . 68
3.4.1 Class mutator . . . 68
3.4.2 Class kamMutator (Function form: KamMutate) . . . 69
3.4.3 Class smmMutator (Function form: SmmMutate) . . . 69
3.4.4 Class gsmMutator (Function form: GsmMutate) . . . 70
3.4.5 Class pyMutator (Function form: PyMutate) . . . 70
3.4.6 Class pointMutator (Function form: PointMutate) . . . 71
3.5 Recombination and gene conversion . . . 71
3.5.1 Class recombinator . . . 71
3.5.2 Gene conversion . . . 73
3.6 Selection . . . 74
3.6.1 Mechanism . . . 74
3.6.2 Class selector . . . 75
3.6.3 Class mapSelector (Function form: MapSelector, Applicable to all ploidy) . . . 76
3.6.4 Class maSelector (Function form: MaSelect) . . . 76
3.6.5 Class mlSelector (Function form: MlSelect) . . . 77
3.6.6 Class pySelector (Function form: PySelect) . . . 77
3.7 Penetrance . . . 78
3.7.1 Class penetrance . . . 78
3.7.2 Class mapPenetrance (Function form: MapPenetrance) . . . 79
3.7.3 Class maPenetrance (Function form: MaPenetrance) . . . 79
3.7.4 Class mlPenetrance (Function form: MlPenetrance) . . . 79
3.7.5 Class pyPenetrance (Function form: PyPenetrance) . . . 80
3.8 Quantitative Trait . . . 81
3.8.1 Class quanTrait . . . 81
3.8.2 Class mapQuanTrait (Function form: MapQuanTrait) . . . 81
3.8.3 Class maQuanTrait (Function form: MaQuanTrait) . . . 81
3.8.4 Class mlQuanTrait (Function form: MlQuanTrait) . . . 82
3.8.5 Class pyQuanTrait (Function form: PyQuanTrait) . . . 82
3.9 Ascertainment . . . 83
3.9.1 Class sample . . . 83
3.9.2 Class pySubset (Function form: PySubset) . . . 84
3.9.3 Class pySample (Function form: PySample) . . . 84
3.9.4 Class randomSample (Function form: RandomSample) . . . 84
3.9.5 Class caseControlSample (Function form: CaseControlSample) . . . 85
3.9.6 Class affectedSibpairSample (Function form: AffectedSibpairSample) . . . 85
3.9.7 Class largePedigreeSample . . . 86
3.9.8 Class nuclearFamilySample . . . 86
3.10 Statistics Calculation . . . 87
3.10.1 Class stator . . . 87
3.10.2 Class stat (Function form: Stat) . . . 87
3.11 Expression and Statements . . . 91
3.11.1 Class dumper . . . 91
3.11.2 Class savePopulation . . . 91
3.11.3 Class pyOutput . . . 92
3.11.4 Class pyEval (Function form: PyEval) . . . 92
3.11.5 Class pyExec (Function form: PyExec) . . . 93
3.11.6 Class infoEval (Function form: infoEval) . . . 93
3.11.7 Class infoExec (Function form: infoExec) . . . 93

3.12 Tagging (used for pedigree tracking) . . . 94
3.12.1 Class tagger . . . 94
3.12.2 Class inheritTagger . . . 94
3.12.3 Class parentTagger . . . 94
3.12.4 Class parentsTagger . . . 95
3.12.5 Class sexTagger . . . 95
3.12.6 Class affectionTagger . . . 96
3.12.7 Class infoTagger . . . 96
3.12.8 Class pyTagger . . . 96
3.13 Terminator . . . 96
3.13.1 Class terminator . . . 96
3.13.2 Class terminateIf . . . 97
3.13.3 Class continueIf . . . 97
3.14 Conditional operator . . . 97
3.14.1 Class ifElse . . . 97
3.15 Debug-related operators/functions . . . 98
3.15.1 Class turnOnDebug (Function form: TurnOnDebug) . . . 99
3.15.2 Class turnOffDebug (Function form: TurnOffDebug) . . . 99
3.16 Miscellaneous . . . 99
3.16.1 Class noneOp . . . 99
3.16.2 Class pause . . . 100
3.16.3 Class ticToc (Function form: TicToc) . . . 100
3.16.4 Class setAncestralDepth . . . 100

4 Global and Python Utility functions **103**
4.1 Global functions . . . 103
4.2 Utility Classes . . . 104
4.2.1 Class RNG . . . 104
4.3 Utility Modules . . . 105
4.3.1 Module simuOpt . . . 105
4.3.2 Module simuUtil . . . 108
4.3.3 Module simuRPy . . . 113
4.3.4 Module hapMapUtil . . . 114

Index **117**

LIST OF EXAMPLES

1.1 Getting help using the `help()` function . . . 1
1.2 Use of standard simuPOP modules . . . 2
1.3 Use of optimized simuPOP modules . . . 2
1.4 Use `simuOpt` to control which simuPOP 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 `InitByFreq` . . . 5
1.8 Usage of the `carray` type . . . 6
1.9 Random number generator . . . 7
2.1 Ancestral populations . . . 21
2.2 Save population variables . . . 22
2.3 Use simuViewPop to view a population . . . 22
2.4 Virtual subpopulation related functions . . . 24
2.5 Function population::individual() . . . 27
2.6 Function population::individuals() . . . 27
2.7 Virtual subpopulation related functions . . . 27
2.8 A generator function that mimicks random mating . . . 43
2.9 Save and load a simulator . . . 51
2.10 Population variables . . . 52
2.11 Local namespaces of populations . . . 53
2.12 Use of operators `pyEval` and `pyExec` . . . 54
2.13 Use regular information field function . . . 55
2.14 Use infoExec and infoEval operators . . . 56
3.1 Define a python operator . . . 60
3.2 Use of python operator . . . 61
3.3 Turn on/off debug information . . . 98

CHAPTER

ONE

Introduction

This reference manual assumes that you are reasonably familiar with the Python programming language. If you are new to Python, you may want to go through a few online tutorials and courses before you continue. Because this is a reference manual to all the features of simuPOP, it is recommended that you learn the basic concepts of simuPOP from the *simuPOP User's Guide* before getting lost in the details.

Most of the help information contained in this document is also available from command line. For example, after you install and import the simuPOP module, 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: three standard modules with short, long or binary alleles, respectively, and their optimized counterparts. 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. A standard module returns proper error messages, while an optimized module returns erroneous results and or simply 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 ?
    #
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 ?
    #
IndexError: src/population.h:462 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 optimied 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 *
>>> # make sure each run generates the same output to avoid unnecessary
>>> # documentation changes.
>>> rng().setSeed(12345)
>>> # 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 refernce to the exsting 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 referece 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* the `rep`th population 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 two simuPOP functions, namely `individual::genotype([p, [ch]])` and `population::genotype([sp])` 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. If you would like to copy the content of this `carray` to a Python list, use the `list()` function.

Example 1.8: Usage of the `carray` type

```
>>> # obtain an object using a genotype function
>>> pop = population(size=2, loci=[1,2])
>>> InitByValue(pop, [1,2,3])
>>> arr = pop.genotype()
>>> # print and expression (just like list)
>>> print arr
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> str(arr)
'[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]'
>>> # count
>>> arr.count(2)
4
>>> # index
>>> arr.index(2)
1
>>> # can read write
>>> arr[0] = 0
>>> # the underlying locus position is also changed
>>> print pop.genotype()
[0, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> # convert to list
>>> arr.tolist()
[0, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> # or simply
>>> list(arr)
[0, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> # compare to list directly
>>> arr == [0, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
True
>>> # you can also convert and compare
>>> list(arr) == [0, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
True
>>> # slice
>>> arr[:] = [3,2,1]*4
>>> # assign from another part
>>> arr[1:3] = arr[3:5]
>>>
```

1.6 Random Number Generator

There are many random number generators (RNG) with different properties. Using a bad RNG can serious 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.

When simuPOP is loaded, it creates a default random number generator of type `mt19937`. This RNG gets its seed from a system random number generator that gurantees different random seeds for all instances of simuPOP even if they are initialized at the same time. After simuPOP is loaded, you can reset this system RNG with a different RNG or re-initialize it with a different seed. Random seed of the current session can be retrieved through function `rng().seed()`.

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 the `RNG(name, seed)` function. Using a single source of random number generator through the `rng()` function allows the whole simulation to be repeated if its original random seed is reused.

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 seprate RNG instance
>>> r=RNG()
>>> for n in range(1,5):
...     print r.randBinomial(10, .7),
...
8 7 6 8
>>>
```

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), 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.

CHAPTER

TWO

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 names and number of chromosomes (c.f. `numChrom(), chromName()`)
- 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 acoordingly.

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.

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. SimuPOP::GenoStruTrait::GenoStruTrait

class `GenoStruTrait``()`

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

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

`alleleNames``()`
Return an array of allele names

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

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

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

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

`chromName`(*chrom*)
Return the name of an chrom

`chromNames``()`
Return an array of chrom names

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

`genoSize``()`
Return the total number of loci times ploidy

`haplodiploid``()`
SimuPOP::GenoStruTrait::haplodiploid

`hasInfoField`(*name*)
Determine if an information field exists

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

`infoFields``()`
Return an array of all information fields

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

infoSize()
Obtain the number of information fields

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

lociCovered(*loc, dist*)
The result will be at least 1, even if dist = 0.

lociDist(*loc1, loc2*)
`LOC2`.

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

lociNames()
Return names of all loci

lociPos()
Return loci positions

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

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

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

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`.

numChrom()
Return the number of chromosomes

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

numLoci()
Return the number of loci on all chromosomes

ploidy()
Return ploidy, the number of homologous sets of chromosomes

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

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

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.

A simuPOP population consists of individuals of the same genotypic structure, which refers to the number of chromosomes, numbers and positions of loci on each chromosome etc. The most important components of a population are:

- 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 subpopualtion 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.

Create a population object with given size and genotypic structure.

class `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 chromosme 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 intepreted 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

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`)

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.

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.

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 alreay exist, they will be re-initialized.

init: Initial value for the new fields.

ancestor(*ind, gen*)

Refrence 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).

ancestor(*ind, subPop, gen*)

Refrence to an individual `ind` in a specified subpopulaton 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).

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()`.

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.

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.

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.

evaluate(*expr="", stmts=""*)

Evaluate a Python statment/expression in the population's local namespace

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

execute(*stmts=""*)

Execute a statement (can be a multi-line string) in the population's local namespace

fitSubPopStru(*newSubPopSizes*)

Of the population will be cleared.

gen()

Current generation during evolution

genotype()

Get an editable array of the genotype of all individuals in a population.

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.

genotype(*subPop*)

Get an editable array of the genotype of all individuals in a subpopulation.

subPop: Index of subpopulation (start from 0)

hasVirtualSubPop()

If a population has any virtual subpopulation

indBegin (*subPop*)
The iterator will skip invisible individuals.

indEnd ()
It is recommended to use it.valid(), instead of it != indEnd().

indEnd (*subPop*)
It is recommended to use it.valid(), instead of it != indEnd(sp).

indInfo (*idx*)
Get information field `idx` of all individuals

idx: Index of the information field

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

name: Name of the information field

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

idx: Index of the information field

subPop: Subpopulation index

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

name: Name of the information field

subPop: Subpopulation index

individual (*ind, subPop=0*)
Refernce to individual `ind` in subpopulation `subPop`

This function is named `individual` in the Python interface.

ind: Individual index within `subPop`

subPop: Subpopulation index

individuals ()
Return an iterator that can be used to iterate through all individuals

Typical usage is

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

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

individuals (*subPop, virtualSubPop*)
SimuPOP::population::individuals

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.

insertAfterLocus (*idx, pos, name=string*)
Append an locus after a given position

`insertAfterLocus(idx, pos, name)` is a shortcut to `insertAfterLoci([idx], [pos], [name])`.

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.

insertBeforeLocus (*idx, pos, name=string*)

Insert an locus before a given position

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

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 themselfs
- 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

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`.

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 popualtions 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.

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`).

newPopByIndID (*keepAncestralPops=-1, id=[], removeEmptySubPops=False*)

Form a new population according to individual subpopulation ID. Individuals with negative subpopulation ID will be removed.

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)
```

numSubPop ()

Number of subpopulations in a population.

numVirtualSubPop ()

Number of virtual subpopulations.

popSize ()

Total population size

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.

rawIndBegin (*subPop*)

The iterator will skip invisible individuals.

rawIndEnd ()

It is recommended to use it.valid(), instead of it != indEnd().

rawIndEnd (*subPop*)

It is recommended to use it.valid(), instead of it != indEnd(sp).

removeEmptySubPops ()

Remove empty subpopulations by adjusting subpopulation IDs

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.

removeLoci (*remove=[], keep=[]*)

Remove some loci from the current population. Only one of the two parameters can be specified.

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.

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 achive order 3,2,0,1, the rank should be 1 0 2 3.

rep ()

Current replicate in a simulator which is not meaningful for a stand-alone population

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.

savePopulation (*filename, format="", compress=True*)

Save population to a file

compress: Obsolete parameter

filename: Save to filename

format: Obsolete parameter

selectionOn ()

Selection is on at any subpopulation?

setAncestralDepth (*depth*)

Set ancestral depth

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

setGenotype (*geno*)

Set genotype to all individuals of a population.

geno: Genotype to be set. It will be reused if its length is less than the genotype length of the population, which is popSize()*ploidy()*totNumLoci().

setGenotype (*geno, subPop*)

Set genotype to all individuals in a subpopulation.

geno: Genotype to be set. It will be reused if its length is less than the genotype length of the population, which is subPopSize(subPop)*ploidy()*totNumLoci().

subPop: Index of subpopulation (start from 0)

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.

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))`.

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, resprenting subpopulation ID of each individual. If the length of is less than population size, it is repeated to fill the whole population.

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.

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 informaton 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 indiviudals referred by `off1` and from geneartion 0 (`pathGen`[3]) 7. Save index of these individuals to information fields `cousin1` and `cousin2` at genearation `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.

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.

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!

setSubPopStru (*newSubPopSizes*)

Set population/subpopulation structure given subpopulation sizes

newSubPopSizes: An array of new subpopulation sizes. The overall population size should not changed.

setVirtualSplitter(*vsp*)
SimuPOP::population::setVirtualSplitter

vsp: A virtual subpop splitter

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.

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.

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

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

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

subPopSize(*subPop*)
Return size of a subpopulation `subPop`.

subPop: Index of subpopulation (start from 0)

subPopSizes()
Return an array of all subpopulation sizes.

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

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.

turnOnSelection()
Turn on selection for all subpopulations.

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).

validate(*msg*)
Evolution

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

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

id: Subpopulation id

vid: Virtual subpopulation id

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     0.179400     0.025350
1     0.167850     0.025300
2     0.160850     0.024450
3     0.152850     0.022650
4     0.147700     0.020600
5     0.138450     0.017800
>>> # restore to the current generation
>>> pop.useAncestralPop(0)
>>>
```

2.2.3 Save and Load a Population

Internally, population can be saved to or loaded from disk files using `savePopulation(file)` member function, global `SavePopulation(pop, file)` and `LoadPopulation`. (Yes, it is `Load..` not `load..` because `savePopulation` is a member function and `LoadPopulation` is a global function.). Although files in any extension can be saved/loaded correctly, extension `.pop` is usually used. Populations are compressed in gzip format to save some disk space.

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().diseaseLoci = [4, 30]
SavePopulation(pop, 'example.pop')
```

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 subpoulation, 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 schems. 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 genetypes 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 provies 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 subpoulation. Operator `stat(popSize=True)` also calculates virtual subpoulation sizes, and save them in a variable `virtualPopSize`.

Example 2.7 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)
137
>>> 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)
156
>>> pop.virtualSubPopSize(0, 1)
641
>>>
>>> for ind in pop.individuals(0, 0):
...     assert ind.allele(2, 0) == 0 and ind.allele(2, 1) == 1
...
>>>
```

2.3.1 Class `sexSplitter`

Split the population into Male and Female virtual subpopulations SimuPOP::sexSplitter::sexSplitter

class `sexSplitter`()

`clone`()
SimuPOP::sexSplitter::clone

`name`(*sp*)
Name of a virtual subpopulation

`numVirtualSubPop`()
Number of virtual subpops of subpopulation sp

2.3.2 Class `affectionSplitter`

Split a subpopulation into unaffected and affected virtual subpopulations. SimuPOP::affectionSplitter::affectionSplitter

class `affectionSplitter`()

`clone`()
SimuPOP::affectionSplitter::clone

`name`(*sp*)
Name of a virtual subpopulation

`numVirtualSubPop`()
Number of virtual subpops of subpopulation sp

2.3.3 Class `infoSplitter`

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. SimuPOP::infoSplitter::infoSplitter

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

cutoff: A list of cutoff values. For example, cutoff=[1, 2] defines three virtual subpopulations with v < 1, 1 <= v < 2, and v >= 2.

info: Name of the information field

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

`name` (*sp*)
Name of a virtual subpopulation

`numVirtualSubPop` ()
Number of virtual subpops of subpopulation sp

2.3.4 Class `proportionSplitter`

Split the population according to a proportion SimuPOP::proportionSplitter::proportionSplitter

class `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.

`clone` ()
SimuPOP::proportionSplitter::clone

`name` (*sp*)
Name of a virtual subpopulation

`numVirtualSubPop` ()
Number of virtual subpops of subpopulation sp

2.3.5 Class `rangeSplitter`

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

class `rangeSplitter` (*ranges*)

range: A shortcut for ranges=[range]

ranges: A list of ranges

`clone` ()
SimuPOP::rangeSplitter::clone

`name` (*sp*)
Name of a virtual subpopulation

`numVirtualSubPop` ()
Number of virtual subpops of subpopulation sp

2.3.6 Class `genotypeSplitter`

Split the population according to given genotype

class 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.

clone ()

SimuPOP::genotypeSplitter::clone

name (*sp*)

Name of a virtual subpopulation

numVirtualSubPop ()

Number of virtual subpops of subpopulation sp

2.3.7 Class `combinedSplitter`

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 fouth (index 3) and fifth (index 4) of the combined splitter. SimuPOP::combinedSplitter::combinedSplitter

class combinedSplitter (*splitters=[]*)

clone ()

SimuPOP::combinedSplitter::clone

name (*sp*)

Name of a virtual subpopulation

numVirtualSubPop ()

Number of virtual subpops of subpopulation sp

2.4 Individuals

2.4.1 Access individuals from a population object

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 subpopulaiton 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()
```

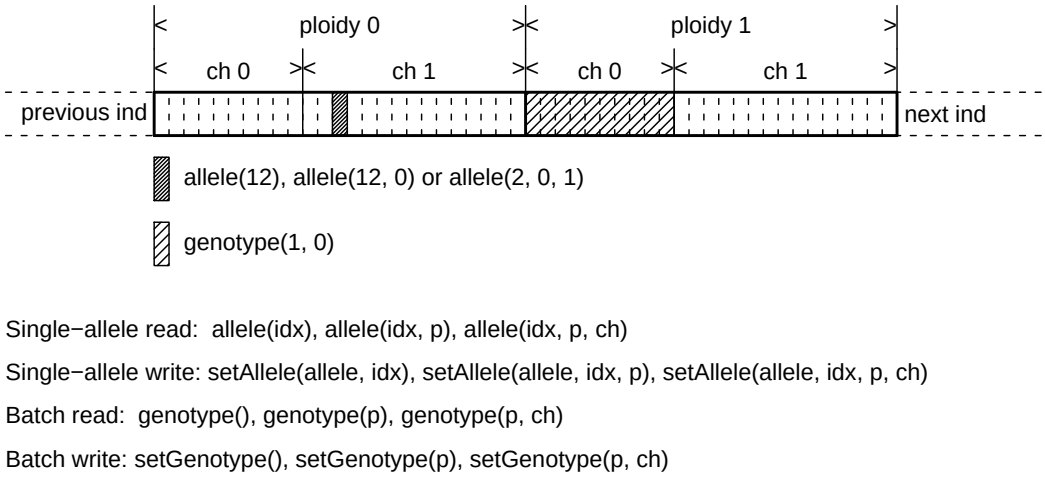
Example 2.6: Function population::individuals()

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

2.4.2 Access individual genotype

simuPOP provies several functions to read/write individual genotype. It is important to understand how simuPOP store these genotypes before you use these functions. Regardless of internal implementation, you can consider the genotype of an individual as an array of alleles, ordered by marker, chromosome and ploidy. This is illustrated in Figure 2.1.

Figure 2.1: Memory layout of individual genotype



It is worth noting that, instead of copying genotypes of an individual to a Python tuple or list, the return value of function `genotype([p, [ch]])` is a special python carray object that reflects the underlying genotypes. Modifying elements of this array will change the genotype of an individual directly. This is demonstrated in the following example.

Example 2.7: Virtual subpopulation related functions

```
>>> pop = population(size=[3, 2], loci=[2])
```

```
>>> # single allele access
>>> for ind in pop.individuals(1):
...     for marker in range(pop.totNumLoci()):
...         ind.setAllele(marker % 2, marker, 0)
...         ind.setAllele(marker % 2, marker, 1)
...         print '%d %d ' % (ind.allele(marker, 0), ind.allele(marker, 1))
...
0 0
1 1
0 0
1 1
>>> # batch access
>>> ind = pop.individual(4)
>>> geno = ind.genotype()
>>> print geno
[0, 1, 0, 1]
>>> geno[2] = 3
>>> print ind.genotype()
[0, 1, 3, 1]
>>> # direct modification of the underlying genotype
>>> geno[2:4] = [3, 4]
>>> print ind.genotype()
[0, 1, 3, 4]
>>> # set genotype
>>> ind.setGenotype([2, 1])
>>> print geno
[2, 1, 2, 1]
>>> # print genotypes of all individuals in the second subpopulation.
>>> print pop.genotype(1)
[0, 1, 0, 1, 2, 1, 2, 1]
>>>
```

2.4.3 Class `individual`

Individuals with genotype, affection status, sex etc.

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. SimuPOP::individual::individual

class `individual``()`

affected()
Whether or not an individual is affected

affectedChar()
Return `A` (affected) or `U` (unaffected) for affection status

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

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

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

index: Index from the begining of the `p-th` set of the chromosomes, ranging from `0` to `totNumLoci()`

p: Index of the ploidy

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 begining of chromosome `ch` of ploidy `p`, ranging from `0` to `numLoci(ch)`

p: Index of the polidy

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

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

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

genotype()
Return an editable array (a carray of length `totNumLoci()*ploidy()`) of genotypes of an individual.

genotype(*p*)
Return an editable array of alleles of the `p-th` copy of the chromosomes

p: Index of the ploidy

genotype(*p, ch*)
Return an editable array of alleles of the `ch-th` chromosome in the `p-th` chromosome set

ch: Index of the chromosome in ploidy `p`

p: Index of the ploidy

info(*idx*)
Get information field `idx`

idx: Index of the information field

info(*name*)
Get information field `name`

Equivalent to `info(infoIdx(name))`.

name: Name of the information field

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

idx: Index of the information field

intInfo(*name*)
Get information field `name` as an integer

Equivalent to `int(info(name))`.

name: Name of the information field

setAffected(*affected*)
Set affection status

setAllele(*allele, index*)
Set the allele at locus `index`

allele: Allele to be set

index: Index from the begining of genotype, ranging from `0` to `totNumLoci()*ploidy()`

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 begining of the ploidy `p`, ranging from `0` to `totNumLoci(p)`

p: Index of the ploidy

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 begining of the chromosome, ranging from `0` to `numLoci(ch)`

p: Index of the ploidy

setGenotype(*geno*)
Set the genotype of an individual

geno: Genotype to be set. It will be reused if its length is less than the genotype length of the individual.

setGenotype(*geno, p*)
Set the genotype of the `p-th` copy of the chromosomes

geno: Genotype to be set. It will be reused if its length is less than the total number of loci.

p: Index of the ploidy

setGenotype(*geno, p, ch*)
Set the genotype of the `ch-th` chromosome in the `p-th` chromosome set

ch: Index of the chromosome in ploidy `p`

geno: Genotype to be set. It will be reused if its length is less than the number of loci on chromosome `ch`.

p: Index of the ploidy

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

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

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

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

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

sexChar()
Return the sex of an individual, `M` or `F`

subPopID()
Return the ID of the subpopulation to which this individual blongs

Note: `subPopID` is not set by default. It only corresponds to the subpopulation in which this individual resides after `pop::setIndSubPopID` is called.

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`

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.

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

class `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 detemine 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 virutal subpopulation has non-zero weight, this virtual subpopulation will produce no offspring (weight 0).
- any negative number -n: the size will be n*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.

`clone` ()

Deep copy of a mating scheme

`submitScratch` (*pop, scratch*)

A common submit procedure is defined.

2.5.2 Class `noMating` (Applicable to `all ploidy`)

A mating scheme that does nothing

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. Creat a scheme with no mating

class 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!

clone()

Deep copy of a scheme with no mating

2.5.3 Class `cloneMating` (Applicable to `all ploidy`)

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

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.

Create a binomial selection mating scheme

class 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.

clone()

Deep copy of a binomial selection mating scheme

2.5.4 Class `binomialSelection` (Applicable to `all ploidy`)

A mating scheme that uses binomial selection, regardless of sex

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.

Create a binomial selection mating scheme

class 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.

clone()

Deep copy of a binomial selection mating scheme

2.5.5 Class `baseRandomMating` (Applicable to `diploid only`)

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, polygemous 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`, `monogemousMating`, `polygemousMating`, `alphaMating` are all special cases of this mating scheme. They should be used whenever possible. SimuPOP::baseRandomMating::baseRandomMating

class 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.

clone()

Deep copy of a random mating scheme

2.5.6 Class `randomMating` (Applicable to `diploid only`)

A mating scheme of basic sexually random mating

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.

class 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`.

clone()

Deep copy of a random mating scheme

2.5.7 Class `selfMating` (Applicable to `diploid only`)

A mating scheme of selfing

In this mating scheme, a parent is choosen 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. Create a self mating scheme

class 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`.

clone()

Deep copy of a self mating scheme

2.5.8 Class `monogamousMating` (Applicable to `diploid only`)

A mating scheme of monogamy

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.

class 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 are exhausted. Default to False. Please refer to class `mating` for descriptions of other parameters.

clone()

Deep copy of a random mating scheme

2.5.9 Class `polygamousMating` (Applicable to `diploid only`)

A mating scheme of polygymy or polyandry

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. SimuPOP::polygamousMating::polygamousMating

class 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.

`clone``()`

Deep copy of a random mating scheme

2.5.10 Class `consanguineousMating` (Applicable to `diploid only`)

A mating scheme of consanguineous mating

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

class `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.

`clone``()`

Deep copy of a consanguineous mating scheme

2.5.11 Class `alphaMating` (Applicable to `diploid only`)

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

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.

class `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 indiviudals 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.

clone()

Deep copy of a random mating scheme

2.5.12 Class `haplodiploidMating` (Applicable to `haplodiploid only`)

Haplodiploid mating scheme of many hymemopterans

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.

class 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.

clone()

Deep copy of a random mating scheme

2.5.13 Class `pedigreeMating` (Applicable to `all ploid`)

A mating scheme that follows a given pedigree

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.

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

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

clone()

Deep copy of a random mating scheme

2.5.14 Class `pyMating` (Applicable to `all ploidy`)

A Python mating scheme

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. Create a Python mating scheme

class 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.

clone()

Deep copy of a Python mating scheme

2.5.15 Class `heteroMating` (Applicable to `diploid only`)

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

class 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 specifc virtual subpopulations.

clone()

Deep copy of a Python mating scheme

2.5.16 Class `sequentialParentChooser` (Applicable to `all ploidy`)

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

class sequentialParentChooser()

clone()

SimuPOP::sequentialParentChooser::clone

2.5.17 Class `sequentialParentsChooser` (Applicable to `all ploidy`)

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

class sequentialParentsChooser()

clone()

SimuPOP::sequentialParentsChooser::clone

2.5.18 Class `randomParentChooser` (Applicable to `all ploidy`)

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`, paremeter `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. SimuPOP::randomParentChooser::randomParentChooser

class 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.

clone()

SimuPOP::randomParentChooser::clone

2.5.19 Class `randomParentsChooser` (Applicable to `all ploidy`)

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.

class 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 indiviudals 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.

clone()

SimuPOP::randomParentsChooser::clone

2.5.20 Class `infoParentsChooser` (Applicable to `all ploidy`)

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. SimuPOP::infoParentsChooser::infoParentsChooser

class 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.

clone()
SimuPOP::infoParentsChooser::clone

2.5.21 Class `pedigreeParentsChooser` (Applicable to `all ploidy`)

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

class pedigreeParentsChooser(*ped*)

clone()
SimuPOP::pedigreeParentsChooser::clone

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

2.5.22 Class `pyParentsChooser` (Applicable to `all ploidy`)

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. SimuPOP::pyParentsChooser::pyParentsChooser

class pyParentsChooser(*parentsGenerator*)

parentsGenerator: A Python generator function

clone()
SimuPOP::pyParentsChooser::clone

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

2.5.23 Class `cloneOffspringGenerator` (Applicable to `all ploidy`)

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

class 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.

clone()
SimuPOP::cloneOffspringGenerator::clone

2.5.24 Class `selfingOffspringGenerator` (Applicable to `diploid only`)

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. SimuPOP::selfingOffspringGenerator::selfingOffspringGenerator

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

clone()

SimuPOP::selfingOffspringGenerator::clone

2.5.25 Class `haplodiploidOffspringGenerator` (Applicable to `haplodiploid only`)

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

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

clone()

SimuPOP::haplodiploidOffspringGenerator::clone

copyParentalGenotype(*parent, it, ploidy, count*)

SimuPOP::haplodiploidOffspringGenerator::copyParentalGenotype

2.5.26 Class `mendelianOffspringGenerator` (Applicable to `diploid only`)

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. SimuPOP::mendelianOffspringGenerator::mendelianOffspringGenerator

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

clone()

SimuPOP::mendelianOffspringGenerator::clone

formOffspringGenotype(*parent, it, ploidy, count*)

Does not set sex if count == -1.

count: Index of offspring, used to set offspring sex

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 determins the number of offpspring to use.
- `MATE_GeometricDistribution`: `numOffspring` is considered as p for a geometric distribution. The number of offspring for each mating is determined by

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

- `MATE_PoissonDistribution`: `numOffspring` is considered as p for a Poission distribution. The number of offspring for each mating is determined by

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

  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=`maxNumOffspring`, the number of offspring for each mating is determined by

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

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

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

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 `intiSize` 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 `intiSize` 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)
```

Instaneously expand the population size from `intiSize` 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 `pyParensChooser` 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 2.8: 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(),
...
(227, 480) (156, 794) (150, 704) (239, 840) (761, 340)
>>> rc2 = randomChooser(pop, 1)
>>> for i in range(5):
...     print rc2.next(),
...
(93, 88) (172, 0) (122, 63) (33, 91) (40, 145)
>>>
```

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 2.8, 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.

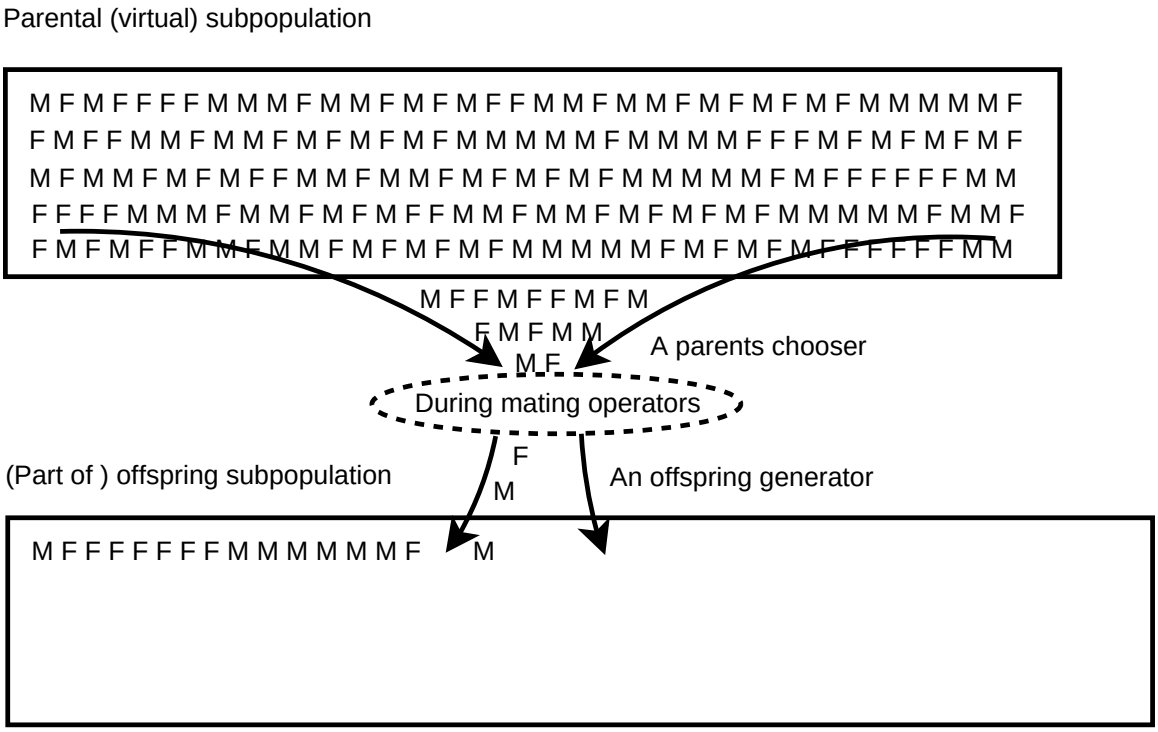
A parents chooser can yield a pair of parents, or a single parent. Obviously, a single diploid parent can not produce offspring using the usual Medelian fashion, so here comes another concept: *offspring generator*, which determines how to produce offspring from given parent or parents. Currently, there 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 homogenou mating scheme is illustrated in Figure

Figure 2.2: A homogeneous mating scheme



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())
```

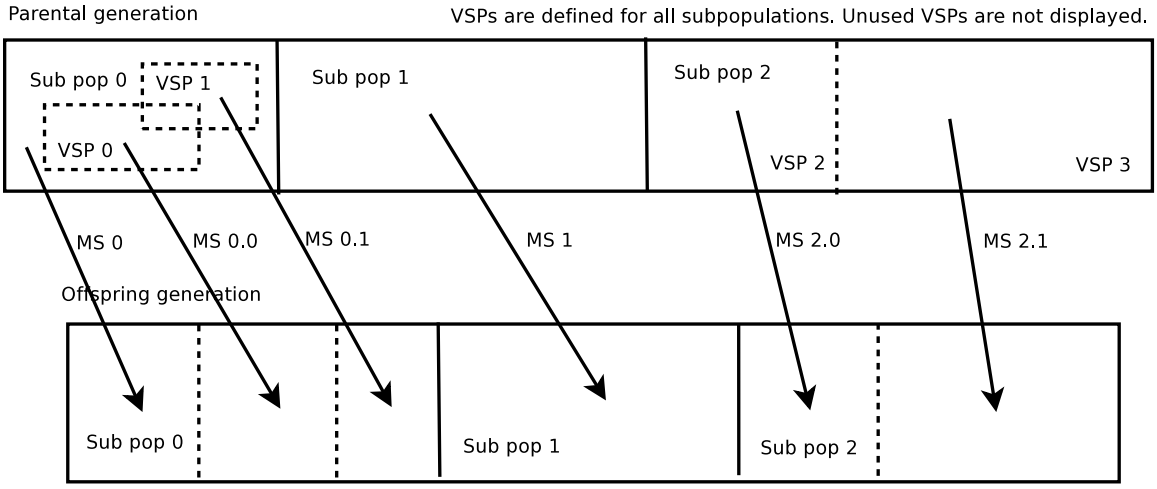
The later simply copy everyone from the parental to the offspring genertion.

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 2.3.

Figure 2.3: 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 schems 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 $\frac{1}{6}N$, $\frac{2}{6}N$, $\frac{1}{3}N$ 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

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

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

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` 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.

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

`clone` ()
Deep copy of an operator

`diploidOnly` ()
Determine if the operator can be applied only for diploid population

`haploidOnly` ()
Determine if the operator can be applied only for haploid population

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

`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

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. Create a simulator

class simulator (*pop, matingScheme, stopIfOneRepStops=False, applyOpToStoppedReps=False, rep=1*)

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

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.

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

addInfoFields (*fields, init=0*)

Add information fields to all replicates

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

clone ()

Deep copy of a simulator

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 opertors
- 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 begining 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 genertion is the -1 genertion if no ending genertion is given?

gen()

Return the current generation number

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

numRep()

Return the number of replicates

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 refered population will be invalid after another round of evolution. If you would like to get a persistent population, please use `getPopulation(rep)`.

saveSimulator(*filename, format="", compress=True*)

Save simulator in `'txt'`, `'bin'` or `'xml'` format

compress: Obsolete parameter

filename: Filename to save the simulator. Default to `simu`.

format: Obsolete parameter

setAncestralDepth(*depth*)

Set ancestral depth of all replicates

setGen(*gen*)

Set the current generation. Usually used to reset a simulator.

gen: New generation index number

setMatingScheme(*matingScheme*)

Set a new mating scheme

step(*ops=[], preOps=[], postOps=[], steps=1, dryrun=False*)

Evolve `steps` generation

vars(*rep, subPop=-1*)

Return the local namespace of population `rep`, 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` parameter). 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. Although files with any extension can be correctly saved/loaded, extension `.sim` is usually used. Note that a mating scheme can not be saved and has to be re-specified in `LoadSimulator()`.

Example 2.9: Save and load a simulator

```
>>> simu.saveSimulator("s.sim")
>>> simu1 = LoadSimulator("s.sim", randomMating())
>>>
```

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):

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

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.allaleFreq[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 2.10: 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
>>> 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]        0.2005
     [1]        0.7995
>>> # print number of allele 1 at locus 0
>>> print pop.vars()['alleleNum'][0][1]
4797
>>> print pop.dvars().alleleNum[0][1]
4797
>>> print pop.dvars().alleleFreq[0]
[0.20050000000000001, 0.79949999999999999]
>>> print pop.dvars(1).alleleNum[0][1]
3223
>>>
```

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::evalulate()` or `population::execute()`. For example:

Example 2.11: 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
 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 ?
    #
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}
>>> # 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}
>>>
```

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 2.12: 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 31, myNum 62
gen 0, rep 1, num 29, myNum 58
gen 1, rep 0, num 39, myNum 78
gen 1, rep 1, num 33, myNum 66
gen 2, rep 0, num 47, myNum 94
gen 2, rep 1, num 42, myNum 84
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 carray 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.setInfIndo(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 2.13: 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 varaibles being the information fields of individuals. Changes to these variables will change the corresponding information fields of an individul. 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 poulation's local namespace can be made available, using option `usePopVars=True`.

Example 2.14: 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)")
(7, 21) (8, 24) (9, 27) (4, 12) (7, 21) >>>
>>> # this is wrong because 'c' is not available
>>> InfoExec(pop, 'b=c+a')
Traceback (most recent call last):
  File "<embed>", line 1, in ?
NameError: name 'c' is not defined
Traceback (most recent call last):
  File "refManual.py", line 1, in ?
    #
  File "/usr/lib64/python2.4/site-packages/simuPOP_la.py", line 11831, in InfoExec
    infoExec(*args, **kwargs).apply(pop)
SystemError: Evalulation 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')
(13.0, 14.0, 15.0, 10.0, 13.0)
>>>
```

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 auxillary 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.

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()) Will be loaded from this file.

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

`addGen` (*subPopSize*)
For the new generation.

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

`clone` ()
Make a copy of this pedigree.

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

`gen` ()
Return the number of generations of this pedigree.

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

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

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

`load` (*filename*)
`PARENTSTAGGER`. The format is described in the simuPOP reference manual

`loadInfo` (*filename, name*)
`AFFECTIONTAGGER`, `pyTagger` and `infoTagger`.

`markUnrelated` ()
Last generation (not marked by selectIndividuals) from the pedigree.

mother (*gen, idx*)
The returned index is the absolute index of mother in the parental generation.

numParents ()
Return the number of parents for each individual

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

removeUnrelated (*adjust_index=True*)
WARNING: if adjust_index=false, an invalid pedigree will be generated.

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

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

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

selectIndividuals (*inds*)
`REMOVEUNRELATED` function will remove these individuals from the pdeigree

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

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

setInfo (*info, gen, idx, name*)
Set information `name` of individual `idx` at generation `gen`.

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

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

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

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

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

CHAPTER

THREE

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.

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

class `pyOperator` (*func, param=None, stage=PostMating, formOffGenotype=False, passOffspringOnly=False, begin=0, end=-1, step=1, at=[], rep=REP_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. Because files specified by `output` or `outputExpr` are controlled (opened/closed) by simulators, they should not be manipulated in a `pyOperator` operator.
- This operator can be applied `Pre-`, `During-` or `Post- Mating` and is applied `PostMating` by default. For example, if you would like to examine the fitness values set by a selector, a `PreMating` Python operator should be used.

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

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.
0.396250        0.201950
0.394650        0.188100
0.372150        0.186400
0.379100        0.175900
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 `pyOpeartor` 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

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.

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

class `pyIndOperator` (*func, loci=[], param=None, stage=PostMating, formOffGenotype=False, begin=0, end=-1, step=1, at=[], rep=REP_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.

`apply` (*pop*)

Apply the `pyIndOperator` operator to one population

`clone` ()

Deep copy of a `pyIndOperator` operator

3.2 Initialization

3.2.1 Class `initializer`

Initialize alleles at the start of a generation

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. Create an initializer. Default to be always active.

class `initializer` (*subPop=[], indRange=[], loci=[], atPloidy=-1, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_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 intialize 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

`clone` ()

Deep copy of an initializer

3.2.2 Class `initSex` (Function form: `InitSex`)

An operator to initialize individual sex. For convenience, this operator is included by other initializers such as initByFreq, initByValue, or pyInit. Initialize individual sex.

class `initSex`(*maleFreq=0.5, sex=[], subPop=[], indRange=[], loci=[], atPloidy=-1, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_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 in turn. If specified, parameter `maleFreq` is ignored.

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

`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

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. Randomly assign alleles according to given allele frequencies

class `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, infoFields=[]*)

alleleFreq: An array of allele frequencies. The sum of all frequencies must be 1; or for a matrix of allele frequencies, each row corresponses 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`.

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

`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

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`.

Initialize a population by given alleles

class initByValue (*value=[], loci=[], atPloidy=-1, subPop=[], indRange=[], proportions=[], maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=REP_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.

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

clone ()
Deep copy of the operator `initByValue`

3.2.5 Class `spread` (Function form: `Spread`)

Copy the genotype of an individual to all individuals

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. Copy genotypes of `ind` to all individuals in `subPop`

class spread (*ind, subPop=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

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

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.

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. Initialize populations using given user function

class pyInit (*func, subPop=[], loci=[], atPloidy=-1, indRange=[], maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=REP_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`.

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

clone()
Deep copy of the operator `pyInit`

3.3 Migration

3.3.1 Class `migrator`

Migrate individuals from (virtual) subpopulations to other subpopulations

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.

Create a migrator

class migrator(*rate, mode=MigrByProbability, fromSubPop=[], toSubPop=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_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 intepretted 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`.

apply(*pop*)
Apply the `migrator`

clone()
Deep copy of a migrator

rate()
Return migration rate

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

$$\begin{pmatrix} 1-r & r/2 & & & r/2 \\ r/2 & 1-r & r/2 & & \\ & & \dots & & \\ & & r/2 & 1-r & r/2 \\ r/2 & & & r/2 & 1-r \end{pmatrix}$$

3.3.3 Class `pyMigrator`

A more flexible Python migrator

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.

Create a hybrid migrator

class pyMigrator(*rateFunc=None, indFunc=None, mode=MigrByProbability, fromSubPop=[], toSubPop=[], loci=[], param=None, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_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`

apply(*pop*)
Apply a `pyMigrator`

clone()
Deep copy of a `pyMigrator`

3.3.4 Class `splitSubPop` (Function form: `SplitSubPop`)

Split a subpopulation Split a subpopulation

class splitSubPop(*which=0, sizes=[], proportions=[], subPopID=[], randomize=True, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_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.

apply(*pop*)
Apply a `splitSubPop` operator

clone()
Deep copy of a `splitSubPop` operator

3.3.5 Class `mergeSubPops` (Function form: `MergeSubPops`)

Merge subpopulations

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

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

subPops: Subpopulations to be merged. Default to all.

`apply` (*pop*)

Apply a `mergeSubPops` operator

`clone` ()

Deep copy of a `mergeSubPops` operator

3.3.6 Class `resizeSubPops` (Function form: `ResizeSubPops`)

Resize subpopulations

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

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

`apply` (*pop*)

Apply a `resizeSubPops` operator

`clone` ()

Deep copy of a `resizeSubPops` operator

3.4 Mutation

3.4.1 Class `mutator`

Base class of all mutators.

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

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

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

There are numbers of possible allelic states. Most theoretical studies assume an infinite number of allelic states to avoid any homoplasy. If it facilitates any analysis, this is however extremely unrealistic. Create a mutator, do not call this constructor directly

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

apply(*pop*)
Apply a mutator

clone()
Deep copy of a `mutator`

maxAllele()
Return maximum allowable allele number

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

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

mutationCounts()
Return mutation counts

rate()
Return the mutation rate

setMaxAllele(*maxAllele*)
Set maximum allowable allele

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

3.4.2 Class `kamMutator` (Function form: `KamMutate`)

K-Allele Model mutator.

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

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

clone()
Deep copy of a `kamMutator`

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.

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. Create a SMM mutator

class smmMutator(*rate=[], loci=[], maxAllele=0, incProb=0.5, output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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`.

clone()

Deep copy of a `smmMutator`

3.4.4 Class `gsmMutator` (Function form: `GsmMutate`)

Generalized stepwise mutation model

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

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

class 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, 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`.

clone()

Deep copy of a `gsmMutator`

mutate(*allele*)

Mutate according to the GSM model

3.4.5 Class `pyMutator` (Function form: `PyMutate`)

A hybrid mutator.

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

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

clone()

Deep copy of a `pyMutator`

mutate(*allele*)

Mutate according to the mixed model

3.4.6 Class `pointMutator` (Function form: `PointMutate`)

Point mutator

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

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

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

inds: Individuals who will mutate

toAllele: Allele that will be mutate to

apply(*pop*)

Apply a `pointMutator`

clone()

Deep copy of a `pointMutator`

mutationCount(*locus*)

Return mutation count at `locus`

mutationCounts()

Return mutation counts

3.5 Recombination and gene conversion

3.5.1 Class `recombinator`

Recombination and conversion

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.

Recombine chromosomes from parents

class 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, 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 ≈ 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/repaired successfully, or a conversion event if the junction is not resolved/repaired. The default `convProb` is 0, meaning no conversion event at all. Note that the ratio of conversion to recombination events varies greatly from study to study, ranging from 0.1 to 15 (Chen et al, Nature Review Genetics, 2007). This translate to 0.1/0.9$\tilde{0}$.1 to 15/16$\tilde{0}$.94 of this parameter. When `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.

clone()

Deep copy of a recombinator

convCount(*size*)

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

convCounts()

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

recCount(*locus*)

Return recombination count at a locus (only valid in standard modules)

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
- Resulution 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 paremters 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 chromosome

   If 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 Convert a random length of chromosome region, using a 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 if 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 translate to 0.1/0.9~0.1 to 15/16~0.94 of this parameter. When \c convProb is 1, all recombination events will be conversion events. The default value if `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.

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

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

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

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

Note:

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

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

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

subPops: Subpopulations that the selector will apply to. Default to all.

`apply`(*pop*)

Set fitness to all individuals. No selection will happen!

`clone`()

Deep copy of a selector

3.6.3 Class `mapSelector` (Function form: `MapSelector`, Applicable to `all ploidy`)

Selection according to the genotype at one or more loci

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

class `mapSelector` (*loci, fitness, phase=False, subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_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. In the haploid case, the genotype should be specified in the form of `'a'` for single locus, and `'a|b|c'` for multi-locus models.

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`.

`clone`()

Deep copy of a map selector

`indFitness` (*ind, gen*)

Calculate/return the fitness value, currently assuming diploid

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)

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.

Create a multiple allele selector

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

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

`indFitness``(`*ind, gen*`)`
Calculate/return the fitness value, currently assuming diploid

3.6.5 Class `mlSelector` (Function form: `MlSelect`)

Selection according to genotypes at multiple loci in a multiplicative model

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

- `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 $f < 0$.

Create a multiple-locus selector

class `mlSelector``(`*selectors, mode=SEL_Multiplicative, subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=["fitness"]*`)`
Please refer to `mapSelector` for other parameter descriptions.

selectors: A list of selectors

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

`indFitness``(`*ind, gen*`)`
Calculate/return the fitness value, currently assuming diploid

3.6.6 Class `pySelector` (Function form: `PySelect`)

Selection using user provided function

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

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

Create a Python hybrid selector

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

clone ()

Deep copy of a `pySelector`

indFitness (*ind, gen*)

Calculate/return the fitness value, currently assuming diploid

3.7 Penetrance

3.7.1 Class `penetrance`

Base class of all penetrance operators.

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

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

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

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

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

class penetrance (*ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_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`.

apply (*pop*)

Set penetrance to all individuals and record penetrance if requested

clone()
Deep copy of a penetrance operator

penet()
Calculate/return penetrance etc.

3.7.2 Class `mapPenetrance` (Function form: `MapPenetrance`)

Penetrance according to the genotype at one locus

Assign penetrance using a table with keys 'X-Y' where X and Y are allele numbers. Create a map penetrance operator

class mapPenetrance(*loci, penet, phase=False, ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_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`.

clone()
Deep copy of a map penetrance operator

3.7.3 Class `maPenetrance` (Function form: `MaPenetrance`)

Multiple allele penetrance operator

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

class maPenetrance(*loci, penet, wildtype, ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

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

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.

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

penet(*ind*)
Currently assuming diploid

3.7.4 Class `mlPenetrance` (Function form: `MlPenetrance`)

Penetrance according to the genotype according to a multiple loci multiplicative model

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

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

Please refer to Neil Risch (1990) for detailed information about these models. Create a multiple locus penetrance operator

class `mlPenetrance` (*peneOps, mode=PEN_Multiplicative, ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

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

peneOps: A list of penetrance operators

`clone`()

Deep copy of a multi-loci penetrance operator

`penet`(*ind*)

Currently assuming diploid

3.7.5 Class `pyPenetrance` (Function form: `PyPenetrance`)

Assign penetrance values by calling a user provided function

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

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

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

class `pyPenetrance` (*loci, func, ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_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__`)

`clone`()

Deep copy of a Python penetrance operator

`penet`(*ind*)

Currently assuming diploid

3.8 Quantitative Trait

3.8.1 Class `quanTrait`

Base class of quantitative trait

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

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

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

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

`clone` ()
Deep copy of a quantitative trait operator

`qtrait` ()
Calculate/return quantitative trait etc.

3.8.2 Class `mapQuanTrait` (Function form: `MapQuanTrait`)

Quantitative trait according to genotype at one locus

Assign quantitative trait using a table with keys 'X-Y' where X and Y are allele numbers. If parameter `sigma` is not zero, the return value is the sum of the trait plus $N\left(0,\sigma^2\right)$. This random part is usually considered as the environmental factor of the trait. Create a map quantitative trait operator

class `mapQuanTrait` (*loci, qtrait, sigma=0, phase=False, ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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 $N\left(mean,\sigma^2\right)$. For multiple loci, the form is 'a-b|c-d|e-f' etc.

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

`clone` ()
Deep copy of a map quantitative trait operator

`qtrait` (*ind*)
Currently assuming diploid

3.8.3 Class `maQuanTrait` (Function form: `MaQuanTrait`)

Multiple allele quantitative trait (quantitative trait according to disease or wildtype alleles)

This is called 'multiple-allele' quantitative trait. It separates alleles into two groups: wildtype and diseased alleles. Wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased alleles.

`maQuanTrait` accepts an array of fitness. Quantitative trait is then set for any given genotype. A standard normal distribution $N\left(0,\sigma^2\right)$ will be added to the returned trait value. Create a multiple allele quantitative trait operator

class maQuanTrait(*loci, qtrait, wildtype, sigma=[], ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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]`.

clone()

Deep copy of a multiple allele quantitative trait

qtrait(*ind*)

Currently assuming diploid

3.8.4 Class `mlQuanTrait` (Function form: `MlQuanTrait`)

Quantitative trait according to genotypes from a multiple loci multiplicative model

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

- `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 σ_i (for f_i) and σ (for f) will be considered. I.e, the trait value should be

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

for `QT_Additive` case. If this is not desired, you can set some of the σ to zero. Create a multiple locus quantitative trait operator

class mlQuanTrait(*qtraits, mode=QT_Multiplicative, sigma=0, ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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

clone()

Deep copy of a multiple loci quantitative trait operator

qtrait(*ind*)

Currently assuming diploid

3.8.5 Class `pyQuanTrait` (Function form: `PyQuanTrait`)

Quantitative trait using a user provided function

For each individual, a user provided function is used to calculate quantitative trait. Create a Python quantitative trait operator

class **pyQuanTrait** (*loci, func, ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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__`)

clone ()

Deep copy of a Python quantitative trait operator

qtrait (*ind*)

Currently assuming diploid

3.9 Ascertainment

3.9.1 Class `sample`

Base class of other sample operator

Ascertainment/sampling refers to the ways of selecting individuals from a population. In simuPOP, ascerntainment 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. Draw a sample

class **sample** (*name="sample", nameExpr="", times=1, saveAs="", saveAsExpr="", format="auto", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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.

apply (*pop*)

Apply the `sample` operator

clone()
Deep copy of a `sample` operator

samples(*pop*)
Return the samples

3.9.2 Class `pySubset` (Function form: `PySubset`)

Shrink population

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. Create a `pySubset` operator

class pySubset(*keep=[], stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

keep: An array of individual subpopulation IDs

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

clone()
Deep copy of a `pySubset` operator

3.9.3 Class `pySample` (Function form: `PySample`)

Python sampler.

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. Create a Python sampler

class 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, 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.

clone()
Deep copy of a Python sampler

drawsample(*pop*)
Draw a Python sample

3.9.4 Class `randomSample` (Function form: `RandomSample`)

Randomly draw a sample from a population

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. Draw a random sample, regardless of the affectedness status

class randomSample(*size=[], name="sample", nameExpr="", times=1, saveAs="", saveAsExpr="", format="auto", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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.

clone()

Deep copy of a `randomSample` operator

3.9.5 Class `caseControlSample` (Function form: `CaseControlSample`)

Draw a case-control sample from a population

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. Draw cases and controls as a sample

class 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, 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

clone()

Deep copy of a `caseControlSample` operator

3.9.6 Class `affectedSibpairSample` (Function form: `AffectedSibpairSample`)

Draw an affected sibling pair sample

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.

Draw an affected sibling pair sample

class 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, 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.

clone()
Deep copy of a `affectedSibpairSample` operator

drawsample(*pop*)
Draw a sample

prepareSample(*pop*)
Preparation before drawing a sample

3.9.7 Class `largePedigreeSample`

Draw a large pedigree sample Draw a large pedigree sample

class 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, 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

clone()
Deep copy of a `largePedigreeSample` operator

drawsample(*pop*)
Draw a a large pedigree sample

prepareSample(*pop*)
Preparation before drawing a sample

3.9.8 Class `nuclearFamilySample`

Draw a nuclear family sample Draw a nuclear family sample

class 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, infoFields=["father_idx", "mother_idx"]*)

Please refer to class `sample` for parameter descriptions.

clone()
Deep copy of a `nuclearFamilySample` operator

drawsample(*pop*)
Draw a nuclear family sample

prepareSample(*pop*)
Preparation before drawing a sample

3.10 Statistics Calculation

3.10.1 Class `stator`

Base class of all the statistics calculator

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. Create a stator

class stator(*output="", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

clone()

Deep copy of a stator

3.10.2 Class `stat` (Function form: `Stat`)

Calculate statistics

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

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

class stat(*popSize=False, numOfMale=False, numOfMale_param={}, numOfAffected=False, numOfAffected_param={}, numOfAlleles=[], numOfAlleles_param={}, alleleFreq=[], alleleFreq_param={}, heteroFreq=[], expHetero=[], expHetero_param={}, homoFreq=[], genoFreq=[], genoFreq_param={}, haploFreq=[], LD=[], LD_param={}, association=[], association_param={}, Fst=[], Fst_param={}, relGroups=[], relLoci=[], rel_param={}, relBySubPop=False, relMethod=[], relMinScored=10, hasPhase=False, midValues=False, output="", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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);
- θ (F_{ST}) the correlation of genes of difference individuals in the same population (will evaluate for each subpopulation and the whole population)
- f (F_{IS}) the correlation of genes within individuals within populations.

This parameter will set the following variables:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

association: Association measures

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

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

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

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

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

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

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 sequencies on loci `haplo`.
- `haploFreq[haplo]`, `subPop[sp]['haploFreq'][haplo]`, the frequency of allele sequencies 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 calcuated and stored in four population variables. For example, `heteroNum[loc][1]` stores number of heterozygotes at locus `loc`, with respect to allele `1`, which is the number of all genotype `1x` or `x1` where does not equal to `1`. All other genotypes such as `02` are considered as homozygotes when `heteroFreq[loc][1]` is calculated. The overall number of heterozygotes (`HeteroNum[loc]`) is the number of genotype `xy` if `x` does not equal to `y`.

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

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

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

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

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

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

- `propOfUnaffected`, `subPop[sp]['propOfUnAffected']` the proportion of unaffected individuals in the population/subpopulation.

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

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

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

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

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

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

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

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. There is no subpopulation level relatedness value.

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

apply (*pop*)

Apply the `stat` operator

clone()
Deep copy of a `stat` operator

3.11 Expression and Statements

3.11.1 Class `dumper`

Dump the content of a population. Dump a population

class 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, 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)

alleleOnly()
Only show alleles (not structure, gene information?

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

clone()
Deep copy of an operator

infoOnly()
Only show info

setAlleleOnly(*alleleOnly*)
SimuPOP::dumper::setAlleleOnly

setInfoOnly(*infoOnly*)
SimuPOP::dumper::setInfoOnly

3.11.2 Class `savePopulation`

Save population to a file Save population

class savePopulation(*output="", outputExpr="", format="", compress=True, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

compress: Obsolete parameter

format: Obsolete parameter

output: Output filename.

outputExpr: An expression that will be evalulated dynamically to determine file name. Parameter `output` will be ignored if this parameter is given.

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

clone ()
Deep copy of an operator

3.11.3 Class `pyOutput`

Output a given string.

A common usage is to output a new line for the last replicate. Create a `pyOutput` operator that outputs a given string.

class pyOutput (*str="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

str: String to be outputted

apply (*pop*)
Simply output some info

clone ()
Deep copy of an operator

setString (*str*)
Set output string.

3.11.4 Class `pyEval` (Function form: `PyEval`)

Evaluate an expression

Python expressions/statements will be executed when `pyEval` is applied to a population by using parameters `expr/stmts`. Statements can also been 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. Evaluate expressions/statments in the local namespace of a replicate

class pyEval (*expr="", stmts="", preStmts="", postStmts="", exposePop=False, name="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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

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

clone ()
Deep copy of a `pyEval` operator

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

This operator takes a list of statements and executes them. No value will be returned or outputted. Evaluate statments in the local replicate namespace, no return value

class pyExec(*stmts="", preStmts="", postStmts="", exposePop=False, name="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

Please refer to class `pyEval` for parameter descriptions.

clone()

Deep copy of a `pyExec` operator

3.11.6 Class `infoEval` (Function form: `infoEval`)

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

class infoEval(*expr="", stmts="", subPops=[], usePopVars=False, exposePop=False, name="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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 avaialbe with option usePopVars. Changes to these variables will change the corresponding information fields of individuals.Please note that, 1. If population variables are used, and there are name conflicts between information fields and variables, population variables will be overridden by information fields, without any warning. 2. Information fields are float numbers. An exceptions will raise if an information field can not be converted to a float number. 3. This operator can be used in all stages. When it is used during-mating, it will act on each offspring.

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`

apply(*pop*)

Apply the `infoEval` operator

clone()

Deep copy of a `infoEval` operator

name()

Return the name of an expression

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

3.11.7 Class `infoExec` (Function form: `infoExec`)

Execute a Python statement for each individual, using information fields

This operator takes a list of statements and executes them. No value will be returned or outputted. Fields, optionally with variable in population's local namespace

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

Please refer to class `infoEval` for parameter descriptions.

clone()

Deep copy of a `infoExec` operator

3.12 Tagging (used for pedigree tracking)

3.12.1 Class `tagger`

Base class of tagging individuals

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

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

Create a `tagger`, default to be always active but no output

class tagger(*output="", outputExpr="", begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

apply(*pop*)

Add a newline

clone()

Deep copy of a tagger

3.12.2 Class `inheritTagger`

Inherite tag from parents

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

An example may be tagging one or a few parents and examining, at the last generation, how many offspring they have. Create an `inheritTagger` that inherits a tag from one or both parents

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

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

applyDuringMating(*pop, offspring, dad=None, mom=None*)

Apply the `inheritTagger`

clone()

Deep copy of a `inheritTagger`

3.12.3 Class `parentTagger`

Tagging according to parental indexes

This during-mating operator set `tag()` each individual with indexes of his/her parent in the parental population. Because only one parent is recorded, this is recommended to be used for mating schemes that requires only one parent (such as selfMating).This tagger record indexes to information field parent_idx, and/or a given file. The usage is similar to parentsTagger. Create a `parentTagger`

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

apply(*pop*)
With a newline.

applyDuringMating(*pop, offspring, dad=None, mom=None*)
Apply the `parentTagger`

clone()
Deep copy of a `parentTagger`

3.12.4 Class `parentsTagger`

Tagging according to parents' indexes

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

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

Create a `parentsTagger`

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

apply(*pop*)
With a newline.

applyDuringMating(*pop, offspring, dad=None, mom=None*)
Apply the `parentsTagger`

clone()
Deep copy of a `parentsTagger`

3.12.5 Class `sexTagger`

Tagging sex status.

This is a simple post-mating tagger that write sex status to a file. By default, 1 for Male, 2 for Female. SimuPOP::sexTagger::sexTagger

class sexTagger(*code=[], begin=0, end=-1, step=1, at=[], rep=REP_ALL, stage=PostMating, output=">", outputExpr="", infoFields=[]*)

code: Code for Male and Female, default to 1 and 2, respectively. This is used by Linkage format.

apply(*pop*)
Add a newline

3.12.6 Class `affectionTagger`

Tagging affection status.

This is a simple post-mating tagger that write affection status to a file. By default, 1 for unaffected, 2 for affected. SimuPOP::affectionTagger::affectionTagger

class affectionTagger(*code=[], begin=0, end=-1, step=1, at=[], rep=REP_ALL, stage=PostMating, output=">", outputExpr="", infoFields=[]*)

code: Code for Male and Female, default to 1 and 2, respectively. This is used by Linkage format.

apply(*pop*)

Add a newline

3.12.7 Class `infoTagger`

Tagging information fields.

This is a simple post-mating tagger that write given information fields to a file (or standard output). SimuPOP::infoTagger::infoTagger

class infoTagger(*begin=0, end=-1, step=1, at=[], rep=REP_ALL, stage=PostMating, output=">", outputExpr="", infoFields=[]*)

apply(*pop*)

Add a newline

3.12.8 Class `pyTagger`

Python tagger.

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

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

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

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

applyDuringMating(*pop, offspring, dad=None, mom=None*)

Apply the `pyTagger`

clone()

Deep copy of a `pyTagger`

3.13 Terminator

3.13.1 Class `terminator`

Base class of all terminators.

Teminators are used to see if an evolution is running as expected, and terminate the evolution if a certain condition fails. Create a terminator

class terminator (*message="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

message: A message that will be displayed when the evolution is terminated.

clone()

Deep copy of a terminator

3.13.2 Class `terminateIf`

Terminate according to a condition

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 opertor 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. Create a `terminateIf` terminator

class terminateIf (*condition="", message="", var="terminate", output="", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

apply (*pop*)

Apply the `terminateIf` terminator

clone()

Deep copy of a `terminateIf` terminator

3.13.3 Class `continueIf`

Terminate according to a condition failure

The same as `terminateIf` but continue if the condition is `True`. Create a `continueIf` terminator

class continueIf (*condition="", message="", var="terminate", output="", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

apply (*pop*)

Apply this operator

clone()

Deep copy of a `continueIf` terminator

3.14 Conditional operator

3.14.1 Class `ifElse`

Conditional operator

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 machanism is to monitor the population statistics and act accordingly. Create a conditional operator

class `ifElse`(*cond, ifOp=None, elseOp=None, output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_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`

`apply`(*pop*)

Apply the `ifElse` operator to one population

`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

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. Create a `turnOnDebug` operator

class turnOnDebug(*code, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

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

clone()
Deep copy of a `turnOnDebug` operator

3.15.2 Class `turnOffDebug` (Function form: `TurnOffDebug`)

Set debug off

Turn off debug. Create a `turnOffDebug` operator

class turnOffDebug(*code, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

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

clone()
Deep copy of a `turnOffDebug` operator

3.16 Miscellaneous

3.16.1 Class `noneOp`

None operator

This operator does nothing. Create a none operator

class noneOp(*output=">", outputExpr="", stage=PostMating, begin=0, end=0, step=1, at=[], rep=REP_ALL, infoFields=[]*)

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

clone()
Deep copy of a `noneOp` operator

3.16.2 Class `pause`

Pause a simulator

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. Stop a simulation. Press `'q'` to exit or any other key to continue.

class pause(*prompt=True, stopOnKeyStroke=False, exposePop=True, popName="pop", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_LAST, 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.

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

clone()
Deep copy of a `pause` operator

3.16.3 Class `ticToc` (Function form: `TicToc`)

Timer operator

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. Create a timer

class ticToc(*output=">", outputExpr="", stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

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

clone()
Deep copy of a `ticToc` operator

3.16.4 Class `setAncestralDepth`

Set ancestral depth

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. Create a `setAncestralDepth` operator

class **`setAncestralDepth`**(*depth, output=">", outputExpr="", stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL, infoFields=[]*)

`apply`(*pop*)

Apply the `setAncestralDepth` operator to one population

`clone`()

Deep copy of a `setAncestralDepth` operator

CHAPTER

FOUR

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 Classes

4.2.1 Class `RNG`

Random number generator

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

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

max()
Maximum value of this RNG.

maxSeed()
Return the maximum allowed seed value

name()
Return RNG name

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

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

randExponential(*v*)
SimuPOP::RNG::randExponential

randGeometric(*p*)
Geometric distribution.

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

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

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

randMultinomialVal(*N, p*)
SimuPOP::RNG::randMultinomialVal

randNormal(*m, v*)
Normal distribution.

randPoisson(*p*)
Poisson distribution.

randUniform01()
Uniform distribution [0,1).

seed()
Return the seed of this RNG

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

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

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

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

4.3 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.

4.3.1 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 fromcommand line or configuraiton file, users will be asked to input them interactively.

Because these options are reserved, you can not use them in your simuPOP script.

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 ofargument `'-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 foruser input.

default: default value for this parameter. It is used to as the default valuein 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 determinedfrom GUI, command line option or config file. This is useful for options that rarely need to be changed. Setting them to useDfault allows shorter command lines, and easy user input.

description: a long description of this parameter, will be put into the usageinformation, 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.ListTypeor` 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 functionsor use the ones defined in this module.

chooseOneOf: if specified, `simuOpt` will choose one from a list of values using alistbox (Tk) or a combo box (wxPython) .

chooseFrom: if specified, `simuOpt` will choose one or more items from a list ofvalues using a listbox (tk) or a combo box (wxPython).

separator: if specified, a blue label will be used to separate groups ofparameters.

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 0x2aaaaaad6198>*)
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, supress banner information when simuPOP is loaded.

debug: a list of debug code (or string). If not set, environmental variableSIMUDEBUG 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

4.3.2 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.

CollectValue(*pop, gen, expr, name*)
EXPERIMENTAL

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.

FreqTrajectoryMultiStochWithSubPop (*curGen, numLoci, freq, NtFunc, minMutAge, maxMutAge, fitness=[], mode='uneven', ploidy=2, restartIfFail=True, fitnessFunc=None*)

EXPERIMENTAL Simulate frequency trajectory with subpopulation structure, migration is currently ignored. The essential part of this script is to simulate the trajectory of each subpopulation independently by calling FreqTrajectoryMultiStoch with properly wrapped NtFunc function. If mode = 'even' (default) When freq is the same length of the number of loci. The allele frequency at the last generation will be multi-nomially distributed. If freq for each subpop is specified in the order of loc1-sp1, loc1-sp2, .. loc2-sp1, ... This freq will be used directly. If mode = 'uneven'. The number of disease alleles will be proportional to the interval lengths of 0 x x x 1 while x are uniform [0,1]. The distribution of interval lengths, are roughly exponential (conditional on overall length 1). ' If mode = 'none', subpop will be ignored. This script assume a single-split model of NtFunc

InstantExpansion (*initSize, endSize, end, burnin=0, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0*)

Instaneously 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, bottleneck-Size=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 returnedby 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

LoadGCData(*file, loci=[]*)

EXPERIMENTAL read GC data file in http://wpicr.wpic.pitt.edu/WPICCompGen/genomic_control/genomic_control.htm

MigrIslandRates(*r, n*)

migration rate matrix x m/(n-1) m/(n-1) .... m/(n-1) x .... m/(n-1) m/(n-1) x where x = 1-m

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

migration rate matrix, circular stepping stone model (X=1-m) X `m/2` m/2 m/2 X `m/2` 0 0 `m/2` `x` m/2 .....0 ... m/2 0 .... `m/2` X or non-circular X `m/2` m/2 m/2 X `m/2` 0 0 `m/2` `X` m/2 .....0 ... ... `m` X

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

fileds: information fields, 'sex' and 'affection' are special fields thatis 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 tooutput 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 tobe 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'slocal 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 diploidindividual.

shift: if combine is not given, output two alleles directly, addingthis 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 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 formatbut 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 formatbut 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.

collector (*name, expr, **kwargs*)
EXPERIMENTAL

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 recorda.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

getGenotype (*pop, atLoci=[], subPop=[], indRange=[], atPloidy=[]*)
EXPERIMENTAL Obtain genotype as specified by parameters

atLoci: subset of loci, default to all

subPop: subset of subpopulations, default ao all

indRange: individual rangesThis is mostly used for testing purposes because the returned array can be large for large populations.

saveFstat (*output=", outputExpr=", **kwargs*)
operator version of the function SaveFstat

saveLinkage (*output=", outputExpr=", **kwargs*)
An operator to save population in linkage format

testDemoFunc (*end, func*)
EXPERIMENTAL

trajFunc (*endingGen, traj*)
EXPERIMENTAL return freq at each generation from a simulated trajctories.

4.3.3 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

4.3.4 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 0x20e428e0> >, expand='exponential', mergeAt=10000, initMultiple=1, recIntensity=0.01, mutRate=9.9999999999999995e-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 avoidrapid loss of genotype variation when population size is small.

endingSize: ending poplation 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 filesshould have been prepared by scripts/loadHapMap.py.

names: names of markers. It can either be a stright list of names, ora dictionary of names categorized by chromosome number.

chroms: a list of chromosomes to look in. If empty, all 22 autosomeswill 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 havethe 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 filesshould have been prepared by scripts/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 lcous 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

absIndIndex() (population method), 13
absLocusIndex() (GenoStruTrait method), 10
activateVirtualSubPop() (population method), 13
addGen() (pedigree method), 57
addInfo() (pedigree method), 57
addInfoField() (population method), 13
addInfoField() (simulator method), 49
addInfoFields() (population method), 13
addInfoFields() (simulator method), 49
affected() (individual method), 29
affectedChar() (individual method), 29
affectedSibpairSample (class in), 85
affectionSplitter (class in), 24
affectionTagger (class in), 96
allele() (individual method), 29
alleleChar() (individual method), 29
alleleName() (GenoStruTrait method), 10
alleleNames() (GenoStruTrait method), 10
alleleOnly() (dumper method), 91
AlleleType() (in module), 103
alleleType, 3
alphaMating (class in), 35
ancestor() (population method), 14
ancestralDepth() (population method), 14
ancestralGen() (population method), 14
apply() (affectionTagger method), 96
apply() (baseOperator method), 48
apply() (continueIf method), 97
apply() (dumper method), 91
apply() (ifElse method), 98
apply() (infoEval method), 93
apply() (infoTagger method), 96
apply() (initByFreq method), 63
apply() (initByValue method), 64
apply() (initSex method), 63
apply() (mergeSubPops method), 68
apply() (migrator method), 66
apply() (mutator method), 69
apply() (noneOp method), 99
apply() (parentTagger method), 95
apply() (parentsTagger method), 95
apply() (pause method), 100
apply() (penetrance method), 78
apply() (pointMutator method), 71
apply() (pyEval method), 92
apply() (pyIndOperator method), 62
apply() (pyInit method), 65
apply() (pyMigrator method), 67
apply() (pyOperator method), 60
apply() (pyOutput method), 92
apply() (pySubset method), 84
apply() (quanTrait method), 81
apply() (resizeSubPops method), 68
apply() (sample method), 83
apply() (savePopulation method), 92
apply() (selector method), 75
apply() (setAncestralDepth method), 101
apply() (sexTagger method), 95
apply() (splitSubPop method), 67
apply() (spread method), 64
apply() (stat method), 90
apply() (tagger method), 94
apply() (terminateIf method), 97
apply() (ticToc method), 100
apply() (turnOffDebug method), 99
apply() (turnOnDebug method), 99
applyDuringMating() (inheritTagger method), 94
applyDuringMating() (parentTagger method), 95
applyDuringMating() (parentsTagger method), 95
applyDuringMating() (pyTagger method), 96
ascertainment, 83

baseOperator (class in), 48
baseRandomMating (class in), 33
binomialSelection (class in), 33

carray, 6
CaseControl_ChiSq() (in module), 108
caseControlSample (class in), 85
ChiSq_test() (in module), 108
chromBegin() (GenoStruTrait method), 10
chromByName() (GenoStruTrait method), 10
chromEnd() (GenoStruTrait method), 10

`chromLocusPair()` (GenoStruTrait method), 10
`chromName()` (GenoStruTrait method), 10
`chromNames()` (GenoStruTrait method), 10
class
    affectedSibpairSample, 85
    affectionSplitter, 24
    affectionTagger, 96
    alphaMating, 35
    baseOperator, 47
    baseRandomMating, 33
    binomialSelection, 32
    caseControlSample, 85
    cloneMating, 32
    cloneOffspringGenerator, 39
    combinedSplitter, 26
    consanguineousMating, 35
    continueIf, 97
    dumper, 91
    GenoStruTrait, 10
    genotypeSplitter, 25
    gsmMutator, 70
    haplodiploidMating, 36
    haplodiploidOffspringGenerator, 40
    heteroMating, 37
    ifElse, 97
    individual, 28
    infoEval, 93
    infoExec, 93
    infoParentsChooser, 38
    infoSplitter, 25
    infoTagger, 96
    inheritTagger, 94
    initByFreq, 63
    initByValue, 63
    initializer, 62
    initSex, 63
    kamMutator, 69
    largePedigreeSample, 86
    maPenetrance, 79
    mapPenetrance, 79
    mapQuanTrait, 81
    mapSelector, 76
    maQuanTrait, 81
    maSelector, 76
    mating, 31
    mendelianOffspringGenerator, 40
    mergeSubPops, 68
    migrator, 65
    mlPenetrance, 79
    mlQuanTrait, 82
    mlSelector, 77
    monogamousMating, 34
    mutator, 68
    noMating, 31
    noneOp, 99
    nuclearFamilySample, 86
    parentsTagger, 95
    parentTagger, 94
    pause, 100
    pedigree, 57
    pedigreeMating, 36
    pedigreeParentsChooser, 39
    penetrance, 78
    pointMutator, 71
    polygamousMating, 34
    population, 12
    proportionSplitter, 25
    pyEval, 92
    pyExec, 93
    pyIndOperator, 61
    pyInit, 64
    pyMating, 36
    pyMigrator, 66
    pyMutator, 70
    pyOperator, 59
    pyOutput, 92
    pyParentsChooser, 39
    pyPenetrance, 80
    pyQuanTrait, 82
    pySample, 84
    pySelector, 77
    pySubset, 84
    pyTagger, 96
    quanTrait, 81
    randomMating, 33
    randomParentChooser, 37
    randomParentsChooser, 38
    randomSample, 84
    rangeSplitter, 25
    recombinator, 71
    resizeSubPops, 68
    RNG, 104
    sample, 83
    savePopulation, 91
    selector, 75
    selfingOffspringGenerator, 39
    selfMating, 34
    sequentialParentChooser, 37
    sequentialParentsChooser, 37
    setAncestralDepth, 100
    sexSplitter, 24
    sexTagger, 95
    simulator, 48
    smmMutator, 69
    splitSubPop, 67
    spread, 64
    stat, 87
    stator, 87

tagger, 94
terminateIf, 97
terminator, 96
ticToc, 100
turnOffDebug, 99
turnOnDebug, 99
clone() (affectedSibpairSample method), 86
clone() (affectionSplitter method), 24
clone() (alphaMating method), 36
clone() (baseOperator method), 48
clone() (baseRandomMating method), 33
clone() (binomialSelection method), 33
clone() (caseControlSample method), 85
clone() (cloneMating method), 32
clone() (cloneOffspringGenerator method), 39
clone() (combinedSplitter method), 26
clone() (consanguineousMating method), 35
clone() (continueIf method), 97
clone() (dumper method), 91
clone() (genotypeSplitter method), 26
clone() (gsmMutator method), 70
clone() (haplodiploidMating method), 36
clone() (haplodiploidOffspringGenerator method), 40
clone() (heteroMating method), 37
clone() (ifElse method), 98
clone() (infoEval method), 93
clone() (infoExec method), 94
clone() (infoParentsChooser method), 39
clone() (inheritTagger method), 94
clone() (initByFreq method), 63
clone() (initByValue method), 64
clone() (initSex method), 63
clone() (initializer method), 62
clone() (kamMutator method), 69
clone() (largePedigreeSample method), 86
clone() (maPenetrance method), 79
clone() (maQuanTrait method), 82
clone() (maSelector method), 77
clone() (mapPenetrance method), 79
clone() (mapQuanTrait method), 81
clone() (mapSelector method), 76
clone() (mating method), 31
clone() (mendelianOffspringGenerator method), 40
clone() (mergeSubPops method), 68
clone() (migrator method), 66
clone() (mlPenetrance method), 80
clone() (mlQuanTrait method), 82
clone() (mlSelector method), 77
clone() (monogamousMating method), 34
clone() (mutator method), 69
clone() (noMating method), 32
clone() (noneOp method), 100
clone() (nuclearFamilySample method), 86
clone() (parentTagger method), 95
clone() (parentsTagger method), 95
clone() (pause method), 100
clone() (pedigreeMating method), 36
clone() (pedigreeParentsChooser method), 39
clone() (pedigree method), 57
clone() (penetrance method), 79
clone() (pointMutator method), 71
clone() (polygamousMating method), 35
clone() (population method), 14
clone() (proportionSplitter method), 25
clone() (pyEval method), 92
clone() (pyExec method), 93
clone() (pyIndOperator method), 62
clone() (pyInit method), 65
clone() (pyMating method), 37
clone() (pyMigrator method), 67
clone() (pyMutator method), 70
clone() (pyOperator method), 60
clone() (pyOutput method), 92
clone() (pyParentsChooser method), 39
clone() (pyPenetrance method), 80
clone() (pyQuanTrait method), 83
clone() (pySample method), 84
clone() (pySelector method), 78
clone() (pySubset method), 84
clone() (pyTagger method), 96
clone() (quanTrait method), 81
clone() (randomMating method), 34
clone() (randomParentChooser method), 38
clone() (randomParentsChooser method), 38
clone() (randomSample method), 85
clone() (rangeSplitter method), 25
clone() (recombinator method), 72
clone() (resizeSubPops method), 68
clone() (sample method), 84
clone() (savePopulation method), 92
clone() (selector method), 75
clone() (selfMating method), 34
clone() (selfingOffspringGenerator method), 40
clone() (sequentialParentChooser method), 37
clone() (sequentialParentsChooser method), 37
clone() (setAncestralDepth method), 101
clone() (sexSplitter method), 24
clone() (simulator method), 49
clone() (smmMutator method), 70
clone() (splitSubPop method), 67
clone() (spread method), 64
clone() (stat method), 91
clone() (stator method), 87
clone() (tagger method), 94
clone() (terminateIf method), 97
clone() (terminator method), 97
clone() (ticToc method), 100
clone() (turnOffDebug method), 99

clone() (turnOnDebug method), 99
cloneMating (class in), 32
cloneOffspringGenerator (class in), 39
collector() (in module), 113
CollectValue() (in module), 108
combinedSplitter (class in), 26
consanguineousMating (class in), 35
constant
    DuringMating, 51
    PostMating, 51
    PreMating, 51
    PrePostMating, 51
ConstSize() (in module), 109
continueIf (class in), 97
convCount() (recombinator method), 72
convCounts() (recombinator method), 72
conversion, 71
copyParentalGenotype() (haplodiploidOffspringGenerator method), 40

dataAggregator() (in module), 113
deactivateVirtualSubPop() (population method), 14
diploidOnly() (baseOperator method), 48
distLeft() (GenoStruTrait method), 10
drawsample() (affectedSibpairSample method), 86
drawsample() (largePedigreeSample method), 86
drawsample() (nuclearFamilySample method), 86
drawsample() (pySample method), 84
dumper (class in), 91

evaluate() (population method), 14
evolve() (simulator method), 49
evolveHapMap() (in module), 114
execute() (population method), 14
ExponentialExpansion() (in module), 109

father() (pedigree method), 57
finalize() (pyParentsChooser method), 39
fitSubPopStru() (population method), 14
formOffspringGenotype() (mendelianOffspringGenerator method), 40
FreqTrajectoryMultiStochWithSubPop() (in module), 109
Function
    migrIslandRates, 66
    migrStepstoneRates, 66
    SavePopulation, 22
    turnOffDebug, 98
    TurnOnDebug, 98
function
    AffectedSibpairSample, 85
    CaseControlSample, 85
    GsmMutate, 70
    InitByFreq, 63
    InitByValue, 63
    InitSex, 63
    KamMutate, 69
    MaPenetrance, 79
    MaQuanTrait, 81
    MaSelect, 76
    MapPenetrance, 79
    MapQuanTrait, 81
    MapSelector, 76
    MergeSubPops, 68
    MlPenetrance, 79
    MlQuanTrait, 82
    MlSelect, 77
    PointMutate, 71
    PyEval, 92
    PyExec, 93
    PyInit, 64
    PyMutate, 70
    PyPenetrance, 80
    PyQuanTrait, 82
    PySample, 84
    PySelect, 77
    PySubset, 84
    RandomSample, 84
    ResizeSubPops, 68
    SmmMutate, 69
    SplitSubPop, 67
    Spread, 64
    Stat, 87
    TicToc, 100
    TurnOffDebug, 99
    TurnOnDebug, 99
    infoEval, 93
    infoExec, 93
    LoadPopulation, 22
    rng, 7

gen() (pedigree method), 57
gen() (population method), 14
gen() (simulator method), 50
genoSize() (GenoStruTrait method), 10
GenoStruTrait
    alleleName, 9
    arrLociPos, 9
    chromName, 9
    infoField, 9
    infoFields, 9
    locusPos, 9
    maxAllele, 9
    numChrom, 9
    numLoci, 9
    ploidy, 9
    ploidyName, 9
    sexChrom, 9

totNumLoci, 9
GenoStruTrait (class in), 10
genotype() (individual method), 29
genotype() (population method), 14
genotypeSplitter (class in), 26
genotypic structure, 9
getGenotype() (in module), 113
getMarkersFromName() (in module), 115
getMarkersFromRange() (in module), 115
getParam() (in module), 105
getPopulation() (simulator method), 50
gsmMutator (class in), 70

haplodiploid() (GenoStruTrait method), 10
haplodiploidMating (class in), 36
haplodiploidOffspringGenerator (class in), 40
haploidOnly() (baseOperator method), 48
hasInfoField() (GenoStruTrait method), 10
hasVirtualSubPop() (population method), 14
heteroMating (class in), 37

ifElse (class in), 98
indBegin() (population method), 15
indEnd() (population method), 15
index
absolute, 5
relative, 5
indFitness() (maSelector method), 77
indFitness() (mapSelector method), 76
indFitness() (mlSelector method), 77
indFitness() (pySelector method), 78
indInfo() (population method), 15
individual() (population method), 15
individual (class in), 28
individuals() (population method), 15
info() (individual method), 29
info() (pedigree method), 57
infoEval (class in), 93
infoExec (class in), 94
infoField() (GenoStruTrait method), 10
infoField() (baseOperator method), 48
infoFields() (GenoStruTrait method), 10
infoIdx() (GenoStruTrait method), 10
infoOnly() (dumper method), 91
infoParentsChooser (class in), 38
infoSize() (GenoStruTrait method), 11
infoSize() (baseOperator method), 48
infoSplitter (class in), 25
infoTagger (class in), 96
inheritTagger (class in), 94
initByFreq (class in), 63
initByValue (class in), 64
initializer, 62
initializer (class in), 62
initSex (class in), 63
insertAfterLoci() (population method), 15
insertAfterLocus() (population method), 15
insertBeforeLoci() (population method), 16
insertBeforeLocus() (population method), 16
InstantExpansion() (in module), 109
intInfo() (individual method), 29

kamMutator (class in), 69

largePedigreeSample (class in), 86
LargePeds_Reg_merlin() (in module), 109
LargePeds_VC_merlin() (in module), 109
Limits() (in module), 103
LinearExpansion() (in module), 110
ListAllRNG() (in module), 103
ListDebugCode, 3
ListDebugCode() (in module), 103
listDebugCode, 98
ListVars() (in module), 110
load() (pedigree method), 57
LoadFstat() (in module), 110
LoadGCData() (in module), 110
loadInfo() (pedigree method), 57
LoadPopulation() (in module), 103
LoadSimulator() (in module), 103
loadSimulator, 51
locateRelatives() (population method), 16
lociByNames() (GenoStruTrait method), 11
lociCovered() (GenoStruTrait method), 11
lociDist() (GenoStruTrait method), 11
lociLeft() (GenoStruTrait method), 11
lociNames() (GenoStruTrait method), 11
lociPos() (GenoStruTrait method), 11
locusByName() (GenoStruTrait method), 11
locusName() (GenoStruTrait method), 11
locusPos() (GenoStruTrait method), 11
LOD_gh() (in module), 109
LOD_merlin() (in module), 109

maPenetrance (class in), 79
mapPenetrance (class in), 79
mapQuanTrait (class in), 81
mapSelector (class in), 76
maQuanTrait (class in), 82
markUnrelated() (pedigree method), 57
maSelector (class in), 76
mating (class in), 31
mating scheme, 31
max() (RNG method), 104
MaxAllele() (in module), 103
maxAllele() (GenoStruTrait method), 11
maxAllele() (mutator method), 69
maxSeed() (RNG method), 104

mendelianOffspringGenerator (class in), 40
mergePopulation() (population method), 16
mergePopulationByLoci() (population method), 16
MergePopulations() (in module), 103
MergePopulationsByLoci() (in module), 103
mergeSubPops() (population method), 17
mergeSubPops (class in), 68
migrator, 65
migrator (class in), 65
MigrIslandRates() (in module), 110
MigrSteppingStoneRates() (in module), 110
mlPenetrance (class in), 80
mlQuanTrait (class in), 82
mlSelector (class in), 77
module
    hapMapUtil, 114
    simuOpt, 105
    simuRPy, 113
    simuUtil, 108
ModuleCompiler() (in module), 103
ModuleDate() (in module), 103
ModulePlatForm() (in module), 103
ModulePyVersion() (in module), 103
monogamousMating (class in), 34
mother() (pedigree method), 58
mutate() (gsmMutator method), 70
mutate() (kamMutator method), 69
mutate() (mutator method), 69
mutate() (pyMutator method), 70
Mutation, 68
mutationCount() (mutator method), 69
mutationCount() (pointMutator method), 71
mutationCounts() (mutator method), 69
mutationCounts() (pointMutator method), 71
mutator (class in), 68

name() (RNG method), 104
name() (affectionSplitter method), 24
name() (combinedSplitter method), 26
name() (genotypeSplitter method), 26
name() (infoEval method), 93
name() (infoSplitter method), 25
name() (proportionSplitter method), 25
name() (pyEval method), 92
name() (rangeSplitter method), 25
name() (sexSplitter method), 24
newPopByIndID() (population method), 17
newPopWithPartialLoci() (population method), 17
noMating (class in), 32
noneOp (class in), 99
nuclearFamilySample (class in), 86
numChrom() (GenoStruTrait method), 11
numLoci() (GenoStruTrait method), 11
numParents() (pedigree method), 58
numRep() (simulator method), 50
numSubPop() (population method), 17
numVirtualSubPop() (affectionSplitter method), 24
numVirtualSubPop() (combinedSplitter method), 26
numVirtualSubPop() (genotypeSplitter method), 26
numVirtualSubPop() (infoSplitter method), 25
numVirtualSubPop() (population method), 17
numVirtualSubPop() (proportionSplitter method), 25
numVirtualSubPop() (rangeSplitter method), 25
numVirtualSubPop() (sexSplitter method), 24

operator
    DuringMating, 51
    PostMating, 51
    PreMating, 51
    PrePostMating, 51
    stat, 51
    turnOffDebug, 98
    turnOnDebug, 98
Optimized() (in module), 103

parentsTagger (class in), 95
parentTagger (class in), 95
pause (class in), 100
pedigree (class in), 57
pedigreeMating (class in), 36
pedigreeParentsChooser (class in), 39
penet() (maPenetrance method), 79
penet() (mlPenetrance method), 80
penet() (penetrance method), 79
penet() (pyPenetrance method), 80
penetrance, 78
penetrance (class in), 78
ploidy() (GenoStruTrait method), 11
ploidyName() (GenoStruTrait method), 11
pointMutator (class in), 71
polygamousMating (class in), 34
popSize() (pedigree method), 58
popSize() (population method), 17
population, 11
    individual, 26
    population, 52
    vars, 52
population() (simulator method), 50
population (class in), 12
prepareSample() (affectedSibpairSample method), 86
prepareSample() (largePedigreeSample method), 86
prepareSample() (nuclearFamilySample method), 86

prettyOutput() (in module), 107
printConfig() (in module), 107
proportionSplitter (class in), 25
pushAndDiscard() (population method), 17
pvalChiSq() (RNG method), 104
pyEval (class in), 92
pyExec (class in), 93
pyIndOperator (class in), 62
pyInit (class in), 64
pyMating (class in), 37
pyMigrator (class in), 67
pyMutator (class in), 70
pyOperator (class in), 59
pyOutput (class in), 92
pyParentsChooser (class in), 39
pyPenetrance (class in), 80
pyQuanTrait (class in), 83
pySample (class in), 84
pySelector (class in), 78
pySubset (class in), 84
pyTagger (class in), 96

qtrait() (maQuanTrait method), 82
qtrait() (mapQuanTrait method), 81
qtrait() (mlQuanTrait method), 82
qtrait() (pyQuanTrait method), 83
qtrait() (quanTrait method), 81
QtraitSibs_Reg_merlin() (in module), 110
QtraitSibs_VC_merlin() (in module), 110
quantitative trait, 81
quanTrait (class in), 81

randBinomial() (RNG method), 104
randExponential() (RNG method), 104
randGeometric() (RNG method), 104
randGet() (RNG method), 104
randInt() (RNG method), 104
randMultinomial() (RNG method), 104
randMultinomialVal() (RNG method), 104
randNormal() (RNG method), 105
randomMating (class in), 33
randomParentChooser (class in), 38
randomParentsChooser (class in), 38
randomSample (class in), 85
randPoisson() (RNG method), 105
randUniform01() (RNG method), 105
rangeSplitter (class in), 25
rate() (migrator method), 66
rate() (mutator method), 69
rawIndBegin() (population method), 17
rawIndEnd() (population method), 17
recCount() (recombinator method), 72
recCounts() (recombinator method), 73
recombination, 71
recombinator (class in), 71
Regression_merlin() (in module), 111
removeEmptySubPops() (population method), 17
removeIndividuals() (population method), 18
removeLoci() (population method), 18
removeSubPops() (population method), 18
removeUnrelated() (pedigree method), 58
reorderSubPops() (population method), 18
rep() (population method), 18
requireRevision() (in module), 107
resize() (population method), 18
resizeSubPops (class in), 68
rmatrix() (in module), 114
RNG (class in), 104
rng() (in module), 104

sample (class in), 83
sample1DSL() (in module), 116
sample2DSL() (in module), 116
samples() (sample method), 84
save() (pedigree method), 58
saveConfig() (in module), 107
SaveCSV() (in module), 111
SaveFstat() (in module), 111
saveFstat() (in module), 113
saveInfo() (pedigree method), 58
SaveLinkage() (in module), 111
saveLinkage() (in module), 113
SaveMerlinDatFile() (in module), 111
SaveMerlinMapFile() (in module), 111
SaveMerlinPedFile() (in module), 111
savePopulation, 22
savePopulation() (population method), 18
savePopulation (class in), 91
SaveQTDT() (in module), 111
saveSimulator() (simulator method), 50
SaveSolarFrqFile() (in module), 112
seed() (RNG method), 105
selectIndividuals() (pedigree method), 58
selection, 74
selectionOn() (population method), 18
selector (class in), 75
selfingOffspringGenerator (class in), 40
selfMating (class in), 34
sequentialParentChooser (class in), 37
sequentialParentsChooser (class in), 37
setAffected() (individual method), 30
setAllele() (individual method), 30
setAlleleOnly() (dumper method), 91
setAncestralDepth() (population method), 18
setAncestralDepth() (simulator method), 50
setAncestralDepth (class in), 101
setFather() (pedigree method), 58
setGen() (simulator method), 50

setGenotype() (individual method), 30
setGenotype() (population method), 18
setIndexesOfRelatives() (population method), 19
setIndInfo() (population method), 18, 19
setIndSubPopID() (population method), 19
setIndSubPopIDWithID() (population method), 19
setInfo() (individual method), 30
setInfo() (pedigree method), 58
setInfoFields() (population method), 19
setInfoOnly() (dumper method), 91
setMatingScheme() (simulator method), 50
setMaxAllele() (mutator method), 69
setMother() (pedigree method), 58
setOptions() (in module), 107
setRate() (mutator method), 69
setRates() (migrator method), 66
SetRNG() (in module), 103
setRNG() (RNG method), 105
setSeed() (RNG method), 105
setSex() (individual method), 30
setString() (pyOutput method), 92
setSubPopByIndID() (population method), 19
setSubPopID() (individual method), 30
setSubPopStru() (population method), 19
setVirtualSplitter() (population method), 20
sex() (individual method), 30
sexChar() (individual method), 30
sexChrom() (GenoStruTrait method), 11
sexSplitter (class in), 24
sexTagger (class in), 95
Sibpair_LOD_gh() (in module), 112
Sibpair_LOD_merlin() (in module), 112
Sibpair_TDT_gh() (in module), 112
SIMUALLELETYPE, 3
Simulator, 48
simulator
    dryun, 51
simulator (class in), 49
simuOpt, 3
simuRev() (in module), 104
simuVer() (in module), 104
smmMutator (class in), 70
splitSubPop() (population method), 20
splitSubPop (class in), 67
splitSubPopByProportion() (population method), 20
spread (class in), 64
stat (class in), 87
stator (class in), 87
step() (simulator method), 50
submitScratch() (mating method), 31
subPopBegin() (population method), 20
subPopEnd() (population method), 20
subPopID() (individual method), 30
subPopIndPair() (population method), 20
subPopSize() (pedigree method), 58
subPopSize() (population method), 20
subPopSizes() (pedigreeParentsChooser method), 39
subPopSizes() (pedigree method), 58
subPopSizes() (population method), 20
swap() (population method), 20

tagger (class in), 94
TDT_gh() (in module), 112
terminateIf (class in), 97
terminator (class in), 97
testDemoFunc() (in module), 113
ticToc (class in), 100
totNumLoci() (GenoStruTrait method), 11
trajFunc() (in module), 113
TurnOffDebug() (in module), 104
turnOffDebug (class in), 99
turnOffSelection() (population method), 20
TurnOnDebug() (in module), 104
turnOnDebug (class in), 99
turnOnSelection() (population method), 20

unaffected() (individual method), 30
usage() (in module), 107
useAncestralPop() (population method), 20

validate() (population method), 20
valueAnd() (in module), 107
valueBetween() (in module), 107
valueEqual() (in module), 107
valueGE() (in module), 107
valueGT() (in module), 107
valueIsList() (in module), 107
valueIsNum() (in module), 108
valueLE() (in module), 108
valueListOf() (in module), 108
valueLT() (in module), 108
valueNot() (in module), 108
valueNotEqual() (in module), 108
valueOneOf() (in module), 108
valueOr() (in module), 108
valueTrueFalse() (in module), 108
valueValidDir() (in module), 108
valueValidFile() (in module), 108
varPlotter() (in module), 114
vars() (population method), 20
vars() (simulator method), 50
VC_merlin() (in module), 113
virtualSubPopName() (population method), 20
virtualSubPopSize() (population method), 21