simuPOP Reference Manual

Release 0.8.0 (Rev: 1167)

Bo Peng

December 2004

Last modified
6th August 2007

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

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

LIST OF EXAMPLES

CHAPTER

ONE

Introduction

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

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

Example 1.1: Getting help using the help() function

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

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:           inital 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 its base class. For example, class `population`, `individual` and `simulator` are all derived from class `GenoStruTrait`. Therefore, you can have access to 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, parameter `begin, end, step, at` etc are shared by all operators, and is explained in details only in class `baseOperator.`

1.1 Loading simuPOP

simuPOP is composed of six libraries: standard short, long and binary alleles (3), each of them have standard and optimized ($\times 2$) modules. A Message Passing Interface (MPI) versions is under development but not yet available. The short libraries 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 libraries can save you a lot of RAM. Depending on applications, binary modules can be faster or slower than regular modules.

Standard libraries have detailed debug and run-time validation mechanism to make sure the simulations run correctly. Whenever something unusual is detected, simuPOP would terminate with detailed error messages. The cost of such run-time checking varies from simulation to simulation but can be very high under some extreme circumstances. Because of this, optimized versions for all libraries 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 libraries whenever possible and only use the optimized version when performance is needed and you are confidence that your simulation is running as expected.

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

Example 1.2: Use of standard simuPOP modules

```
>>> pop = population(10, loci=[2])
>>> pop.locusPos(10)
Traceback (most recent call last):
  File "refManual.py", line 1, in ?
    #
IndexError: src/genoStru.h:444 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:451 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 be 'short', 'long' or 'binary', `SIMUOPTIMIZED` to use the optimized modules, and `SIMUMPI` to use MPI modules. 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` means suppress initial output, and `debug` should be a comma-separated list of debug options specified by `listDebugCode`().
- If you are running a simuPOP script that conforms to simuPOP convention, you should be able to use optimized library 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`: `1` for binary libraries, usually `255` for short libraries and $2^{32}-1$ for long libraries. Note that these numbers for short and long libraries might be changed on different platforms.
- `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, this can limit the size of population you can simulate.

Example 1.4: set options through simuOpt

```
>>> import simuOpt
>>> simuOpt.setOptions(optimized=False, alleleType='long', quiet=True)
>>> from simuPOP import *
>>> print alleleType()
long
>>> print optimized()
False
>>>
```

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. Assume that `simu` is a simulator with several populations,

1. `pop = simu.population(idx)` will get a reference to the `idx`'th population. You can, although not recommended, modify simulator through this `pop` reference. Because this population reference will become invalid when the simulator is destroyed, the following calling sequence can crash Python:

   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()
   ```

   In this example, the simulator `simu` will be destroyed after the call to `func()` is ended, leaving `pop` as a referece to an invalid population object.

2. To get an independent copy of a population, you can use `pop = simu.getPopulation(idx)`. Population `pop` returned in this way in Example 1.5 is valid.

3. If the simulator will be destroyed as the case in Example 1.5,

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

   can be used. This function will *extract* population `idx` 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 loci 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 a subpopulation.

Example 1.6: Conversion between absolute and relative indices

```
>>> pop = population(subPop=[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. An operator is created and passed as a parameter to the `evolve` function of a simulator. During evolution, the `evolve()` function determine if an operator can be applied to a population and apply it when appropriate.

Example 1.7: Population and operators

```
>>> simu = simulator(pop, randomMating(), rep=3)
>>> simu.evolve(
...     preOps = [ initByFreq([.8, .2])],
...     ops = [
...         stat(alleleFreq=[0,1], Fst=[1], step=10),
...         kamMutator(rate=0.001, rep=1),
...         kamMutator(rate=0.0001, rep=2)
...     ],
...     end=10
... )
True
>>>
```

In Example 1.7, operators `initByFreq, stat` and two copies of `kamMutator` are created. During evolution, `simu` will apply `initByFreq` once to each replicate of the simulator; apply the first `kamMutator` to the first replicate and the second `kamMutator` to the second replicate at every generation; apply `stat` to count allele frequency and calculate F_{st} every 10 generations. More details about operators will be described later.

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( [0.3, .2, .5] ).apply(pop)
```

or dump the content of a population by

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

This style of calling 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 1.8: Function InitByFreq

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

The function form will be marked as `Function form` in this reference manual.

1.5 `carray` type

The return value of simuPOP functions with names start with `arr` is of a special Python type `carray`. This object reflects the underlying C/C++ array and you can read/write array elements just as a regular list, with the exception that you can not change the size of the array. Therefore, only `count` and `index` list function can be used.

Example 1.9: Usage of the carray type

```
>>> # obtain an object using one of the arrXXX functions
>>> pop = population(loci=[3,4], lociPos=[1,2,3,4,5,6,7])
>>> arr = pop.arrLociPos()
>>> # print and expression (just like list)
>>> print arr
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> str(arr)
'[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]'
>>> # count
>>> arr.count(2)
1
>>> # index
>>> arr.index(2)
1
>>> # can read write
>>> arr[0] = 0.5
>>> # convert to list
>>> arr.tolist()
[0.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> # or simply
>>> list(arr)
[0.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> # compare to list directly
>>> arr == [0.5, 1.0, 3.0, 3.5, 5.0, 6.0, 7.0]
False
>>> # you can also convert and compare
>>> list(arr) == [0.5, 1.0, 3.0, 3.5, 5.0, 6.0, 7.0]
False
>>> # slice
>>> arr[:] = [1,2,3,4,5,6,7]
>>> # assign from another part
>>> arr[1:3] = arr[3:5]
>>> # arr1 is 1,2,3
>>> arr1 = arr[:3]
>>> # assign slice from a number
>>> # arr will also be affected since arr1 point to a part of arr
>>> arr1[:] = 10
>>> # assign vector of the same length
>>> len(arr1)
3
>>> arr1[:] = [30,40, 50]
>>>
```

No other operation is allowed.

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

1.6 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()
  ```

- 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 (equivalent to `initByFreq(vars).apply(pop)`).

- Constants start with Capital characters. For example

  ```
  MigrByProportion, StatNumOfFemale
  ```

- The following words in function names are abbreviated:

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

1.7 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, 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

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

Information fields refer to the 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.

Individuals in the same population share the same genotypic structure. Consequently, *the genotypic information can be accessed from individual, population and simulator* (consists of populations with the same genotypic structure) *levels*.

2.1.1 Class `GenoStruTrait`

genotypic structure related functions, can be accessed from both individuals and populations

Details

Genotypic structure refers to the number of chromosomes, positions, the number 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.

Initialization

This object can not be created directly. It is created by a population.

`GenoStruTrait()`

Member Functions

`x.absLocusIndex(chrom, locus)` return the absolute index of a locus on a chromosome

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

`x.alleleNames()` return an array of allele names

`x.arrLociPos()` return an (editable) array of loci positions of all loci

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

`x.arrLociPos(chrom)` return an (editable) array of loci positions on a given chromosome

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

`x.chromBegin(chrom)` return the index of the first locus on a chromosome

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

Note: From the description of this function, the returned value may not be a valid index. (This is consistant with Python ranges.)

`x.chromLocusPair(locus)` return a `(chrom, locus)` pair of an absolute locus index

`x.genoSize()` return the total number of loci times ploidy

`x.hasInfoField(name)` determine if an information field exists

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

`x.infoFields()` return an array of all information fields

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

`x.infoSize()` obtain the number of information fields

`x.lociByNames(names)` return an array of locus indices by locus names

`x.lociNames()` return names of all loci

`x.lociPos()` return loci positions

`x.locusByName(name)` return the index of a locus by its locus name

`x.locusName(loc)` return the name of a locus

`x.locusPos(locus)` return the position of a locus

`x.maxAllele()` return the maximum allele value for all loci. Default to maximum allowed allele state.

Maximum allele value has to be `1` for binary modules. `maxAllele` is the maximum possible allele value, which allows `maxAllele+1` alleles `0, 1, ..., maxAllele`.

`x.numChrom()` return the number of chromosomes

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

`x.numLoci()` return the number of loci on all chromosomes

`x.ploidy()` return ploidy, the number of homologous sets of chromosomes

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

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

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

Example

Example 2.1: Genotypic structure

```
>>> # create a population, most parameters have default values
>>> pop = population(size=5, ploidy=2, loci=[5,10],
...     lociPos=[range(0,5),range(0,20,2)],
...     alleleNames=['A','C','T','G'],
...     subPop=[2,3], maxAllele=3)
>>> print pop.popSize()
5
>>> print pop.ploidy()
2
>>> print pop.ploidyName()
diploid
>>> print pop.numChrom()
2
>>> print pop.locusPos(2)
2.0
>>> print pop.alleleName(1)
C
>>> # get the fourth individual of the population
>>> ind = pop.individual(3)
>>> # access genotypic structure info
>>> print ind.ploidy()
2
>>> print ind.numChrom()
2
>>> print ind.numLoci(0)
5
>>> print ind.genoSize()
30
>>> # and from simulator level
>>> simu = simulator(pop, randomMating(), rep=3)
>>> print simu.numChrom()
2
>>>
```

2.1.2 Introduction

2.1.3 Sex chromosome

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.

2.1.4 Information fields

An individual will by default have genotype, sex and affection status information, but other information is needed for some operations. For example, the fitness value of an individual is needed for selection, one or more trait values may be needed to calculate quantitative traits, and age may be needed if age-dependent mating schemes are used. 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.

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 fields are needed.

The information fields can be accessed from each individual (c.f. `info(idx), info(name), setInfo(value, idx), setInfo(value, name), arrInfo()` of the individual), or from the population level (c.f. `setIndInfo(value), arrIndInfo(subPop)`). 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.

2.2 Population

`population` objects are essential to simuPOP. They are composed of subpopulations each with certain number of individuals, all have the same genotypic structure. A population can store arbitrary number of ancestral populations to facilitate pedigree analysis.

2.2.1 Class `population`

a collection of individuals with the same genotypic structure

Details

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

- subpopulation. 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 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 in simuPOP there is no sub-subpopulation or family in subpopulations.

- variables. Every population has its own variable space, or *localnamespaces* 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 these namespaces. For example, function `Stat` set 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 make a certain ancestral generation *current* are provided so that one can examine and modify ancestral generations.

Other important concepts like *information fields* are explained in class `individual`???.

Note

Although a large number of member functions are provided, most of the operations are performed by *operators*. These functions will only be useful when you need to manipulate a population explicitly.

Initialization

Create a population object with given size and genotypic structure.

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

FIXME: Details of constructure is missing. ???This is techniquely the `__init__` function of the population object.

alleleNames an array of allele names. The first element should be given to invalid/unknown allele. For example, for a locus with alleles A,C,T,G, you can specify `alleleNames` as `('_','A','C','T','G')`. Note that simuPOP uses `1,2,3,4` internally and these names will only be used for display purpose.

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 populations may quickly exhaust your computer RAM. If you really need to do that, use `savePopulation` operator to save each generation to a file is a much better choice.

infoFields name of information fields that will be attached to each individual. For example, if you need to record the parents of each individual you will need two, if you need to record the age of individual, you need an additional one. Other possibilities include offspring IDs etc. Note that you have to plan this ahead of time since, for example, tagger will need to know what info unit to use. Default to `none`.

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 loci. Default to `"locX-X"` where `X-X` is a chromosome-loci index 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 locations, although they are usually intepreted as centiMorgans. The default values are `1`, `2`, etc. on each chromosome.

maxAllele maximum allele number. Default to the max allowed allele states of current library (standard or long allele version) maximum allele state for the whole population. This will set a cap for all loci. For individual locus, you can specify maxAllele in mutation models, which can be smaller than 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 chromosomes. Default to `2` (diploid).

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

size population size. Can be ignored if `subPop` is specified. In that case, `size` is the sum of `subPop`. Default to `0`.

subPop an array of subpopulation sizes. Default value is `[size]` which means a single subpopulation of the whole population. If both `size` and `subPop` are are provided, `subPop` should add up to `size`.

Member Functions

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

index index of an individual in a subpopulation `subPop`

subPop subpopulation index (start from `0`)

`x.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 inital value for the new field.

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

fields new information fields. If one or more of the fields alreay exist, they will simply be re-initialized.

init initial value for the new fields.

`x.adjustInfoPosition(order)` simuPOP::population::adjustInfoPosition

`x.ancestralDepth()` ancestral depth of the current population

Note: The returned value is the number of ancestral generations exists in the population, not necessarily equal to the number set by setAncestralDepth().

`x.ancestralGen()` currently used ancestral population (0 for the latest generation)

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

`x.arrGenotype(order)` get the whole genotypes

Return an editable array of all genotypes of the population. You need to know how these genotypes are organized to safely read/write genotype directly. Individuals will be in order before exposing their genotypes.

order if order is `true`, respect order; otherwise, do not repect population structure.

`x.arrGenotype(subPop, order)` get the whole genotypes

Return an editable array of all genotype of a subpopulation. Individuals will be in order before exposing their genotypes.

order if order is `true`, keep order; otherwise, respect `subpop` structure.

subPop index of subpopulation (start from 0)

`x.arrIndInfo(order)` get an editable array (Python list) of all information fields

order whether or not the list has the same order as individuals

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

x.equalTo(rhs) compare two populations

x.evaluate(expr="", stmts="") evaluate a python statment/expression

This function evaluates a python statment/expression and return its result as a string. Optionally run statement first.

x.execute(stmts="") evaluate a statement (can be multi-line string)

x.gen() current generation during evolution

x.grp() current group ID in a simulator

Group number is not meaningful for a stand-alone population.

x.ind(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

x.indInfo(idx, order) get information `idx` of all individuals.

idx index in all information fields

order if true, sort returned vector in individual order

x.indInfo(name, order) get information `name` of all individuals.

name name of the information field

order if true, sort returned vector in individual order

x.indInfo(idx, subPop, order) get information `idx` of all individuals in a subpopulation `subPop`.

idx index in all information fields

order if true, sort returned vector in individual order

subPop subpopulation index

x.indInfo(name, subPop, order) get information `name` of all individuals in a subpopulation `subPop`.

name name of the information field

order if true, sort returned vector in individual order

subPop subpopulation index

x.individuals() return an iterator that can be used to iterate through all individuals

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

x.insertAfterLoci(idx, pos, names=[]) append loci at given locations

Append loci at some given locations. In an appended location, alleles will be zero.

idx an array of locus index. The loci will be added *after* each index. If you need to append to the first locus, please use `insertBeforeLoci`. If your index is the last locus of a chromosome, the appended locus will become the last of that chromosome. If you need to append multiple loci after a locus, please repeat that locus number.

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

pos an array of locus positions. You need to make sure that the position will make the appended locus between adjacent markers.

`x.insertAfterLocus(idx, pos, name=string)` append an locus at a given location

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

`x.insertBeforeLoci(idx, pos, names=[])` insert loci at given locations

Insert loci at some given locations. In an inserted location, alleles will be zero.

idx an array of locus index. The loci will be inserted *before* each index. If you need to append to the last locus, please use `insertAfterLoci`. If your index is the first locus of a chromosome, the inserted locus will become the first of that chromosome. If you need to insert multiple loci before a locus, please repeat that locus number.

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

pos an array of locus positions. You need to make sure that the position will make the inserted locus between adjacent markers.

`x.insertBeforeLocus(idx, pos, name=string)` insert an locus at given location.

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

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

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

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 may run `mergePopulation` without this parameter, and then adjust subpopulation sizes generation by generation.

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

`x.mergePopulationByLoci(pop, newNumLoci=[], newLociPos=[], byChromosome=False)` merge populations by loci

Two populations should have the same number of individuals. This also holds for any ancestral generations. By default, chromosomes of `pop` are added to the current population. That is to say, chromosomes from `pop` is added, as new chromosomes to this populaton. You can change this arrangement in two ways

- specify new chromosome structure using parameter `newLoci`. loci from new and old population are still in their original order, but chromosome number and position 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`. On each chromosome, loci position should in order.

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

Note:

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

`x.mergeSubPops(subPops=[], removeEmptySubPops=False)` merge given subpopulations

Merge subpopulations, the first subpopulation ID (the first one in array `subPops`) will be used as the ID of the new subpopulation. That is to say, all subpopulations will take the ID of the first one.

`x.newPopByIndID(keepAncestralPops=-1, id=[], removeEmptySubPops=False)` Form a new population according to the parameter information. Information can be given directly as

- `keepAncestralPops=-1`: keep all
- `keepAncestralPops=0`: only current
- `keepAncestralPops=1`: keep one ...

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

Copy current population to a new one with selected loci and remove specified loci. (No change on the current population.)

`x.numSubPop()` number of subpopulations in a population

`x.popSize()` obtain total population size

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

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

`x.rearrangeLoci(newNumLoci, newLociPos)` rearrange loci on chromosomes, e.g. combine two chromosomes into one

This is used by `mergeByLoci`.

`x.removeEmptySubPops()` remove empty subpopulations by adjusting subpopulation IDs

`x.removeIndividuals(inds=[], subPop=-1, removeEmptySubPops=False)` remove individuals

If a valid `subPop` is given, remove individuals from this subpopulation.

`x.removeLoci(remove=[], keep=[])` remove some loci from the current population. Loci that will be removed or kept can be specified.

`x.removeSubPops(subPops=[], shiftSubPopID=True, removeEmptySubPops=False)` remove subpopulations and adjust subpopulation IDs so that there will be no *'empty'* subpopulation left

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

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

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

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

`x.rep()` current replicate in a simulator

Replication number is not meaningful for a stand-alone population.

`x.resize(newSubPopSizes, propagate=False)` resize population

Resize population by giving new subpopulation sizes.

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

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

Note: This function only resizes the current generation.

`x.savePopulation(filename, format="auto", compress=True)`
simuPOP::population::savePopulation

filename save to filename

format format to save. Can be one of the following: 'text', 'bin', or 'xml'. The default format is 'text' but the output is not supposed to be read. 'bin' has smaller size than the other two and should be used for large populations. 'xml' is the most readable format and should be used when you would like to convert simuPOP populations to other formats.

`x.setAncestralDepth(depth)` set ancestral depth.

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

`x.setIndInfo(values, idx)` set individual information for the given information field (index),

idx index to the information field.

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

`x.setIndInfo(values, name)` set individual information for the given information field (name)

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

`x.setIndSubPopID(id)` set subpopulation ID with given ID

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

id an array of the same length of population size, resprenting subpopulation ID of each individual.

`x.setIndSubPopIDWithID()` set subpopulation ID of each individual with their current subpopulation ID???

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

fields an array of fields

init initial value for the new fields.

`x.setSubPopByIndID(id=[])` adjust subpopulation according to individual subpopulation ID.

Rearrange individuals to their new subpopulations according to their subpopulation ID (or the new given `ID`). Order within each subpopulation is not respected.

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

Note: Individual with negative info will be removed!

x.setSubPopStru(newSubPopSizes, allowPopSizeChange=False) set population/subpopulation structure given subpopulation sizes

allowPopSizeChange if this parameter is `true`, population will be resized.

subPopSize an array of subpopulation sizes. The population may or may not change according to parameter `allowPopSizeChange` if the sum of `subPopSize` does not match `popSize`.

x.splitSubPop(which, sizes, subPopID=[]) split a subpopulation into subpopulations of given sizes

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

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

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

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

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

x.subPopBegin(subPop) index of the first individual of a subpopulation

subPop subpopulation index

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

subPop subpopulation index

Note: As with all ...End functions, the returning index is out of the range so that the actual range is [xxxBegin, xxxEnd). This agrees with all STL conventions and Python range.

x.subPopIndPair(ind) return the `(sp, idx)` pair from an absolute index of an individual

x.subPopSize(subPop) return size of a subpopulation `subPop`

subPop index of subpopulation (start from 0)

x.subPopSizes() return an array of all subpopulation sizes

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

x.turnOffSelection() Turn off selection for all subpopulations.

If you really want to apply another selector, run turnOffSelection to eliminate the effect of the previous one.

x.useAncestralPop(idx) use an ancestral generation. `0` for the latest generation.

idx Index of the ancestral generation. `0` for current, `1` for parental, etc. idx can not exceed ancestral depth (see setAncestralDepth).

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

Example

Example 2.2: Population initialization

```
>>> # a Wright-Fisher population
>>> WF = population(size=100, ploidy=1, loci=[1])
>>>
>>> # a diploid population of size 10
>>> # there are two chromosomes with 5 and 7 loci respectively
>>> pop = population(size=10, ploidy=2, loci=[5, 7], subPop=[2, 8])
>>>
>>> # a population with SNP markers (with names A,C,T,G
>>> # range() are python functions
>>> pop = population(size=5, ploidy=2, loci=[5,10],
...     lociPos=[range(0,5),range(0,20,2)],
...     alleleNames=['A','C','T','G'],
...     subPop=[2,3], maxAllele=3)
>>>
```

2.2.2 Population overview

simuPOP uses one-level population structure. That is to say, there is no sub-subpopulation or family in 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.

`population` has a large number of member functions, ranging from reviewing simple properties to generating a new population from the current one. However, you do not have to know all the member functions to use a population. As a matter of fact, you will only use a small portion of these functions unless you need to write pure Python functions/operators that involves complicated manipulation of populations.

2.2.3 Creating a population

A population can be created by the following methods:

- call `population` function to create an instance of population from `population` class.
- call `LoadPopulation`, `LoadFstat` etc. to load a population from a saved file.
- be generated as a subset of an existing population by operators such as `randomSample`, `caseControlSample` or equivalent functions `RandomSample`, `CaseControlSample`.

- be obtained from an existing simulator through `simulator::getPopulation()`.

Help contents of all functions of `population` class can be displayed by `help(population)`. Help on a member function can be viewed by `help(population.func)`. In Python, constructors are named `__init__` and you can use the class name to create an instance of this class. Therefore, to display parameters of `population` function, you need to run

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

Please refer to `population` class reference for detailed information of parameters.

Example 2.3 shows a few examples of using the population function to create populations.

Example 2.3: Use of population function

```
>>> # a Wright-Fisher population
>>> WF = population(size=100, ploidy=1, loci=[1])
>>>
>>> # a diploid population of size 10
>>> # there are two chromosomes with 5 and 7 loci respectively
>>> pop = population(size=10, ploidy=2, loci=[5, 7], subPop=[2, 8])
>>>
>>> # a population with SNP markers (with names A,C,T,G
>>> # range() are python functions
>>> pop = population(size=5, ploidy=2, loci=[5,10],
...     lociPos=[range(0,5),range(0,20,2)],
...     alleleNames=['A','C','T','G'],
...     subPop=[2,3], maxAllele=3)
>>>
```

2.2.4 Population structure

Please refer to the class reference of population for detailed information of its member functions. Here is an example of population structure.

Example 2.4: population structure functions

```
>>> print pop.popSize()
5
>>> print pop.numSubPop()
2
>>> print pop.subPopSize(0)
2
>>> print pop.subPopSizes()
(2, 3)
>>> print pop.subPopBegin(1)
2
>>> print pop.subPopEnd(1)
5
>>> print pop.subPopIndPair(3)
(1, 1)
>>> print pop.absIndIndex(1,1)
3
>>>
```

There is another set of functions that deal with population/subpopulation size changes. In these functions, the `info` field of each individual plays an important role. This field represents an individual's (new) subpopulation ID most of the times. For example, function `rearrangeByIndID()` rearranges individuals in the order of their `info` values.

These functions may look useful and appealing but you will almost never use them directly. All these operations will be performed by various operators, in a more user-friendly way. Only when you begin to write your own operators will you have to read about the details of these functions.

Example 2.5 demonstrates the use of functions `setIndSubPopID, setSubPopByIndID` and `removeLoci`.

Example 2.5: population structure functions

```
>>> pop.setIndSubPopID([1,2,2,3,1])
>>> pop.setSubPopByIndID()
>>> pop.removeLoci(keep=range(2,7))
>>> Dump(pop)
Ploidy:                 2
Number of chrom:        2
Number of loci:         3 2
Maximum allele state:   3
Loci positions:
                2 3 4
                0 2
Loci names:
                loc1-3 loc1-4 loc1-5
                loc2-1 loc2-2
population size:        5
Number of subPop:       4
Subpop sizes:           0 2 2 1
Number of ancestral populations:        0
individual info:
sub population 1:
   0: MU CCG CA | GTA TT
   1: FU AAG TT | CCT AC
sub population 2:
   2: MU CTT AC | CAA AC
   3: MU ACC TT | TCT TT
sub population 3:
   4: FU ACT GT | ATC AA
End of individual info.


No ancenstral population recorded.
>>>
```

2.2.5 Population variables

Populations are associated with Python variables. These variables are usually set by various operators. For example, `stat` operator calculates many population statistics and store results in population namespace. Example 2.2.5 demonstrates how `stat` set variables `popSize`, `alleleFreq` etc.

You can refer to these variables using `population::vars()` or `population::dvars()` function. The returned values of `vars()` and `dvars()` reflect the same dictionary. However, `dvars()` uses a little Python magic so that you can use attribute syntax to access dictionary keys. Since `a.allaleFreq[0]` is much easier to read than `a['alleleFre'][0]`, `dvars()` is always preferred to `vars()`. A function `ListVars` defined in `simuUtil`

can be used to display the variables. With wxPython installed, this function will open a nice window with a tree representing the variables. Without wxPython (or use parameter `useWxPython=False`), variables will be displayed in a 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 2.6: Population variables

```
>>> from simuUtil import ListVars
>>> ListVars(pop.vars(), useWxPython=False)
 rep :  -1
 grp :  -1
>>> Stat(pop, popSize=1, alleleFreq=[0])
>>> # subPop is True by default, use name to limit the variables to display
>>> ListVars(pop.vars(), useWxPython=False, subPop=False, name='alleleFreq')
 alleleFreq :
   [0]
     [0]        0.4
     [1]        0.5
     [2]        0.0
     [3]        0.1
>>> # print number of allele 1 at locus 0
>>> print pop.vars()['alleleNum'][0][1]
5
>>> print pop.dvars().alleleNum[0][1]
5
>>> print pop.dvars().alleleFreq[0]
[0.40000000000000002, 0.5, 0.0, 0.10000000000000001]
>>>
```

These variables form a Python dictionary, and furthermore a local namespace for functions like `population::evaluate`. *Local namespace* means that you can use dictionary items as variables during evaluation. For example:

Example 2.7: Local namespaces of populations

```
>>> print pop.evaluate('alleleNum[0][1] + alleleNum[0][2]')
5
>>> pop.execute('newPopSize=int(popSize*1.5)')
>>> ListVars(pop.vars(), level=1, useWxPython=False)
 newPopSize :   7
 grp :  -1
 rep :  -1
 popSize :      5
 numSubPop :    4
 alleleNum :
   list of length 1
 subPopSize :
   list of length 4
 alleleFreq :
   list of length 1
 subPop
```

```
   list of length 4
>>> # 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
>>>
```

As you can see, these variables are *local* to the population and is not directly accessible from the main namespace. `vars(subPop)` and `dvars(subPop)` function can be used. Both functions take an optional `subPop` option. If ignored, they will return the population dictionary; otherwise, they will return the dictionary for subpopulation `subPop`. This is a very convenient feature, because subpopulations and populations have similar keys, you can calculate the same statistics for the whole population and individual subpopulations, just by specifying different namespaces.

2.2.6 FIXME: duplicate information?

All populations have their own attached variables. We have seen the structure of a population dictionary: it is empty at the beginning and will have many variables created by various operators later. You can access the local namespace of each replicate through a simulator's `vars(rep)` function:

```
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
    ...                 ...
```

It is important to know that

- `simulator::vars(0), vars(1)` etc. are the *local namespaces* for each replicate.
- `subPop[0]`, `subPop[1]` etc. have almost the same set of keys as those for the whole population. This is because `stat` operator calculates statistics of each replicate of population, and all subpopulations.

To list these variables, you can use the `ListVars()` function defined in `simuUtil.py`. For example

```
ListVars(simu.vars(0), level=2)
```

list all variables for the first replicate. `level=2` stops `ListVars` from expanding lists and dictionaries after two levels.

Two functions can be used to access simulator and population variables: `vars()` and `dvars()`. We have known `population::vars()` and `population::dvars()`, `simulator::vars()` and `simulator::dvars()` work in almost the same way.

- `simulator::vars(rep)`, `dvars(rep)`: return replicate `rep`'s local namespace.

- `simulator::vars(rep, subPop)`, `dvars(rep, subPop)`: return the namespace of `subPop` subpopulation of replicate `rep`.

The return values of `vars()` and `dvars()` are different. `vars()` returns a Python dictionary. You should access their keys in the usual Python way. `dvars()` returns a 'wrapped' Python dictionary. You can access their dictionary keys as attributes. `dvars()` is usually considered to be easier to use than `vars()`.

2.2.7 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 to store most recent `ancestry` generations; or `-1` to store all populations.

Several important usage of ancestral populations:

- `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 sample from the population.
- `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 population. You should always call `setAncestralPop(0)` after you examined the ancestral populations.

A typical use of this function is demonstrated in example 2.8. 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 2.8: 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
... )
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().allel
...
0     0.173200     0.022300
1     0.165450     0.021200
2     0.161800     0.020300
3     0.155350     0.018850
4     0.147750     0.018650
5     0.138600     0.016350
>>> # restore to the current generation
>>> pop.useAncestralPop(0)
>>>
```

2.2.8 Save and Load a Population

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

- `xml:` most readable, easy transformation to other formats, largest file size
- `bin:` not readable, small file size. May not be portable.
- `txt:` human readable with no structure, portable, median file size.

Example 2.9: Save and load population

```
>>> # save it in various formats, default format is "txt"
>>> pop = population(1000, loci=[2, 5, 10])
>>> pop.savePopulation("pop.txt")
>>> pop.savePopulation("pop.txt", compress=False)
>>> pop.savePopulation("pop.xml", format="xml")
>>> pop.savePopulation("pop.bin", format="bin")
>>>
>>> # load it in another population
>>> pop1 = LoadPopulation("pop.xml", format="xml")
>>>
```

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

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

It is also possible to save a bunch of populations in a single file, provided that they have the same genotypic structure. The functions are

- `SavePopulations([pop1,pop2,...,], filename, format='auto', compress=True)`
- `LoadPopulations(filename)`

Shared variables will also be 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

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

2.2.9 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,

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

2.3 Individuals

2.3.1 Class `individual`

individuals with genotype, affection status, sex etc.

Details

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

- shared genotypic structure information
- genotype
- sex, affection status, subpopulation ID
- optional information fields

Individual genotypes are arranged by locus, chromosome, ploidy, in that order, and can be accessed from a single index. For example, for a diploid individual with two loci on the first chromosome, one locus on the second, its genotype is arranged as `1-1-1 1-1-2 1-2-1 2-1-1 2-1-2 2-2-1` where `x-y-z` represents ploidy `x` chromosome `y` and locus `z`. An allele `2-1-2` can be accessed by `allele(4)` (by absolute index), `allele(2, 1)` (by index and ploidy) or `allele(1, 1, 0)` (by index, ploidy and chromosome).

Initialization

Individuals are created by populations automatically. Do not call this function directly.

```
individual()
```

Member Functions

x.affected() whether or not an individual is affected

x.affectedChar() return A or U for affection status

x.allele(index) return the allele at locus index

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

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

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

p index of the ploidy

x.allele(index, p, ch) return the allele at locus index of the ch-th chromosome of 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

x.alleleChar(index) return the name of allele(index)

x.alleleChar(index, p) return the name of allele(index, p)

x.alleleChar(index, p, ch) return the name of allele(idx, p, ch)

x.arrGenotype() return an editable array (a Python list of length totNumLoci()*ploidy()) of genotypes of an individual

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

x.arrGenotype(p) return only the p-th copy of the chromosomes

x.arrGenotype(p, ch) return only the ch-th chromosome of the p-th copy

x.arrInfo() return an editable array of all information fields (a Python list of length infosSize())

x.info(idx) get information field idx

idx index of the information field

x.info(name) get information field name

Equivalent to info(infoIdx(name)).

name name of the information field

x.setAffected(affected) set affection status

x.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()

x.setAllele(allele, index, p) set the allele at locus index of the p-th copy of the chromosomes

allele allele to be set

index index from the begining of the poloidy p, ranging from 0 to totNumLoci(p)

p index of the poloidy

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

allele allele to be set

ch index of the chromosome in ploidy `p`

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

p index of the ploidy

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

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

x.setSex(sex) set the sex. You should use `setSex(Male)` or `setSex(Female)` instead of `1` and `2`.

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

x.sex() return the sex of an individual, `1` for males and `2` for females. However, this is not guranteed so please use `sexChar()`.

x.sexChar() return the sex of an individual, `M` or `F`

x.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.

x.unaffected() equals to `not affected()`

You can access individuals of a population through `individual()` function. There are four forms of this function, two with and two without parameter `subPop`,

- `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(), individuals(subPop)` return an iterator that can be used to iterate through all individuals.

The iterator can simplify the access of individuals, by using

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

instead of the older

```
for i in range(pop.popSize()):
  ind = pop.individual(i)
  print ind.affected()
```

The returned `individual` object also has its own member functions. You can retrieve genotypic information of an individual through the same set of functions. You can also get/set genotypes of an individual. Note that you can not create an individual object directly.

Example 2.10: Individual member functions

```
>>> # get an individual
>>> ind = pop.individual(9)
Traceback (most recent call last):
  File "refManual.py", line 1, in ?
    #
IndexError: src/population.h:451 individual index (9) is out of range of 0 ~ 4
>>> # oops, wrong index
>>> ind = pop.individual(3)
>>> # you can access genotypic structure info
>>> print ind.ploidy()
2
>>> print ind.numChrom()
2
>>> # ...
>>> # as well as genotype
>>> print ind.allele(1)
1
>>> ind.setAllele(1,5)
>>> print ind.allele(1)
1
>>> # you can also use an overloaded function
>>> # with a second parameter being the ploidy index
>>> print ind.allele(1,1) # second locus at the second copy of chromosome
1
>>> # other information
>>> print ind.affected()
False
>>> print ind.affectedChar()
U
>>> ind.setAffected(1)
>>> print ind.affectedChar()
A
>>> print ind.sexChar()
M
>>>
```

Again, you will very seldom have to use these functions directly unless when you write pure Python operators.

2.3.2 Information fields

The information fields are information that is attached to each individual. For example, an individual may need `father_idx` and `mother_idx` to track pedigree information, may need `penetrance` to set affectedness.

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

- `pop.setInfoFields(fields)`
- `pop.addIndField(field)`

Note that changing information fields for a simulator is dangerous since all populations in a simulator share the same genotypic structure. You should use `addIndField` to all populations to avoid potential problems.

One can set/retrieve information at individual level:

- `ind.info(idx or field)`
- `ind.setInfo(idx or field)`
- `ind.arrInfo()`

or set at the population level

- `pop.indInfo(idx or field, order)`
- `pop.indInfo(idx or field, subPop, order)`
- `pop.setIndInfo(idx or field, [subPop])`
- `pop.arrIndInfo(order)`
- `pop.arrIndInfo(subPop, order)`

`idx` or `field` means that you can use field index obtained from `infoIdx(field)`, or use field name directly. `field` is easier to use but `idx` is faster. Although population information is kept in a population object linearly, there is no guarantee that they are ordered. If you would like to access `info` individual by individual, passing `order=True` will ensure that the returned information fields are ordered by individual order. If you only need to get a summary of some information fields, passing `order=False` will speed up the process.

For each individual, `ind.arrInfo()` will return `f1, f2, f3, ...` etc. for that individual. From a population point of view, `pop.arrIndInfo([subPop])` will return a list of `f1, f2, f3, ..., f1, f2, f3, ...`. Note that the order of individuals may not be kept in this (sub)population-wise array. That is to say, `pop.arrIndInfo()[0]` does not have to be the first field of the first individual. This property is also true for `setIndInfo(values, idx or name)`. That is to say, if you want to set information field for individuals in a population unordered, you can use

```
setIndInfo(values, idx)
```

Otherwise, you will have to use the less efficient way:

```
for i in range(pop.popSize()):
    pop.individual(i).setInfo(values[i], idx)
```

Note that `indInfo` is more convenient but it is less efficient (fields must be copied out) than `arrIndInfo`. To handle the returned value of `arrIndInfo`, you would usually do:

```
idx = pop.infoIdx('trait2')
step = pop.infoSize()
arr = pop.arrIndInfo(subPop=2)
for i in range(pop.subPopSize(2)):
  # note again that arr is writable.
  arr[idx + step*i] = something
```

2.4 Mating Scheme

Mating schemes define the rules of offspring generating. A mating scheme is required when a simulator is created.

2.4.1 Class `mating`

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

Details

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

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

Initialization

create a mating scheme

```
mating(numOffspring=1.0, *numOffspringFunc=NULL, maxNumOffspring=0,
mode=MATE_NumOffspring, newSubPopSize=[], newSubPopSizeExpr="",
*newSubPopSizeFunc=NULL)
```

By default, a mating scheme keeps a constant population size, generate one offspring per mating event. These can be changed using a variety of parameters. First, `newSubPopSize`, `newSubPopSizeExpr` and `newSubPopSizeFunc` can be used to specify subpopulation sizes of the offspring generation. `mode`, `numOffspring`, `maxNumOffspring` can be used to specify how many offspring will be produced for each mating event. This `mode` parameter can be one of

- `MATE_NumOffspring:` a fixed number of offspring for all families at this generation. If `numOffspring` is given, all generations use this fixed number. If `numOffspringFunc` is given, the number of offspring at each generation is determined by the value returned from this function.
- `MATE_NumOffspringEachFamily:` each family can have its own number of offspring. Usually, `numOffspringFunc` is used to determine the number of offspring of each family. If `numOffspring` is used, `MATE_NumOffspringEachFamily` is equivalent to `MATE_NumOffspring`.
- `MATE_GeometricDistribution:` a Geometric distribution with parameter `numOffspring` is used to determine the number of offspring of each family.
- `MATE_PoissonDistribution:` a Poisson distribution with parameter `numOffspring` is used to determine the number of offspring of each family.
- `MATE_BinomialDistribution:` a Binomial distribution with parameter `numOffspring` is used to determine the number of offspring of each family.
- `MATE_UniformDistribution:` a Uniform distribution `[a, b]` with parameter `numOffspring` (a) and `maxNumOffspring` (b) is used to determine the number of offspring of each family.

maxNumOffspring used when `numOffspring` is generated from a binomial distribution

mode can be one of `MATE_NumOffspring, MATE_NumOffspringEachFamily, MATE_GeometricDistribution, MATE_PoissonDistribution, MATE_BinomialDistribution, MATE_UniformDistribution`.

newSubPopSize an array of subpopulaitons sizes

newSubPopSizeExpr an expression that will return the new subpopulation size

newSubPopSizeFunc a function that accepts an `int` parameter(generation), an array of current population size and return an array of subpopulation sizes. This is usually easier to use than its expression version of this parameter.

numOffspring the number of offspring or *p* for a random distribution. Default to `1`. This parameter determines the number of offspring that a mating event will produce. Therefore, it determines the family size.

numOffspringFunc a Python function that returns the number of offspring or *p*

Member Functions

`x.clone()` deep copy of a mating scheme

2.4.2 Class `noMating`

a mating scheme that does nothing

Details

In this scheme, there is

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

Initialization

creat a scheme with no mating

```
noMating()
```

Note

There is no new `subPopsize` parameter.

Member Functions

`x.clone()` deep copy of a scheme with no mating

2.4.3 Class `binomialSelection`

a mating scheme that uses binomial selection, regardless of sex

Details

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

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

- haploid populaton is allowed.

Initialization

create a binomial selection mating scheme

```
binomialSelection(numOffspring=1., *numOffspringFunc=NULL,
maxNumOffspring=0, mode=MATE_NumOffspring, newSubPopSize=[],
newSubPopSizeExpr="", *newSubPopSizeFunc=NULL)
```

Please refer to class `mating` for parameter descriptions.

Member Functions

`x.clone()` deep copy of a binomial selection mating scheme

2.4.4 Class `randomMating`

a mating scheme of basic sexually random mating

Details

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

Initialization

create a random mating scheme

```
randomMating(numOffspring=1., *numOffspringFunc=NULL,
maxNumOffspring=0, mode=MATE_NumOffspring, newSubPopSize=[],
*newSubPopSizeFunc=NULL, newSubPopSizeExpr="", contWhenUniSex=True)
```

contWhenUniSex continue when there is only one sex in the population, default to `true`

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

maxNumOffspring used when `numOffspring` is generated from a binomial distribution

mode can be one of `MATE_NumOffspring, MATE_NumOffspringEachFamily, MATE_GeometricDistribution, MATE_PoissonDistribution, MATE_BinomialDistribution`

newSubPopSize an array of subpopulation sizes, should have the same number of subpopulations as the current population

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

newSubPopSizeFunc an function that have parameter `gen` and `oldSize` (current subpopulation size)

numOffspring number of offspring or *p* in some modes

numOffspringFunc a python function that determines the number of offspring or *p*

Member Functions

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

2.4.5 Class pyMating

a Python mating scheme

Details

Hybird mating scheme. This mating scheme takes a Python function that accepts both the parental and offspring populations and this function is responsible for setting genotype, sex of the offspring generation. During-mating operators, if needed, have to be applied from this function as well. Note that the subpopulaton size parameters are honored and the passed offspring generation has the desired (sub) population sizes. Parameters that control the number of offspring of each family are ignored.

This is likely an extremely slow mating scheme and should be used for experimental uses only. When a mating scheme is tested, it is recommended to implement it at the C++ level.

Initialization

create a Python mating scheme

```
pyMating(*func=NULL, newSubPopSize=[], newSubPopSizeExpr="",
*newSubPopSizeFunc=NULL)
```

func a Python function that accepts two parameters: the parental and the offspring populations. The offspring population is empty, and this function is responsible for setting genotype, sex etc. of individuals in the offspring generation.

Member Functions

`x.clone()` deep copy of a Python mating scheme

2.4.6 Determine the number of offspring during mating

The default value of `numOffspring` parameter makes a mating scheme produce one offspring per mating. This is the real random mating and should be used whenever possible. However, various situations require a larger family size or even changing the family size. 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:` if `numOffspringFunc` is not given, the number of offspring will be the constant `numOffspring` all the time. Otherwise, `numOffspringFunc(gen)` will be called **once** for each generation to get the number of offspring for the matings happen in this generation.
- `MATE_NumOffspringEachFamily:` `numOffspringFunc` has to be given and will be called whenever a mating happens. Since `numOffspringFunc` can be **any** Python function, this mode allows arbitrary model of assigning the number of offspring during mating. The mode can be slow though.
- `MATE_GeometricDistribution:` `numOffspring` or the result of `numOffspringFunc` (evaluated at each generation) will be 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` or result of `numOffspringFunc` (evaluated at each generation) will be 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` or the result of `numOffspringFunc` (evaluated at each generation) will be 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` or the result of `numOffspringFunc` (evaluated at each generation), and `maxNumOffspring` will 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.

2.4.7 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. Note that the expression uses variables from the local namespace.
- 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). For example, the following function defines a linear expansion demographic scenario where a population splits at generation 200 and starts expanding .

```
def lin_exp(gen, oldSize=[]):
  if gen < 200:   # burn in, constant population size
    return [1000]
  else:           # increase subpopulation sizes
    incSize = (10000-1000)/(500-200)/len(oldSize)
    return [oldSize[x]+incSize for x in range(0, len(oldSize))]
```

you can then use this function as follows

```
...randomMating(newSubPopSizeFunc=lin_exp) ...
```

2.4.8 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.

2.4.9 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 Operators

Operators are objects that act on populations. They define manipulations on populations. Operators are basic and important components in simuPOP.

2.5.1 Class `baseOperator`

base class of all classes that manipulate populations

Details

Operators are objects that act on populations. They can be applied to populations directly using their function forms, but they are usually managed and applied by a simulator.

Operators can be applied at different stages of the life cycle of a generation. More specifically, they 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)` 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/ending generation, gaps between applicable generations, or even specific generations. For example, you might want to start applying migrations after certain burn-in generations, or calculate certain statistics only sparsely.

Operators can have outputs. Output can be standard (terminal) or a file, which can vary with replicates and/or generations. Outputs from different operators can be accumulated to the same file to form table-like outputs.

Operators are applied to every replicate of a simulator by default. However, you can apply operators to one or a group of replicates using parameter `rep` or `grp`.

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. Output from several operators is allowed;
- `'>>>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);
- `''` supress output.

Initialization

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

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

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

begin the starting generation. Default to `0`. Negative numbers are allowed.

end stop applying after this generation. Negative numbers are allowed.

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

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

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

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

step the number of generations between active generations. Default to `1`.

Note

Negative generation numbers are allowed for `begin`, `end` and `at`. They are intepretted as `endGen + gen + 1`. For example, `begin = -2` in `simu.evolve(..., end=20)` starts at generation `19`.

Member Functions

`x.MPIReady()` determine if this operator can be used in a MPI module

`x.applicableGroup()` return applicable group

`x.applicableReplicate()` return applicable replicate

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

`x.canApplyDuringMating()` set if this operator can be applied *during-mating*

`x.canApplyPostMating()` set if this operator can be applied *post-mating*

`x.canApplyPreMating()` set if this operator can be applied *pre-mating*

`x.canApplyPreOrPostMating()` set of this operator can be applied *pre-* or *post-mating*

`x.clone()` deep copy of an operator

`x.diploidOnly()` determine if the operator can be applied only for diploid population

`x.haploidOnly()` determine if the operator can be applied only for haploid population

`x.infoField(idx)` get the information field specified by user (or by default)

`x.infoSize()` get the length of information fields for this operator

`x.setActiveGenerations(begin=0, end=-1, step=1, at=[])` set applicable generation parameters: `begin`, `end`, `step` and `at`

`x.setApplicableGroup(grp=GRP_ALL)` set applicable group

Default to `GRP_ALL` (applicable to all groups). Otherwise, the operator is applicable to only *one* group of replicates. Groups can be set in `simulator::setGroup()`.

`x.setApplicableReplicate(rep)` set applicable replicate

`x.setApplicableStage(stage)` set applicable stage. Another way to set `stage` parameter.

`x.setOutput(output="", outputExpr="")` set ouput stream, if was not set during construction

2.5.2 Types of operators

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 simulation. Less efficient. For example, a hybrid mutator `pyMutator` will determine if an allele will be mutated and call a user-defined Python function to mutate it.
- *pure Python*: written in Python. The same speed as Python. For example, a `varPlotter` can plot Python variables that are set by other operators.

You do not have to know the type of an operator to use them. The interfaces of them are all the same. Note that although it is possible to write pure Python operators to operate directly on populations, it might work very slowly compared to the built-in ones.

2.5.3 Applicable stages

Operators can be applied at *pre-*, *during-* or *post-mating*, or a combination of these stages. Note that it is possible for an operator to apply multiple times in a life cycle. For example, a save-to-file operator might be applied before and after mating to trace parental information. Also please refer to the class reference for setting the `stage` parameter.

Example 2.11: Operator stage

```
>>> d = dumper()
>>> print d.canApplyPreMating()
False
>>> print d.canApplyDuringMating()
False
>>> # so dumper is a post mating operator
>>> print d.canApplyPostMating()
True
>>>
```

2.5.4 Active generations

You can specify `begin`, `end`, `step`, and `at` parameters of an operator during initialization. For example,

Example 2.12: Set active generations of an operator

```
>>> simu = simulator(population(1),binomialSelection(), rep=3)
>>> op1 = output("a", begin=5, end=20, step=3)
>>> op2 = output("a", begin=-5, end=-1, step=2)
>>> op3 = output("a", at=[2,5,10])
>>> op4 = output("a", at=[-10,-5,-1])
>>> simu.evolve( [ pyEval(r"str(gen)+'\n'", begin=5, end=-1, step=2)],
...                                 end=10)
5
5
5
7
7
```

```
7
9
9
9
True
>>>
```

The last example displays variable `gen` for each replicate. Note that you can use negative generation number whenever you specify the `end` parameter of the evolution. In this case, generation -1 is the last generation (`end`), -2 is `end-1`, and so on.

2.5.5 Replicates and groups

Most operators are applied to every replicate of a simulator during evolution. However, you can apply operators to one or a group of replicates only. For example, you can initialize different replicates with different initial values and then start evolution. c.f. `simulator::setGroup` .

The most useful example is

```
output('\n',rep=REP_LAST)
```

which will output `\n` at the end of each generation. Here is an example of using replicate groups:

Example 2.13: Replicate group

```
>>> from simuUtil import *
>>> simu = simulator(population(1),binomialSelection(), rep=4,
...                                    grp=[1,2,1,2])
>>> simu.step([ pyEval(r"grp+3", grp=1),
...                            pyEval(r"grp+6", grp=2),
...                            output('\n', rep=REP_LAST)]
... )
4848
True
>>>
```

2.5.6 Output Specification

The following example shows the difference between `">"` and `">>"`

Example 2.14: operatoroutput

```
>>> simu = simulator(population(100), randomMating(), rep=2)
>>> simu.step(
...     preOps=[
...         initByFreq([0.2, 0.8], rep=0),
...         initByFreq([0.5, 0.5], rep=1) ],
...     ops = [
...         stat(alleleFreq=[0]),
...         pyEval('alleleFreq[0][0]', output='a.txt')
...     ]
... )
True
>>> # only from rep 1
>>> print open('a.txt').read()
```

```
0.48
>>>
>>> simu.step(
...     ops = [
...         stat(alleleFreq=[0]),
...         pyEval('alleleFreq[0][0]', output='>>a.txt')
...     ])
True
>>> # from both rep0 and rep1
>>> print open("a.txt").read()
0.2250.505
>>>
>>> outfile='>>>a.txt'
>>> simu.step(
...     ops = [
...         stat(alleleFreq=[0]),
...         pyEval('alleleFreq[0][0]', output=outfile),
...         output("\t", output=outfile),
...         output("\n", output=outfile, rep=0)
...     ],
... )
True
>>> print open("a.txt").read()
0.2250.5050.295
0.5
>>>
```

In the first simulator, all operators use `"a.txt"` (the same as `">a.txt"`). This file is repeatedly covered by other operators, so what we finally get is a new line written by `output("\n")`. The second simulator works fine by using `">>a.txt"`.

The output filename does not have to be fixed. If `outputExpr` parameter is used (`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.

Example 2.15: operatoroutputexpr

```
>>> outfile="'>>a'+str(rep)+'.txt'"
>>> simu.step(
...     ops = [
...         stat(alleleFreq=[0]),
...         pyEval('alleleFreq[0][0]', outputExpr=outfile)
...     ]
... )
True
>>> print open("a0.txt").read()
0.285
>>> print open("a1.txt").read()
0.495
>>>
```

2.5.7 Expressions and statements

Expressions are used extensively in operators so some basic knowledge of Python is required. If you know almost nothing about Python, please spend some time on the Python tutorial from Python website.

Unlike C/C++, assignments in Python do not return values. This is the most notable difference between Python expressions and statements:

- expressions consist of constants, variables, operators, functions, but *no* assignment, condition, loop etc. An expression returns a value when executed. An example of expression is `range(1,5)+10`.
- statements consist of arbitrary valid Python codes. A statement does *not* return a value when executed. An example of statement is `a=range(1,5)`.

2.5.8 `evaluate` function and `pyEval` and `pyExec` operators

Function `population::evaluate` and operator `pyEval/pyExec` will work in local namespaces. For example, if there are `a` and `b` in the main namespace and `a` in `pop`, `pop.evaluate('a')` will return `pop.vars()['a']`, `pop.evaluate('b')` will return global `b` since there is no `b` in the local namespace. If this is still too abstract, here is a real example

Example 2.16: python expression

```
>>> simu = simulator(population(10),noMating(), rep=2)
>>> # evaluate an expression in different areas
>>> print simu.vars(0)
{'rep': 0, 'gen': 0, 'grp': 0}
>>> print simu.population(0).evaluate("grp*2")
0
>>> print simu.population(1).evaluate("grp*2")
2
>>> print simu.population(0).evaluate("gen+1")
1
>>> # a statement (no return value)
>>> simu.population(0).execute("myRep=2+rep*rep")
>>> simu.population(1).execute("myRep=2*rep")
>>> print simu.vars(0)
{'rep': 0, 'myRep': 2, 'gen': 0, 'grp': 0}
>>>
```

- `simulator` creates a simulator with two replicates 0 and 1.
- We evaluate `grp*2` in different replicates and get different results.
- `gen` is not in either replicate's namespace so the global one will be used.
- Using statements can create variables in the local namespaces. (You can use `global` statement to create global variables if you are familiar with Python.)

`pyEval` and `pyExec` operators execute Python expressions/statements, *using the local namespaces*.

- `pyEval` (operator) evaluates a Python expression and returns its value, optionally executes a list of statements beforehand.
- `pyExec` (operator) executes a list of statements in the form of a multi-line string. No return value or output.

Here, `expr` is a simple string containing an expression that will return a value when executed; `stmts` is a string of statements, separated by `'\n'`.

For example, you can return a string of "`gen:rep`" using the following function

```
pop.evaluate(r'%d:%d' % (gen,rep))
```

but if you would like to change/create variables, you have to use statements like

```
pop.evalulate(rmyval, stmts=rmyval=rep+1)
```

Since you are executing Python statements, you can of course do it directly in Python. For example, the above function does exactly the following

```
pop.vars()['myval'] = pop.vars()['rep'] + 1
pop.vars()['myvar']
```

As a matter of fact, we seldom use `evaluate` function directly (maybe for debugging), usually

- we use expressions for dynamic parameters. For example:

  ```
  newSubPopSizeExpr=range(10,20)*1.2
  outputExpr= ' saveAt%s.txt % gen'
  ```

  These parameters will be evaluated whenever they are referred.

- we use expressions/statements in `pyEval/pyExec` operators. These statements will work in local namespaces. For example:

Example 2.17: Expression evaluation

```
>>> simu.step([ pyExec("myRep=2+rep*rep") ])
True
>>> print simu.vars(0)
{'rep': 0, 'selection': False, 'myRep': 2, 'gen': 0, 'grp': 0}
>>>
```

Because of the interactive nature of Python, it is very easy to write short programs, quote them in `r"'program"'` and put them into `pyEval/pyExec` operators.

2.6 Simulator

The population evolution is implemented in simulator, which manages and manipulates other components of simuPOP. Simulator is the basic and important simulation function or environment in simuPOP. Without it, we may only call the simuPOP 'POP'.

2.6.1 Class `simulator`

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

Details

Simulators combine three important components of simuPOP: population, mating scheme and operators 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 evolution 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.

Simulators separate 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.

Operators can be applied to a specific replicate, a group of replicates, or specific generations, determined by the `rep`, `grp`, `begin`, `end`, `step`, and `at` parameters.

Simulators can evolve a given number of generations (the `end` parameter of `evolve`), or evolve indefinitely until a certain type of operators called terminators 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.

Finally, a simulator can be saved to a file in the format of `'txt'`, `'bin'`, or `'xml'`. So we can stop a simulation and resume it at another time or on another machine. It is also a good idea to save a snapshot of a simulation every several hundred generations.

Initialization

create a simulator

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

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

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

matingScheme a mating scheme

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

rep number of replicates. Default to `1`.

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

Member Functions

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

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

field information field to be added

`x.addInfoFields(fields, init=0)` add information fields to all replicates

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

`x.clone()` deep copy of a simulator

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

Evolve to the `end` generation unless an operator (terminator) stops it earlier.

`ops` will be applied 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`. This is exactly how terminators work.

dry run mode. Default to `False`.

end ending generation. Default to `-1`. In this case, there is no ending generation and a simulator will only be ended by a terminator. Otherwise, it should be a number greater than current generation number.

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

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

Note: When `end = -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?

`x.gen()` return the current generation number

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

return a temporary reference of one of the populations. *'Reference'* means that the changes to the referred population will reflect to the one in simulator. *'Temporary'* means that the referred population might be invalid after evolution.

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

rep the index number of the replicate which will be obtained

`x.group()` return group indices

`x.numRep()` return the number of replicates

`x.pop(rep)` the `rep` replicate of this simulator

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

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. Therefore, the use of this function should be limited to *immediateafterretrival*. If you would like to get a persistent population, please use `getPopulation(rep)`.

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

The default format is `'txt'` but the output is not supposed to be read. `'bin'` has smaller size and should be used for large populations. `'xml'` format is most verbose and should be used when you would like to convert `simuPOP` populations to other formats.

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

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

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

`x.setAncestralDepth(depth)` set ancestral depth of all replicates

`x.setGen(gen)` set the current generation. Usually used to reset a simulator.

gen new generation index number

x.setGroup(grp) set groups for replicates

x.setMatingScheme(matingScheme) set mating scheme

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

x.vars(rep, subPop=-1) get simulator namespace. If `rep > 0` is given, return the namespace of replicate `rep`

2.6.2 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 three times. Generation 0, generation 1, and generation 2. 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`).

When you use `start=0, step=5, end=10` for your operator, it will be applied at generations `0,5,10` etc.

2.6.3 Operator calling sequence

In a simulation, operators are applied at different stages, pre-, during-, and post-mating. However, operators are not always active. They can be applied to certain generations or certain replicate(s) of a population. A simulator will always apply `preOps` and `postOps` operators, but will ask if an operator is active (by providing `rep, grp, gen` information) before it is called.

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. However, it 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.

2.6.4 Save and Load

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

Example 2.18: save and load a simulator

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

CHAPTER

THREE

Operator References

3.1 Python operators

This chapter will list all functions, types and operators by category.

3.1.1 Class `pyOperator`

the one and only Python operator???

Details

This operator accepts a function that can take the form of

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

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

Initialization

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

```
pyOperator(*func, *param=NULL, stage=PostMating, formOffGenotype=False,
passOffspringOnly=False, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

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

func a Python function. Its form is determined by other parameters.

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

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

Note

- Output to `output` or `outputExpr` is not supported. That is to say, you have to open/close/append to files explicitly in the Python function.
- This operator can be applied `Pre-`, `During-` or `Post-` mating and is applied `PostMating` by default. For example, if you would like to examine the fitness values set by a selector, a `PreMating` Python operator should be used.

Member Functions

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

`x.clone()` deep copy of a `pyOperator` operator

This is the single most powerful hybrid operator. Whenever you think that something is too complicated to be done by standard operators, you can do it here in Python. This operator accepts a Python function which accepts a population and optionally a parameter. To use this operator, 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. `para` will be omitted.

- use `pyOperator` like

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

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

When `pyOperator` is called, it will simply pass the accepted population to the function. If your function returns `False`, the simulation will be stopped.

This operator allows implementation of arbitrarily complicated operators, at a cost of efficiency. Of course, 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
... #end
...
```

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
... )
0.401700        0.206200
0.395850        0.213400
0.420450        0.212450
0.431450        0.206000
True
>>>
```

Note that

- Currently, `pyOperator` does not support parameters `output` and `outputExpr`. This is because of the incompatibility between Python and underlying C++ in the way of handling file I/O stream. Consequently, you will have to handle file input/output by yourself through `param` parameter. Be careful that you ***can not*** mix output of `pyOperator` with those of other (normal) operators.
- If parameter `param` is ignored, `myOperator` must be without `para` as well. Note that you can pass arbitrary number of parameters by putting them into a tuple and passing to `myOpeartor`.
- Since you can attach any information to a population, you can in practice use `pop.dvars()` to pass parameters.
- `pyOperator` is a post-mating operator by default. Remember to use `stage` parameter to change this when necessary.

`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)
```

Two additional parameters are:

- `formOffGenotype`: (default to `False`) By default, a mating scheme will set the genotype of offspring by copying one of the parental chromosomes. However, if `formOffGenotype` is `True`, the mating scheme will let you do the job. You will have to set offspring genotype and sex by yourself, using, most likely, a recombinator.
- `passOffspringOnly`: In case that your function will only deal with offspring, you can set this parameter to `True` and use a short form of the function.

Note that 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

Details

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

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

Initialization

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

```
pyIndOperator(*func, loci=[], *param=NULL, stage=PostMating,
formOffGenotype=False, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

func a Python function that accepts an individual and optional genotype and parameters.

infoFields if given, `func` is expected to return an array of the same length and fill these `infoFields` of an individual.

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

Member Functions

`x.apply(pop)` apply the `pyIndOperator` operator to one population

`x.clone()` deep copy of a `pyIndOperator` operator

Another general Python operator is `pyIndOperator` which is similar to `pyOpertor` but it passes the user individuals, rather than the whole population.

```
def func(ind, param):
  ind.setInfo(param[0], 'myinfo)
pyIndOperator(func=func, param=(1,))
```

is the same as

```
def func(pop, param):
  for ind in pop.individuals():
    ind.setInfo(param[0], 'myinfo')
pyIndOperator(func=func, param=(1,))
```

The `pyIndOperator` may have some performance advantages over `pyOperator` in some cases.

3.2 Initialization

3.2.1 Class `initializer`

initialize alleles at the start of a generation

Details

Initializers are used to initialize populations before evolution. They are set to be `PreMating` operators by default. simuPOP provides three initializers. One assigns alleles by random, one assigns a fixed set of genotypes, and the last one calls a user-defined function.

Initialization

create an initializer. default to be always active

```
initializer(subPop=[], indRange=[], loci=[], atPloidy=-1,
maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=-1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Member Functions

`x.clone()` deep copy of an initializer

`x.initSexIter()` ???

`x.nextSex()` ???

`x.setRanges(pop)` set the range of a population`pop`???

3.2.2 Class `initByFreq` (Function form: `InitByFreq`)

initialize genotypes by given allele frequencies, and sex by male frequency

Details

This operator accepts `alleleFreq` or `alleleFreqs`.??? The first one ignores subpopulation structure while the second one gives different initial allele frequencies to different `subpop` or ranges. Allele frequencies can differ in `subpop`. Sex is also assigned randomly.

Initialization

randomly assign alleles according to given allele frequencies

```
initByFreq(alleleFreq=[], identicalInds=False, subPop=[],
indRange=[], loci=[], atPloidy=-1, maleFreq=0.5, sex=[],
stage=PreMating, begin=0, end=1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

alleleFreq an array of allele frequencies. The sum of all the frequencies must be 1; or for a matrix of allele frequencies, each row corresponses to a subpopulation.

atPloidy initialize which copy of chromosomes. Default to all.

identicalInds whether or not make individual genotypies identical in all subpopulation. If `True`, this operator will randomly generate genotype for an individual and spread it to the whole subpopulation in the given range.

indRange a `[begin, end]` pair of the range of absolute indices 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 indices at which initialization will be done. If empty, apply to all loci.

locus a shortcut to `loci`

maleFreq male frequency. Default to `0.5`. Sex will be initialized with this parameter.

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`

subPop an array specifies applicable subpopulations

Member Functions

`x.apply(pop)` apply operator `initByFreq`???

`x.clone()` deep copy of the operator `initByFreq`

Example

Example 3.3: Operator initByFreq

```
>>> simu = simulator(
...     population(subPop=[2,3], loci=[5,7], maxAllele=1),
...     randomMating(), rep=1)
```

```
>>> simu.step([
...     initByFreq(alleleFreq=[ [.2,.8],[.8,.2]]),
...     dumper(alleleOnly=True)
...   ])
individual info:
sub population 0:
   0: FU 10000 1111111 | 11111 1111111
   1: FU 10000 1101101 | 11111 1111111
sub population 1:
   2: MU 00000 0000000 | 00000 0001000
   3: MU 00001 0000000 | 00000 0000100
   4: MU 00000 0000100 | 00001 0111000
End of individual info.


No ancenstral population recorded.
True
>>>
```

3.2.3 Class `initByValue` (Function form: `InitByValue`)

initialize genotype by value and then copy to all individuals

Details

`initByValue` operator 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 assign 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 values. Each item of values will be assigned to each `subPop` or `indRange`.

Initialization

initialize populations by given alleles

```
initByValue(value=[], loci=[], atPloidy=-1, subPop=[], indRange=[],
proportions=[], maleFreq=0.5, sex=[], stage=PreMating, begin=0,
end=1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

atPloidy initialize which copy of chromosomes. Default to all.

indRange a `[begin, end]` pair of range of individuals; or an array of `[begin, end]` pairs.

loci a vector of loci indices. If empty, apply to all loci.

locus a shortcut to `loci`

maleFreq male frequency

proportions an array of percentages for each item in values. 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`

subPop an array of applicable subpopulations. If values are given, should have equal length to values.

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 chromosome). This parameter can also be an array of arrays of genotypes of one individual. Should have length one or equal to `subpop` or ranges or proportion. If value is an array of values, it should have the same length as `subpop`, `indRange` or `proportions`.

Member Functions

`x.apply(pop)` apply operator `initByValue`???

`x.clone()` deep copy of the operator `initByValue`

Example

Example 3.4: Operator initByValue

```
>>> simu = simulator( population(subPop=[2,3], loci=[5,7]),
...     randomMating(), rep=1)
>>> simu.step([
...     initByValue([1]*5 + [2]*7 + [3]*5 +[4]*7),
...     dumper(alleleOnly=True)])
individual info:
sub population 0:
   0: MU   3  3  3  3  3   2  2  2  2  2  2  2 |   1  1  1  1  1   4  4
4  4  4  4  4
   1: FU   3  3  3  3  3   4  4  4  4  4  4  4 |   1  1  1  1  1   2  2
2  2  2  2  2
sub population 1:
   2: MU   1  1  1  1  1   4  4  4  4  4  4  4 |   3  3  3  3  3   4  4
4  4  4  4  4
   3: FU   1  1  1  1  1   4  4  4  4  4  4  4 |   3  3  3  3  3   4  4
4  4  4  4  4
   4: MU   3  3  3  3  3   4  4  4  4  4  4  4 |   3  3  3  3  3   4  4
4  4  4  4  4
End of individual info.


No ancenstral population recorded.
True
>>>
```

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

initialize genotype by value and then copy to all individuals

Details

`Spread(ind, subPop)` spreads the genotype of `ind` to all individuals in an array of subpopulations. The default value of `subPop` is the subpopulation where `ind` resides.

Initialization

copy genotypes of `ind` to all individuals in `subPop`

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

Member Functions

`x.apply(pop)` apply operator `spread`???

`x.clone()` deep copy of the operator `spread`

3.2.5 Class `pyInit` (Function form: `PyInit`)

a hybrid initializer???

Details

`pyInit` is a hybrid initializer. User should define a function with parameters allele, ploidy and subpopulation indices, and return an allele value. Users of this operator must supply a Python function with parameter `(index, ploidy, subpop)`. This operator will loop through all individual in each subpopulation and call this function to initialize populations. The arrange of parameters allows different initialization scheme for each subpop.

Initialization

initialize populations using given user function

```
pyInit(*func, subPop=[], loci=[], atPloidy=-1, indRange=[],
maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

atPloidy initialize which copy of chromosomes. Default to all.

func a Python function with parameter `(index, ploidy, subpop)`, where

- `index` is the allele index ranging from `0` to `totNumLoci(-1)`,
- `ploidy` is the index of the copy of chromosomes)
- `subpop` is the subpopulation index.

The return value of this function should be an integer.

loci a vector of loci indices. If empty, apply to all loci.

locus a shortcut to `loci`

stage default to `PreMating`

Member Functions

`x.apply(pop)` apply operator `pyInit`???

`x.clone()` deep copy of the operator `pyInit`

Example

Example 3.5: Operator pyInit

```
>>> def initAllele(ind, p, sp):
...   return sp + ind + p
...
```

```
>>> simu = simulator(
...     population(subPop=[2,3], loci=[5,7]),
...     randomMating(), rep=1)
>>> simu.step([
...     pyInit(func=initAllele),
...     dumper(alleleOnly=True, dispWidth=2)])
individual info:
sub population 0:
   0: FU   0  1  2  3  4   6  7  8  9 10 11 12 |   0  1  2  3  4   6  7
8  9 10 11 12
   1: MU   1  2  3  4  5   5  6  7  8  9 10 11 |   1  2  3  4  5   5  6
7  8  9 10 11
sub population 1:
   2: FU   1  2  3  4  5   7  8  9 10 11 12 13 |   2  3  4  5  6   7  8
9 10 11 12 13
   3: FU   1  2  3  4  5   6  7  8  9 10 11 12 |   1  2  3  4  5   6  7
8  9 10 11 12
   4: FU   2  3  4  5  6   7  8  9 10 11 12 13 |   2  3  4  5  6   6  7
8  9 10 11 12
End of individual info.


No ancenstral population recorded.
True
>>>
```

3.3 Migration

3.3.1 Class `migrator`

migrate individuals from a (sub) population to another (sub) population

Details

Migrator is the only way to mix genotypes of several subpopulations because mating is strictly within subpopulations in simuPOP. Migrators are quite flexible in simuPOP in the sense that

- Migration can happen from and to a subset of subpopulations.
- Migration can be done by probability, proportion or by counts. In the case of probability, if the migration rate from subpopulation `a` to `b` is `r`, then everyone in subpopulation `a` will have this probability to migrate to `b`. In the case of proportion, exactly `r*size_of_subPop_a` individuals (chosen by random) will migrate to subpopulation `b`. In the last case, a given number of individuals will migrate.
- New subpopulation can be generated through migration. You simply need to migrate to a new subpopulation number.

Initialization

create a migrator

```
migrator(rate, mode=MigrByProbability, fromSubPop=[], toSubPop=[],
stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

fromSubPop an array of 'from' subpopulations. Default to all. If a single `subpop` is specified, `[]` can be ignored. I.e., `[a]` is equvalent to `a`.

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 `r*ones(m,n)`???.

stage default to `PreMating`

toSubPop an array of 'to' subpopulations. Default to all subpopulations. If a single `subpop` 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 size after migration, you can use the `newSubPopSize` or `newSubPopSizeExpr` parameter of a mating scheme.
- `rate` is a matrix with dimensions determined by `fromSubPop` and `toSubPop`. By default, `rate` is a matrix with element `r(i,j)`, where `r(i, j)` is the migration rate, probability or count from subpopulation `i` to `j`. If `fromSubPop` and/or `toSubPop` are given, migration will only happen between these subpopulations. An extreme case is 'point migration', `rate=[[r]], fromSubPop=a, toSubPop=b` which migrate from subpopulation `a` to `b` with given rate `r`.???

Member Functions

`x.apply(pop)` apply the `migrator`

`x.clone()` deep copy of a migrator

`x.rate()` return migration rate

`x.setRates(rate, mode)` set migration rate

Format should be `0-0 0-1 0-2, 1-0 1-1 1-2, 2-0, 2-1, 2-2`. For mode `MigrByProbability` or `MigrByProportion`, `0-0,1-1,2-2` will be set automatically regardless of input.

Operator `migrator` is used to migrate from `'fromSubPop'` to `'toSubPop'`. From and to subpop can be a number or an array of subpopulations. The migration probability/rate/counts from `i->j` is specified in the rate matrix. The `'fromSubPop'` and `'toSubPop'` are default to all subpopulations.

An detailed example can be found in 'some real examples' -> 'complex Migration Scheme' section.

3.3.2 Functions (Python) `MigrIslandRates, MigrStepstoneRates` (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. Several functions are defined in simuUtil.py, however, for easy use of popular migration models:

- `MigrIslandRates(r, n)` returns a migration matrix

$$\begin{pmatrix} 1-r & \frac{r}{n-1} & \dots & \dots & \frac{r}{n-1} \\ \frac{r}{n-1} & 1-r & \dots & \dots & \frac{r}{n-1} \\ & & \dots & & \\ \frac{r}{n-1} & \dots & \dots & 1-r & \frac{r}{n-1} \\ \frac{r}{n-1} & \dots & \dots & \frac{r}{n-1} & 1-r \end{pmatrix}$$

- `MigrStepstoneRates(r, n, circular=False)` returns a migration matrix

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

  and if `circular=True`, returns

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

3.3.3 Class `pyMigrator`

a more flexible Python migrator

Details

This migrator can be used in two ways

- define a function that accepts a generation number and returns a migration rate matrix. This can be used in the varying migration rate cases.
- define a function that accepts individuals etc, and returns the new subpopulation ID.

More specifically, `func` can be

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

Initialization

create a hybrid migrator

```
pyMigrator(*rateFunc=NULL, mode=MigrByProbability, fromSubPop=[],
toSubPop=[], *indFunc=NULL, loci=[], *param=NULL, stage=PreMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

indFunc a Python function that accepts an individual, optional genotype and parameter, then returns a subpopulation id. This method can be used to separate a population according to individual genotype.

rateFunc a Python function that accepts a generation number, current subpopulation sizes, and returns a migration rate matrix. The migrator then migrate like a usual migrator.

stage default to `PreMating`

Member Functions

x.apply(pop) apply a pyMigrator

x.clone() deep copy of a pyMigrator

For even more complicated migration schemes, you may DIY using a pyMigrator. This operator is not strictly hybrid since it does not call Python function. However, it takes a carray as subpopulation IDs for each individual. pyMigrator then complete migration according its content.

Note that the application sequence of the operators is initByFreq, dumper, pyMigrator and then dumper again since its stage is set to PrePostMating.

3.3.4 Class splitSubPop (Function form: SplitSubPop)

split a subpopulation

Details

Initialization

split a subpopulation or the whole population as subpopulation 0

```
splitSubPop(which=0, sizes=[], proportions=[], subPopID=[],
randomize=True, stage=PreMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

proportions proportions of new subpopulations. Should be added up to 1. Optionally, you can use one of subPopID or proportions to split. ???

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. If given, should have the same length as subPop or proportions.??? Since subpop with negative id will be removed. You can remove part of a subpop by setting a new negative ID.???

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

Member Functions

x.apply(pop) apply a splitSubPop operator

x.clone() deep copy of a splitSubPop operator

3.3.5 Class mergeSubPops (Function form: MergeSubPops)

merge subpopulations

Details

This operator merges subpopulations subPops (the only parameter???) to a single subpopulation. If subPops is ignored, all subpopulations will be merged.

Initialization

merge subpopulations

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

subPops subpopulatinos to be merged. Default to all.

Member Functions

`x.apply(pop)` apply a `mergeSubPops` operator

`x.clone()` deep copy of a `mergeSubPops` operator

3.4 Mutation

The only difference between the following mutators is the way they actually mutate an allele, and corresponding input parameters.

Mutators record the number of mutation events at each loci. You can retrieve this information using

```
mut.mutationCount(locus)
mut.mutationCounts()
```

where `mut` is any mutator and `locus` is locus index.

3.4.1 Class `mutator`

mutator class

Details

The base class of all functional mutators. It is not supposed to be called directly.Every mutator can specify `rate` (equal rate or different rates for different loci) and a vector of applicable loci (default to all but should have the same length as `rate` if `rate` has length greater than one).Maximum allele can be specified as well but more parameter, if needed, should be implemented by individual mutator classes.There are number of possible allelic states. Most theoretical studies assume an infinite number of allelic states to avoid any homoplasy. If it facilitates any analysis, this is however extremely unrealistic.

Initialization

create a mutator

```
mutator(rate=[], loci=[], maxAllele=0, output=">", outputExpr="",
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

All mutators have the following common parameters. However, the actual meaning of these parameters may vary according to different model. The only differences between the following mutators are they way they actually mutate an allele, and corresponding input parameters. Mutators record the number of mutation events at each locus.

loci a vector of loci indices. Will be ignored only when single rate is specified. Default to all loci.

maxAllele maximum allowable allele. Interpreted by each sub mutaor class. Default to `pop.maxAllele()`.

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

Member Functions

`x.apply(pop)` apply a mutator

`x.clone()` deep copy of a `mutator`

`x.maxAllele()` return maximum allowable allele number

`x.mutate(allele)` describe how to mutate a single allele

`x.mutationCount(locus)` return mutation count at `locus`

`x.mutationCounts()` return mutation counts

`x.rate()` return the mutation rate

`x.setMaxAllele(maxAllele)` set maximum allowable allele

`x.setRate(rate, loci=[])` set an array of mutation rates

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

K-Allele Model mutator.

Details

This mutator mutate an allele to another allelic state with equal probability. The specified mutation rate is actually the 'probability to mutate'. So the mutation rate to any other allelic state is actually $(rate/(K-1))$, where K is specified by parameter `maxAllele`. You can also specify states for this mutator. If the state parameter is given, all alleles must be one of the states, and mutation will happen among them. states is defaulted to `1-maxAllele`.???

Initialization

create a K-Allele Model mutator

```
kamMutator(rate=[], loci=[], maxAllele=0, output=">", outputExpr="",
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

loci a vector of loci indices. Will be ignored only when single rate is specified. Default to all loci.

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 `rates/(K-1)`.

Member Functions

`x.clone()` deep copy of a `kamMutator`

`x.mutate(allele)` mutate to a state other than current state with equal probability

Example

Example 3.6: Operator kamMutator

```
>>> simu = simulator(population(size=5, loci=[3,5]), noMating())
>>> simu.step([
...     kamMutator( rate=[.2,.6,.5], loci=[0,2,6], maxAllele=9),
...     dumper(alleleOnly=True)])
individual info:
sub population 0:
   0: MU   1  0  7   0  0  0  0  0 |   0  0  7   0  0  0  9  0
   1: MU   0  0  0   0  0  0  0  0 |   7  0  9   0  0  0  0  0
   2: MU   0  0  0   0  0  0  0  0 |   0  0  0   0  0  0  6  0
   3: MU   0  0  6   0  0  0  5  0 |   0  0  0   0  0  0  0  0
   4: MU   0  0  4   0  0  0  1  0 |   6  0  4   0  0  0  0  0
End of individual info.


No ancenstral population recorded.
True
>>>
```

3.4.3 Class `smmMutator` (Function form: `SmmMutate`)

stepwise mutation model

Details

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 loci (VNTR), the allele value is generally taken as the number of tandem repeats in the DNA sequence.

Initialization

create a SMM mutator

```
smmMutator(rate=[], loci=[], maxAllele=0, incProb=0.5, output=">",
outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

The stepwise mutation model (SMM) is developed for allozymes. It provides better description for these kinds of evolutionary processes. Please see `mutator` for the description of other parameters.

incProb probability to increase allele state. Default to `0.5`.

Member Functions

`x.clone()` deep copy of a `smmMutator`

`x.mutate(allele)` mutate according to the SMM model ???

Example

Example 3.7: Operator smmMutator

```
>>> simu = simulator(population(size=3, loci=[3,5]), noMating())
>>> simu.step([
```

```
...     initByFreq( [.2,.3,.5]),
...     smmMutator(rate=1,  incProb=.8),
...     dumper(alleleOnly=True, stage=PrePostMating)])
individual info:
sub population 0:
   0: FU   1  2  1   2  1  2  2  2 |   1  2  1   1  0  2  1  1
   1: MU   2  1  0   0  1  1  2  0 |   2  2  0   1  1  0  2  2
   2: MU   0  0  0   1  2  2  0  2 |   2  2  1   0  2  2  1  2
End of individual info.


No ancenstral population recorded.
individual info:
sub population 0:
   0: FU   2  3  2   1  2  3  3  1 |   2  1  0   0  1  3  2  2
   1: MU   3  2  1   1  2  0  3  1 |   3  1  1   2  2  1  3  3
   2: MU   1  0  1   0  3  3  0  1 |   3  1  2   1  3  3  0  3
End of individual info.


No ancenstral population recorded.
True
>>>
```

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

generalized stepwise mutation model

Details

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

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

Initialization

create a `gsmMutator`

```
gsmMutator(rate=[], loci=[], maxAllele=0, incProb=0.5, p=0,
*func=NULL, output=">", outputExpr="", stage=PostMating, begin=0,
end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

The generalized stepwise mutation model (GMM) is developed for allozymes. It provides better description for these kinds of evolutionary processes. Please see `mutator` for the description of other parameters.

func return number of steps. No parameter.???

incProb probability to increase allele state. Default to `0.5`.

Member Functions

x.clone() deep copy of a gsmMutator

x.mutate(allele) mutate according to the GSM model

3.4.5 Class pyMutator (Function form: PyMutate)

hybrid mutator

Details

Hybrid mutator. Mutation rate etc. are set just like others and you are supposed to provide a Python function to return a new allele state given an old state. pyMutator will choose an allele as usual and call your function to mutate it to another allele.

Initialization

create a pyMutator

```
pyMutator(rate=[], loci=[], maxAllele=0, *func=NULL, output=">",
outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Member Functions

x.clone() deep copy of a pyMutator

x.mutate(allele) mutate according to the mixed model

Example

Example 3.8: Operator pyMutator

```
>>> def mut(x):
...   return 8
...
>>> simu = simulator(population(size=3, loci=[3,5]), noMating())
>>> simu.step([
...   pyMutator(rate=.5, loci=[3,4,5], func=mut),
...   dumper(alleleOnly=True)])
individual info:
sub population 0:
   0: MU   0  0  0   8  8  8  0  0 |   0  0  0   8  0  0  0  0
   1: MU   0  0  0   0  0  0  0  0 |   0  0  0   0  8  0  0  0
   2: MU   0  0  0   8  0  0  0  0 |   0  0  0   8  8  8  0  0
End of individual info.


No ancenstral population recorded.
True
>>>
```

3.4.6 Class pointMutator (Function form: PointMutate)

point mutator

Details

Mutate specified individuals at a 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.

Initialization

create a `pointMutator`

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

Please see `mutator` for the description of other parameters.

inds individuals who will mutate

toAllele allele that will be mutate to

Member Functions

`x.apply(pop)` apply a `pointMutator`

`x.clone()` deep copy of a `pointMutator`

`x.mutationCount(locus)` return mutation count at `locus`

`x.mutationCounts()` return mutation counts

3.5 Recombination

3.5.1 Class `recombinator`

recombination

Details

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

- it only works for diploid (and for females in haplodiploid) populations.
- the recombination rate must be comprised between `0.0` and `0.5`. A recombination rate of `0.0` means that the loci are completely linked, and thus behave together as a single linked locus. A recombination rate of `0.5` is equivalent to free 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.

Initialization

recombine chromosomes from parents

```
recombinator(intensity=-1, rate=[], afterLoci=[], maleIntensity=-1,
maleRate=[], maleAfterLoci=[], begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

afterLoci an array of locus indices. 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.

intensity intensity of recombination. The actually recombination rate between two loci is determined by `intensity*locus distance` between them.

maleAfterLoci if given, males will recombine at different locations. This is rarely used.???

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()`. If `totNumLoci`, the last item can be ignored.??? The recombination rates are independent of locus distance.

Note

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

Member Functions

`x.applyDuringMating(pop, offspring, *dad=NULL, *mom=NULL)` apply the recombinator during mating???

`x.clone()` deep copy of a recombinator

`x.recCount(locus)` return recombination count

`x.recCounts()` return recombination counts

Example

Example 3.9: Operator recombinator

```
>>> simu = simulator(population(4, loci=[4,5,6],
...     infoFields=['father_idx', 'mother_idx']),
...     randomMating())
>>> simu.step([
...   parentsTagger(),
...   ],
...   preOps = [initByFreq([.2,.2,.4,.2]), dumper(alleleOnly=True) ],
...   postOps = [ dumper(alleleOnly=True)]
... )
individual info:
sub population 0:
   0: MU   2  2  1  1   3  0  2  3  2   0  3  2  0  0  2 |   2  2  1  0
0  2  0  1  2   3  1  3  1  0  3
   1: FU   1  0  2  2   1  3  0  1  0   2  0  3  3  2  0 |   1  2  2  2
1  2  2  1  2   1  1  2  2  3  2
   2: MU   2  1  0  2   2  3  3  1  0   1  2  3  2  3  2 |   2  2  3  3
2  2  2  3  2   1  2  3  2  2  3
   3: FU   3  3  1  1   2  2  2  3  0   1  1  1  0  2  1 |   2  2  1  3
2  2  2  0  2   0  1  1  0  2  2
End of individual info.


No ancenstral population recorded.
individual info:
sub population 0:
   0: FU   2  2  3  3   2  3  3  1  0   1  2  3  2  3  2 |   1  0  2  2
1  3  0  1  0   2  0  3  3  2  0
   1: MU   2  2  1  0   0  2  0  1  2   0  3  2  0  0  2 |   1  0  2  2
1  2  2  1  2   1  1  2  2  3  2
   2: FU   2  2  3  3   2  2  2  3  2   1  2  3  2  2  3 |   2  2  1  3
2  2  2  0  2   1  1  1  0  2  1
   3: FU   2  1  0  2   2  2  2  3  2   1  2  3  2  2  3 |   2  2  1  3
2  2  2  3  0   0  1  1  0  2  2
End of individual info.


No ancenstral population recorded.
True
>>> simu.step([
...   parentsTagger(),
...   recombinator(rate=[1,1,1], afterLoci=[2,6,10])
...   ],
...   postOps = [ dumper(alleleOnly=True)]
... )
individual info:
sub population 0:
```

```
   0: MU   2  1  0  3   2  2  2  3  0   0  1  3  2  2  3 |   2  2  1  2
1  2  2  1  2   1  1  2  0  0  2
   1: MU   2  2  1  3   2  2  2  0  2   1  2  1  0  2  1 |   2  2  1  2
0  2  0  1  2   1  1  2  0  0  2
   2: MU   2  2  1  2   2  2  2  3  2   0  1  3  2  2  3 |   2  2  1  2
1  2  2  1  2   1  1  2  0  0  2
   3: FU   2  2  3  3   2  2  2  0  2   1  2  1  0  2  1 |   2  2  1  2
1  2  2  1  2   1  1  2  0  0  2
End of individual info.


No ancenstral population recorded.
True
>>>
```

This operator takes similar parameters as a mutator. However, because of potentially uneven allelic distance, you should use one of the two parameters listed in the last section.

The following example forces recombination (with rate 1, an unrealistic value since the maximum recombination rate should be .5) at loci 2, 6 and 10. Here I use a `parentsTagger` to mark the parents of each individual so you can (if you have enough patience) see exactly how recombination works.

Example 3.10: Recombinator

```
>>> simu = simulator(population(4, loci=[4,5,6],
...         infoFields=['father_idx', 'mother_idx']),
...         randomMating())
>>> simu.step([
...     parentsTagger(),
...     ],
...     preOps = [initByFreq([.2,.2,.4,.2]), dumper(alleleOnly=True) ],
...     postOps = [ dumper(alleleOnly=True)]
... )
individual info:
sub population 0:
   0: MU   1  2  0  0   2  0  2  3  3   2  2  1  0  1  1 |   1  0  0  0
1  2  0  0  2   1  2  0  2  0  0
   1: FU   2  2  3  2   2  0  2  2  1   2  3  1  2  2  0 |   1  0  0  3
2  3  1  0  2   0  2  2  3  1  0
   2: FU   3  0  1  0   3  0  1  2  3   0  2  2  0  3  0 |   2  2  2  2
2  2  3  2  1   3  2  2  0  2  2
   3: MU   2  1  3  2   3  1  3  2  3   0  3  2  1  3  3 |   1  1  2  2
2  2  1  3  0   2  3  2  0  1  0
End of individual info.


No ancenstral population recorded.
individual info:
sub population 0:
   0: MU   1  2  0  0   2  0  2  3  3   1  2  0  2  0  0 |   3  0  1  0
2  2  3  2  1   3  2  2  0  2  2
   1: FU   1  2  0  0   2  0  2  3  3   1  2  0  2  0  0 |   3  0  1  0
3  0  1  2  3   0  2  2  0  3  0
   2: MU   2  1  3  2   2  2  1  3  0   2  3  2  0  1  0 |   2  2  2  2
3  0  1  2  3   0  2  2  0  3  0
   3: FU   1  0  0  0   1  2  0  0  2   2  2  1  0  1  1 |   2  2  2  2
```

```
3  0  1  2  3   3  2  2  0  2  2
End of individual info.


No ancenstral population recorded.
True
>>> simu.step([
...     parentsTagger(),
...     recombinator(rate=[1,1,1], afterLoci=[2,6,10])
...     ],
...     postOps = [ dumper(alleleOnly=True)]
... )
individual info:
sub population 0:
   0: FU   3  0  1  0   3  0  1  3  3   1  2  2  0  3  0 |   1  2  0  0
2  2  3  3  3   1  2  2  0  2  2
   1: MU   1  2  0  0   3  0  1  3  3   0  2  0  2  0  0 |   2  2  2  2
3  0  1  3  0   0  2  2  0  1  0
   2: FU   1  2  0  0   2  0  2  2  3   1  2  2  0  3  0 |   2  1  3  2
2  2  1  2  3   2  3  2  0  3  0
   3: MU   3  0  1  0   2  0  2  2  3   0  2  0  2  0  0 |   3  0  1  0
2  0  2  2  1   3  2  0  2  0  0
End of individual info.


No ancenstral population recorded.
True
>>>
```

Recombinations after each locus will be recorded. You can retrieve this information through functions

```
rec.recCount(locus)
rec.recCounts()
```

where `rec` is the recombinator, `locus` is locus index.

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 since 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 alone can not do selection! Only mating schemes can actually select on individuals.
- 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`

genetic selection

Details

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

- `PreMating` selectors assign fitness values to each individual.
- During sexless mating (e.g. `binomialSelection`???), 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 and mate.

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 since we do not know who will determine the number of offspring if two parents are involved.)All of the selection operators, when applied, will set a variable `fitness` and an indicator so that 'selector-aware' mating scheme can select individuals according to these values. Hence, two consequences are stated below:

- selector alone can not do selection! Only mating schemes can actually select individuals.
- 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`.

Initialization

create a selector

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

subPop a shortcut to `subPops=[subPop]`

subPops subpopulations that the selector will apply to. Default to all.

Member Functions

`x.apply(pop)` set fitness to all individuals???

`x.clone()` deep copy of a selector

`x.indFitness(*, gen)` calculate/return the fitness value???

3.6.3 Class `mapSelector` (Function form: `MapSelector`)

selection according to the genotype at one locus

Details

This map selector implements selection at one locus. A user provided dictionary (map) of genotypes will be used in this selector to set each individual's fitness value.

Initialization

create a map selector

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

fitness a dictionary of fitness values. The genotype must be in the form of `'a-b'` for a single locus, and `'a-b|c-d|e-f'` for multi-loci.

loci the locus indices. The genotypes of these loci will be examed.

locus the locus index. The genotype of this locus will be examed.???

output and other parameters please refer to help(baseOperator.__init__)???

phase if `True`, genotypes `a-b` and `b-a` will have different fitness values. Default to false.

Member Functions

`x.clone()` deep copy of a map selector

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

The following example 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 ($s_1 = .1$, $s_2 = .2$).

Example 3.11: map selector

```
>>> simu = simulator(
...         population(size=1000, ploidy=2, loci=[1], infoFields=['fitness']),
...         randomMating())
>>> s1 = .1
>>> s2 = .2
>>> simu.evolve([
...         stat( alleleFreq=[0], genoFreq=[0]),
...         mapSelector(locus=0, fitness={'0-0':(1-s1), '0-1':1, '1-1':(1-s2)}),
...         pyEval(r"'%.4f\n' % alleleFreq[0][1]", step=100)
...         ],
...         preOps=[    initByFreq(alleleFreq=[.2,.8])],
...         end=300)
0.7820
0.3725
0.3610
0.3205
True
>>>
```

3.6.4 Class `maSelector` (Function form: `MaSelect`)

multiple allele selector (selection according to wildtype or diseased alleles)

Details

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

Initialization

create a multiple allele selector

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

Please refer to `mapSelector` 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 now.
- `wildtype` at all loci are the same.

Member Functions

x.clone() deep copy of a maSelector

x.indFitness(*ind, gen) calculate/return the fitness value, currently assuming diploid

maSelector accepts an array of fitness values:

- For single-locus, fitness is the fitness for genotype AA, Aa, aa, while A stands for wildtype alleles.
- For a two-locus model, fitness is the fitness for genotype

	BB	Bb	bb
AA	w_{11}	w_{12}	w_{13}
Aa	w_{21}	w_{22}	w_{23}
aa	w_{31}	w_{32}	w_{33}

  in the order of $w_{11}, w_{12,}..., w_{32}, w_{33}$.
- For a model with more than two loci, use a table of length 3^n in a order similar to the two-locus model.

3.6.5 Class mlSelector (Function form: MlSelect)

selection according to genotypes at multiple loci in a multiplicative model

Details

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

- SEL_Multiplicative: the fitness is calculated as $f = \prod_i f_i$.
- SEL_Additive: the fitness is calculated as $f = \max(0, 1 - \sum_i (1 - f_i)) = \max(0, 1 - \sum_i s_i)$. f will be set to 0 when $f < 0$. In this case, s_i are added, not f_i directly.

Initialization

create a multi-loci selector

```
mlSelector(selectors, mode=SEL_Multiplicative, subPops=[],
stage=PreMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["fitness"])
```

Please refer to mapSelector for other parameter descriptions.

selectors a list of selectors

Member Functions

x.clone() deep copy of a mlSelector

x.indFitness(*ind, gen) calculate/return the fitness value, currently assuming diploid

3.6.6 Class pySelector (Function form: PySelect)

selection using user provided function

Details

pySelector assigns fitness values by calling a user provided function. It accepts a list of susceptibility loci and a Python function. For each individual, this operator will pass the genotypes at these loci (in the order of 0-0,0-1,1-0,1-1 etc. where X-Y represents locus X - ploidy Y, in the case of diploid population), generation number,??? and expect a returned fitness value. This, at least in theory, can accommodate all selection scenarios.

Initialization

create a Python hybrid selector

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

func a Python function that accepts genotypes at susceptibility loci generation number, and return fitness value.???

loci susceptibility loci. The genotype at these loci will be passed to func.

output and other parameters please refer to help(baseOperator.__init__)???

Member Functions

x.clone() deep copy of a pySelector

x.indFitness(*ind, gen) calculate/return the fitness value, currently assuming diploid

The following example simulates the same scenario as before, with $s_1 = .2, s_2 = .3$ (so$p = .6$) and a pySelector. Note that although alleles at two loci are passed, the sel function only uses alleles from the first one.

Example 3.12: python selector

```
>>> simu = simulator(
...         population(size=1000, ploidy=2, loci=[3], infoFields=['fitness'] ),
...         randomMating())
>>>
>>> s1 = .2
>>> s2 = .3
>>> # the second parameter gen can be used for varying selection pressure
>>> def sel(arr, gen=0):
...     if arr[0] == 1 and arr[1] == 1:
...         return 1 - s1
...     elif arr[0] == 1 and arr[1] == 2:
...         return 1
...     elif arr[0] == 2 and arr[1] == 1:
...         return 1
...     else:
...         return 1 - s2
...
>>> # test func
>>> print sel([1,1])
0.8
>>>
>>> simu.evolve([
```

```
...         stat( alleleFreq=[0], genoFreq=[0]),
...         pySelector(loci=[0,1],func=sel),
...         pyEval(r"'%.4f\n' % alleleFreq[0][1]", step=25)
...         ],
...         preOps=[    initByFreq(alleleFreq=[.2,.8])],
...         end=100)
0.8310
0.9900
1.0000
1.0000
1.0000
True
>>>
```

3.7 Penetrance

3.7.1 Class `penetrance`

basic class of a penetrance operator

Details

Penetrance is the probability that one will have the disease when he has certain genotype(s). Calculation and the parameter set of penetrance are similar to those of fitness. An individual will be randomly marked as affected/unaffected according to his 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`). It will be calculated during mating, and then the affected status will be set for each offspring. Penetrance can also be used as `PreMating`, `PostMating` or even `PrePostMating`??? operator. In these cases, the affected status will be set to all individuals according to their penetrance values. It is also possible to store penetrance in a given information field specified by `infoFields` parameter (e.g. `infoFields=['penetrance']`). This is useful to check the penetrance values at a later time.

Affected status will be used for statistical purpose, and most importantly, ascertainment. They will be calculated along with fitness although they might not be used at every generation. You can use two operators: one for fitness/selection, active at every generation; one for affected status, active only at ascertainments, to avoid unnecessary calculation of the affected status.

Pentrance values are used to set the affectedness of individuals, and 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 `ancestralGen` parameter is ignored if the penetrance operator is used as a during mating operator.

Initialization

create a penetrance operator

```
penetrance(ancestralGen=-1, stage=DuringMating, begin=0, end=-1,
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

default to be always active.

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

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

stage specify the stage this operator will be applied, default to `DuringMating`.

Member Functions

x.apply(pop) set penetrance to all individuals and record penetrance if requested

x.applyDuringMating(pop, offspring, *dad=NULL, *mom=NULL) set penetrance to all individuals

x.clone() deep copy of a penetrance operator

x.penet(*) calculate/return penetrance etc.

3.7.2 Class mapPenetrance (Function form: MapPenetrance)

penetrance according to the genotype at one locus

Details

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

Initialization

create a map penetrance operator

```
mapPenetrance(loci, penet, phase=False, ancestralGen=-1,
stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

loci the loci indices. The genotypes of these loci will be examed.

locus the locus index. The genotype of this locus will be examed.???

output and other parameters please refer to help(baseOperator.__init__)???

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

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

Member Functions

x.clone() deep copy of a map penetrance operator

x.penet(*ind) currently assuming diploid???

An example of this operator is

```
mapPenetrance(locus=1, penetrance={'1-1':0, '1-2':0.5, '2-2':1})
```

Note that this dictionary can take more than three elements to accommendate more than one disease alleles.

3.7.3 Class maPenetrance (Function form: MaPenetrance)

multiple allele penetrance operator

Details

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

Initialization

create a multiple allele penetrance operator (penetrance according to diseased or wildtype alleles)

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

loci the loci indices. The genotypes of these loci will be examed.

locus the locus index. The genotype of this locus will be examed.???

output and other parameters please refer to help(baseOperator.__init__)???

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

wildtype an array of alleles in the wildtype group. Any other alleles will be considered as in the disease allele group.

Member Functions

`x.clone()` deep copy of a multi-allele penetrance operator

`x.penet(*ind)` currently assuming diploid???

An example of operator `maPenetrance` is

```
maPenetrance(loci=[1,5], wildtype=[1], penetrance=[0,0.5,1,0,0,1,0,1,1])
```

This operator behaves the same as the `mapPenetrance` example but will work if there are more than one disease alleles.

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

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

Details

`mlPentrance` is the 'multiple-loci'??? 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.

Initialization

create a multiple loci penetrance operator using a multiplicative model

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

mode can be one of `PEN_Multiplicative`, `PEN_Additive`, and `PEN_Heterogeneity`

peneOps a list of selectors???

Member Functions

`x.clone()` deep copy of a multi-loci penetrance operator

`x.penet(*ind)` currently assuming diploid???

For example, if each locus follows an additive penetrance model, we can have

```
pen = []
for loc in loci:
  pen.append( maPenetrance(locus=loc, wildtype=[1],
    penetrance=[0.0.3,0.6] ) )
# the multi-loci penetrance
penMulti = mlPenetrance(mode=PEN_Multiplicative, peneOps=pen)
```

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

assign penetrance values by calling a user provided function

Details

For each individual, users provide a function to calculate penetrance. This method is very flexible but will be slower than previous operators since a function will be called for each individual.

Initialization

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

```
pyPenetrance(loci, *func, ancestralGen=-1, stage=DuringMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

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

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

output and other parameters please refer to help(baseOperator.__init__)???

Member Functions

`x.clone()` deep copy of a Python penetrance operator

`x.penet(*ind)` currently assuming diploid???

For example, for the same multi-locus model before, we can define the following function using `pyPenetrance` as

```
def peneFunc(geno):
  p = 1
  for l in range(len(geno)/2):
    p *= (geno[l*2]+geno[l*2+1]-2)*0.3
  return p
penMulti = pyPenetrance(loci=loci, func=peneFunc)
```

As you can see, using this operator, you can define arbitrarily complex penetrance functions. Typical such penetrance functions are interaction between loci (using a multi-locus penetrance table), even random ones.

It would be useful to let `peneFunc` take parameters. This can be done by defining a Python function that returns a penetrance function. This may sound intimidating but it is really easy:

```
def peneFunc(table):
  def func(geno):
    return table[geno[0]-1][geno[1]-1]
  return func
# then, given a table, you can do
pen = pyPenetrance(loci=loci, func=peneFunc( ((0,0.5),(0.3,0.8)) ) )
```

Now, for any table, you can use `peneFunc` to return a penetrance function that uses this table.

3.8 Quantitative Trait

3.8.1 Class `quanTrait`

basic class of quantitative trait

Details

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

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

Initialization

create a quantitative trait operator, default to be always active

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

Member Functions

`x.apply(pop)` set `qtrait` to all individual

`x.clone()` deep copy of a quantitative trait operator

`x.qtrait(*)` calculate/return quantitative trait etc.

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

quantitative trait according to genotype at one locus

Details

Assign quantitative trait using a table with keys 'X-Y' where X and Y are allele numbers. If parameter `sigma` is not zero, the returned 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.

Initialization

create a map quantitative trait operator

```
mapQuanTrait(loci, qtrait, sigma=0, phase=False, ancestralGen=-1,
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["qtrait"])
```

loci an array of locus indices. The quantitative trait is determined by genotype 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(mean, sigma^2)`. For multiple loci, the form is 'a-b|c-d|e-f' etc.

sigma standard deviation of the environmental factor `N(0, sigma^2)`.

Member Functions

`x.clone()` deep copy of a map quantitative trait operator

`x.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)

Details

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

Initialization

create a multiple allele quantitative trait operator

```
maQuanTrait(loci, qtrait, wildtype, sigma=[], ancestralGen=-1,
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["qtrait"])
```

Please refer to `quanTrait` for other parameter descriptions.

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

qtrait an array of quantitative traits of AA, Aa, aa. A is the wild type 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 disease alleles. Default to `[0]`.

Member Functions

`x.clone()` deep copy of a multiple allele quantitative trait

`x.qtrait(*ind)` currently assuming diploid

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

quantitative trait according to genotypes from a multiple loci multiplicative model

Details

`mlQuanTrait` is a 'multiple-loci' 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 all 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.

Initialization

multiple loci quantitative trait using a multiplicative model

```
mlQuanTrait(qtraits, mode=QT_Multiplicative, sigma=0,
ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=["qtrait"])
```

Please refer to `quanTrait` for other parameter descriptions.

mode can be one of `QT_Multiplicative` and `QT_Additive`

qtraits a list of quantitative traits

Member Functions

`x.clone()` deep copy of a multiple loci quantitative trait operator

`x.qtrait(*ind)` currently assuming diploid

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

quantitative trait using a user provided function

Details

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

Initialization

create a Python quantitative trait operator

```
pyQuanTrait(loci, *func, ancestralGen=-1, stage=PostMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=["qtrait"])
```

Please refer to `quanTrait` for other parameter descriptions.

func a Python function that accepts genotypes at susceptibility loci and returns the quantitative trait value.

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

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

Member Functions

`x.clone()` deep copy of a Python quantitative trait operator

`x.qtrait(*ind)` currently assuming diploid

3.9 Ascertainment (subset of population)

3.9.1 Class `sample`

basic class of other sample operator

Details

Ascertainment/sampling refers to ways to select individuals from a population. In simuPOP, ascerntainment operators form separate populations in a population's 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 indices in the whole population are stored in a information field `oldindex`. That is to say, you can use `ind.info('oldindex')` to check the original position of an individual.

Two forms of sample size specification are supported: with or without subpopulation structure. For example, the `size` parameter of `randomSample` can be a number or an array (which has the length of the number of subpopulations). If a number is given, a sample will be drawn from the whole population, regardless of the population structure. If an array is given, individuals will be drawn from each subpopulation `sp` according to `size[sp]`.

An important special case of sample size specification occurs when `size=[]` (default). In this case, usually all qualified individuals will be returned.

The function forms of these operators are a little different from others. They do return a value: an array of samples.

Initialization

draw a sample

```
sample(name="sample", nameExpr="", times=1, saveAs="", saveAsExpr="",
format="auto", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Please refer to `baseOperator::__init__` for other parameters.

format format to save the samples

name name of the sample in local namespace. This variable is an array of populations of size `times`. Default to `sample`. If `name=''` is set, samples will not be saved in local namespace.

nameExpr expression version of parameter `name`. If both `name` and `nameExpr` are empty, do not store `pop`. This expression will be evaluated dynamically in population's local namespace.

saveAs filename to save the samples

saveAsExpr expression version of parameter `saveAs`. It will be evaluated dynamically in population's local namespace.

times how many times to sample from the population. This is usually `1`, but we may want to take several random samples.

Member Functions

`x.apply(pop)` apply the `sample` operator

`x.clone()` deep copy of a `sample` operator

`x.findOffspringAndSpouse(pop, ancestralDepth, maxOffspring, fatherField, motherField, spous`
find offspring and spouse

x.resetParentalIndex(pop, fatherField="father_idx", motherField="mother_idx", indexField="
reset `father_idx` and `mother_idx`

x.resetSubPopID(pop) set all subpopulation IDs to `-1` (remove)

x.samples(pop) return the samples

x.saveIndIndex(pop, indexField="oldindex") save the index of each individual to a field (usually `oldindex`)

3.9.2 function `population::shrinkByIndID()`

This function looks at the `subPopID()` field of each individual and remove anyone with a negative value.

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

shrink population

Details

This operator shrinks a population according to a given array or the `subPopID()` value of each indvidual. Subpopulations are kept intact.

Initialization

create a `pySubset` operator

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

keep an array of subpopulation IDs for each individual.

Member Functions

x.apply(pop) apply the `pySubset` operator

x.clone() deep copy of a `pySubset` operator

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

Python sampler.

Details

A Python sampler that generate a sample with given individuals.

Initialization

create a Python sampler

```
pySample(*keep, keepAncestralPops=-1, name="sample", nameExpr="",
times=1, saveAs="", saveAsExpr="", format="auto", stage=PostMating,
begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

Please refer to class `sample` for other parameter descriptions.

keep subpopulation IDs of all individuals

keepAncestralPop the number of ancestral populations that will be kept. If `-1`, keep all ancestral populations (default). If `0`, no ancestral population will be kept.

Member Functions

`x.clone()` deep copy of a Python sampler

`x.drawsample(pop)` draw a Python sample

`PySample(pop, info, name, saveAs, format)` or `Sample(pop)` if you already set subpopulation ID for each individual using `setSubPopID()` function. The operator version of these functions are `pySample(info, times, name, nameExpr, saveAs, saveAsExpr, format)`.

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

randomly draw a sample from a population

Details

This operator will randomly choose `size` individuals (or `size[i]` individuals from subpopulation `i`) and return a new population. The function form of this operator returns the samples directly. The operator keeps samples in an array `name` in the local namespace. You may access them through `dvars()` or `vars()` functions.

The original subpopulation structure/boundary is kept in the samples.

Initialization

draw a random sample, regardless of the affected status

```
randomSample(size=[], name="sample", nameExpr="", times=1, saveAs="",
saveAsExpr="", format="auto", stage=PostMating, begin=0, end=-1,
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Please refer to class `sample` for other parameter descriptions.

size size of the sample. It can be either a number which represents the overall sample size, regardless of the population structure; or an array which represents the number of samples drawn from each subpopulation.

Note

Ancestral populations will not be copied to the samples.

Member Functions

`x.clone()` deep copy of a `randomSample` operator

`RandomSample(pop, size, times, name, saveAs, format)` will randomly choose size individuals (or `sizes` from subpopulations) and return a new population. The operator version is `randomSample(size, times, name, nameExpr, saveAs, saveAsExpr, format)`.

3.9.6 Class caseControlSample (Function form: CaseControlSample)

draw a case-control sample from a population

Details

This operator will randomly choose cases affected individuals and controls unaffected individuals as a sample. The affected status is usually set by penetrance functions/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/controls) though.

A special case of this sampling scheme occurs when one of or both cases and controls are omitted (zeros). In this case, all cases and/or controls are chosen. If both parameters are omitted, the sample is effectively the same population with affected and unaffected individuals separated into two subpopulations.

Initialization

draw cases and controls as a sample

```
caseControlSample(cases=[], controls=[], spSample=False,
name="sample", nameExpr="", times=1, saveAs="", saveAsExpr="",
format="auto", stage=PostMating, begin=0, end=-1, step=1, at=[],
rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

Please refer to class sample for other parameter descriptions.

cases the number of cases, or an array of the numbers of cases from each subpopulation

controls the number of controls, or an array of the numbers of controls from each subpopulation

Member Functions

x.clone() deep copy of a caseControlSample operator

CaseControlSample(pop, cases, controls, times, name, saveAs, format) will randomly choose cases affected individuals and controls unaffected individuals. The operator version of this function is caseControlSample(case, cases, control, controls, times, name, nameExpr, saveAs, saveAsExpr, format).

The following example shows how to draw a random sample (without replacement of course) from an existing population.

Example 3.13: random sample

```
>>> # random sample
>>> # [0]: RandomSample already return
>>> #    a list of samples even if times=1 (default)
>>> Dump( RandomSample(pop, 3)[0])
Ploidy:                 2
Number of chrom:        3
Number of loci:         2 5 10
Maximum allele state:   65535
Loci positions:
                1 2
                1 2 3 4 5
                1 2 3 4 5 6 7 8 9 10
```

```
Loci names:
                loc1-1 loc1-2
                loc2-1 loc2-2 loc2-3 loc2-4 loc2-5
                loc3-1 loc3-2 loc3-3 loc3-4 loc3-5 loc3-6 loc3-7 loc3-8 loc3-9 loc3-10
population size:        3
Number of subPop:       1
Subpop sizes:           3
Number of ancestral populations:        0
individual info:
sub population 0:
   0: MU   0  0   0  0  0  0  0   0  0  0  0  0  0  0  0  0  0 |   0  0
0  0  0  0  0   0  0  0  0  0  0  0  0  0  0
   1: MU   0  0   0  0  0  0  0   0  0  0  0  0  0  0  0  0  0 |   0  0
0  0  0  0  0   0  0  0  0  0  0  0  0  0  0
   2: MU   0  0   0  0  0  0  0   0  0  0  0  0  0  0  0  0  0 |   0  0
0  0  0  0  0   0  0  0  0  0  0  0  0  0  0
End of individual info.


No ancenstral population recorded.
>>>
```

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

draw an affected sibling pair sample

Details

Special preparation for the population is needed in order to use this operator. Obviously, to obtain affected sibling pairs, we need to know the parents and the affectedness status of each individual. Furthermore, to get parental genotype, 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 *realrandom* 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.

All these requirements come at a cost: multiple ancestral populations, determining affectedness status and tagging will slow down evolution; multiple offspring will reduce effective population size. Fortunately, simuPOP is flexible enough to let all these happen only at the last several generations.

Initialization

draw an affected sibling pair sample

```
affectedSibpairSample(size=[], chooseUnaffected=False,
countOnly=False, name="sample", nameExpr="", times=1, saveAs="",
saveAsExpr="", format="auto", stage=PostMating, begin=0, end=-1,
step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=["father_idx",
"mother_idx"])
```

Please refer to class `sample` for other parameter descriptions.

chooseUnaffected instead of affected sibpairs, choose unaffected families.

countOnly set variables about 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 population structure. Otherwise, given number of sibpairs are sampled from subpopulations. If size is unspecified, this operator will return all affected sibpairs.

Member Functions

`x.clone()` deep copy of a `affectedSibpairSample` operator

`x.drawsample(pop)` draw a sample

`x.prepareSample(pop)` preparation before drawing a sample

For example, you can do

```
endGen = 1000
# having two offsprings only at the last three generations
def numOffsprings(gen):
  if gen >= endGen - 3:
    return 2
  else
    return 1
# evolve ...
simu = simulator(pop, randomMating(numOffspringsFunc = numOffsprings))
simu.evolve( ...
  parentsTagger(begin = endGen - 3),
  mapPenetrance(...., begin = endGen - 2),
  setAncestralDepth(1, at = endGen -2 )
...)
```

to let your population evolve *normally* and start to store ancestral generations and allow multiple offspring at the last several generations.

Briefly, you should

- set the ancestral depth to at least `1` to allow analyzing of the parental generation.
- use `parentsTagger` to track parents for each individual, with the usual limit of no post-mating migration.
- allow multiple offspring at least at the last generation. (You do not have to use fixed number of offspring. Other mating mode like `MATE_GeometricDistribution` can also be used.)
- use a penetrance operator to set affected status of each individual

and finally use this operator (or function)

```
affectedSibpairSample(size,  times, name, saveAs, format)
```

to get samples accessible from `dvars().name[i]`. Each sample will

- have `2*size` of paired individuals. (e.g. `individual(2n)` and `individual(2n+1)`, `n=0,1,...`, `size-1` are siblings.)
- have an ancestral generation of the same size, with parents to the sibling pairs.
- if `size` is an array, get `size[sp]` sibling pairs from subpopulation `sp` .

Other than samples `name`, variable `numSibpairs` will be set to indicate the total number of affected sibling pairs in the population. Subpopulation structure will be kept in the samples so you will know how many individuals are drawn from each subpopulation. (This information is also saved in variable `numSibpairs` of each sample.)

3.9.8 Class `largePedigreeSample`

draw a large pedigree sample

Initialization

draw a large pedigree sample

```
largePedigreeSample(size=[], minTotalSize=0, maxOffspring=5,
minPedSize=5, minAffected=0, countOnly=False, name="sample",
nameExpr="", times=1, saveAs="", saveAsExpr="", format="auto",
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["father_idx", "mother_idx"])
```

Please refer to class `sample` for other parameter descriptions.

countOnly set variables about number of affected sibpairs, do not actually draw the sample.

maxOffspring the maximum number of offspring a parent may have

minAffected minimal number of affected individuals in each pedigree, default to `0`

minPedSize minimal pedigree size, default to `5`

minTotalSize the minimum number of individuals in the sample

Member Functions

`x.clone()` deep copy of a `largePedigreeSample` operator

`x.drawsample(pop)` draw a a large pedigree sample

`x.prepareSample(pop)` preparation before drawing a sample

3.9.9 Class `nuclearFamilySample`

draw a nuclear family sample

Initialization

draw a nuclear family sample

```
nuclearFamilySample(size=[], minTotalSize=0, maxOffspring=5,
minPedSize=5, minAffected=0, countOnly=False, name="sample",
nameExpr="", times=1, saveAs="", saveAsExpr="", format="auto",
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["father_idx", "mother_idx"])
```

Please refer to class `sample` for parameter descriptions.

Member Functions

`x.clone()` deep copy of a `nuclearFamilySample` operator

`x.drawsample(pop)` draw a nuclear family sample

`x.prepareSample(pop)` preparation before drawing a sample

3.10 Statistics Calculation

3.10.1 Class `stator`

basic class of all the statistics

Details

Operator `stator` calculate various basic statistics for the population and set variables in the local namespace. Other operators/functions can refer to the results from the namespace after `stat` is applied. `Stat` is the function form of the operator. ????

Initialization

create a stator

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

Member Functions

`x.clone()` deep copy of a stator

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

calculate statistics

Details

Operator `stat` calculatse various basic statistics for the population and sets variables in the local namespace. Other operators/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.

Initialization

create an `stat` operator

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

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`. The names are potentially very confusing but I have no better idea.

- `ld['loc1-loc2']['allele1-allele2']`, `subPop[sp]['ld']['loc1-loc2']['allele1-allele2']`

- `ld_prime['loc1-loc2']['allele1-allele2']`, `subPop[sp]['ld_prime']['loc1-loc2']['allele1-a`
- `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']`

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:

- 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. Population refers to subpopulations in simuPOP term.???

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`,

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

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

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

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

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

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

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, ...]` where `loc1` etc. are loci where genotype frequencies will be calculated). All the genotypes in the population will be counted. You may use `hasPhase` to set 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 indices `a`, `b` of `genoFreq[a][b]` are dictionary keys, so you will get a *KeyError* when you used a wrong key. Usually, `genoNum.setDefault(a,{})` is preferred.

- `genoNum[a][geno]` and `subPop[sp]['genoNum'][a][geno]`, the number of genotype `geno` at allele `a`. `geno` has the form `x-y`.
- `genoFreq[a][geno]` and `subPop[sp]['genoFreq'][a][geno]`, the frequency of genotype `geno` at allele `a`.

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 to calaulate observed heterozygosities and expected heterozygosities (`heteroFreq=[loc1, loc2, ...]`). This parameter will set the following variables (arrays of observed heterozygosities). Note that `heteroNum[loc][1]` is the number of heterozygote **1x**, $x \neq 1$. Numbers and frequencies (proportions) of heterozygotes are calculated for each allele. `HeteroNum[loc]` and `HeterFreq[loc]` are the overall heterozygosity number and frequency. I.e., the number/frequency of genotype **xy**, $x \neq y$. From this number, we can easily derive the number of homozygosity.

- `HeteroNum[loc]`, `subPop[sp]['HeteroNum'][loc]`, the overall heterozygote number
- `HeteroFreq[loc]`, `subPop[sp]['HeteroFreq'][loc]`, the overall heterozygote frequency
- `heteroNum[loc][allele]`, `subPop[sp]['heteroNum'][loc][allele]`
- `heteroFreq[loc][allele]`, `subPop[sp]['heteroFreq'][loc][allele]`

homoFreq an array of loci to calaulate 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/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 indices). 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 *allloci*. Unrequested loci will have `0` distinct alleles.):

- `numOfAlleles`, `subPop[sp]['numOfAlleles']`, 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/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 sizes. This parameter will set the following variables:

- `numSubPop` the number of subpopulations

- `subPopSize` an array of subpopulation sizes. Not available for subpopulations.
- `popSize`, `subPop[sp]['popSize']` population/subpopulation size.

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

relLoci loci on which relatedness values are calculated

relMethod method used to calculate relatedness. Can be either `REL_Queller` or `REL_Lynch`. The relatedness values between two individuals, or two groups of individuals are calculated according to Queller & Goodnight (1989) (`method=REL_Queller`) and Lynch et al. (1999) (`method=REL_Lynch`). The results are pairwise relatedness values, in the form of a matrix. Original group or subpopulation numbers are discarded. `relatedness[grp1][grp2]` is the relatedness value between `grp1` and `grp2`. There is no subpopulation level relatedness values.

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`.

Member Functions

`x.apply(pop)` apply the `stat` operator

`x.clone()` deep copy of a `stat` operator

Linkage disequilibrium

- parameter:

  ```
  LD: LD = [ [1,2], [ 0,1,1,2],[1,2,1,2]]
  ```

  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,

$$\begin{aligned} D &= P_{AB} - P_A P_B \\ D' &= D/D_{max} \\ D_{max} &= \begin{cases} \min\left(P_A\left(1-P_B\right), \left(1-P_A\right)P_B\right) & \text{if } D > 0 \\ \min\left(P_A P_B, \left(1-P_A\right)\left(1-P_B\right)\right) & \text{if } D < 0 \end{cases} \\ r^2 &= \frac{D^2}{P_A\left(1-P_A\right)P_B\left(1-P_B\right)} \end{aligned}$$

  If A and B are not specified, D, D' and r^2 will be the averaged value: (basically $\sum\sum P_A P_B \,||\,$)

$$\begin{aligned} D &= \sum_i \sum_j P_i P_j \left|D_{ij}\right| \\ D' &= \sum_i \sum_j P_i P_j \left|D'_{ij}\right| \\ r^2 &= \sum_i \sum_j P_i P_j r_{ij}^2 = \sum_i \sum_j \frac{D_{ij}^2}{\left(1-P_i\right)\left(1-P_j\right)} \end{aligned}$$

  where p_i and q_j are the population allele frequencies of the ith allele on loc1 and the jth allele on loc2. Please note that some other authors uses

$$r^2 = \sum_i \sum_j \frac{D_{ij}^2}{P_i P_j}$$

  If you are sure the later is correct, please send me an email (with reference).

3.11 Expression and Statements

3.11.1 Operator (C++) `output`

This operator output a simple string. For example,

```
output(r'\n', rep=REP_LAST)
```

output a newline at the last replicate.

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

evaluate an expression

Details

Python expressions/statements will be executed when `pyEval` is applied to a population by using parameters `expr/stmts`. Statements can also 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.

Initialization

evaluate expressions/statments in the local namespace of a replicate

```
pyEval(expr="", stmts="", preStmts="", postStmts="", exposePop=False,
name="", output=">", outputExpr="", stage=PostMating, begin=0,
end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=[])
```

exposePop if true, expose current population as variable `pop`

expr the expression to be evaluated. Its result will be sent to `output`.

name used to let pure Python operator to identify themselves

output default to `>`. I.e., output to standard output.

postStmts the statement that will be executed when the operator is destroyed

preStmts the statement that will be executed when the operator is constructed

stmts the statement that will be executed before the expression

Member Functions

`x.apply(pop)` apply the `pyEval` operator

`x.clone()` deep copy of a `pyEval` operator

`x.name()` return the name of an expression

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

3.11.3 Class pyExec (Function form: PyExec)

execute a Python statement

Details

This operator takes a list of statements and execute them. No value will be returned or outputted.

Initialization

evaluate statments in the local replicate namespace, no return value

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

default to >. I.e., output to standard output.

exposePop if true, expose current population as variable `pop`

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 statements (a single or multi-line string) that will be executed when this operator is applied.

Member Functions

`x.clone()` deep copy of a `pyExec` operator

3.11.4 Function (Python) `ListVars (defined in simuUtil.py)`

`ListVars(variable)`

This function lists any variable in an indented text format. You can use `listVar(simuVars)` to have a look at all replicates or `listVar(simuVars[0]['subPop'][0])` to see variables for the first subpopulation in replicate one.

3.12 Tagging (used for pedigree tracking)

3.12.1 Class `tagger`

basic class of tagging individuals

Details

`tagger` is a during mating operator that tag individuals with various information. Potential usages are:

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

Initialization

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

```
tagger(begin=0, end=-1, step=1, at=[], rep=REP_ALL, grp=GRP_ALL,
infoFields=[])
```

Member Functions

`x.clone()` deep copy of a tagger

3.12.2 Class `inheritTagger`

inherite tag from parents.

Details

This during-mating operator will copy the tag information from his/her parents. Depending on `mode` parameter, this tagger will obtain tag from his/her father (two tag fields), mother (two tag fields) or both (first tag field from both father and mother).An example may be tagging one or a few parents and see, at the last generation, how many offspring they have.

Initialization

create an `inheritTagger`, default to be always active

```
inheritTagger(mode=TAG_Paternal, begin=0, end=-1, step=1,
at=[], rep=REP_ALL, grp=GRP_ALL, infoFields=["paternal_tag",
"maternal_tag"])
```

mode can be one of `TAG_Paternal`, `TAG_Maternal`, and `TAG_Both`

Member Functions

`x.applyDuringMating(pop, offspring, *dad=NULL, *mom=NULL)` apply the `inheritTagger`

`x.clone()` deep copy of a `inheritTagger`

3.12.3 Class `parentsTagger`

tagging according to parents' indices

Details

This during-mating operator set
c tag(), currently a pair of numbers, of each individual with indices of his/her parents in the parental population. This information will be used by pedigree-related operators like `affectedSibpairSample` to track the pedigree information. Since parental population will be discarded or stored after mating, and tagging information will be passed with individuals, mating/population change etc. will not interfere with this simple tagging system.

Initialization

create a `parentsTagger`, default to be always active

```
parentsTagger(begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=["father_idx", "mother_idx"])
```

Member Functions

x.applyDuringMating(pop, offspring, *dad=NULL, *mom=NULL) apply the `parentsTagger`

x.clone() deep copy of a `parentsTagger`

Details

This tagger takes some information fields from both parents, pass to a Python function and set the individual field with the returned value.

This operator can be used to trace the inheritance of trait values.

Initialization

simuPOP::pyTagger::pyTagger

```
pyTagger(*func=NULL, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, 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 returned value is assigned to fields `'A'` and `'B'` of the offspring. The returned 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.

Member Functions

x.applyDuringMating(pop, offspring, *dad=NULL, *mom=NULL) apply the `pyTagger`

x.clone() deep copy of a `pyTagger`

3.13 Data collector

Sometimes, instead of output data directly, we may want to collect history data on some expression. Data collector is designed for this purpose.

3.13.1 operator (Python) `collector`, in `simuUtil.py`

This operator accepts the following parameters:

- `name:` name by which the collected data will be displayed. Variable names will be a list of stored values. (`generation` is not stored. You can always put it in `expr` though.)
- `expr:` an expression that will be evaluated. The result will be converted to a list (if needed) and stored in `name[gen]`.

When this operator is called, it will evaluate `expr` and store its result in `name[gen]`. After evolution, you will get a dictionary of values indexed by generation numbers.

3.14 Output

3.14.1 operator (C++) `savePopulation`

3.14.2 function (Python) `SaveFstat` (in `simuUtil.py`)

3.14.3 operator (Python) `saveFstat` (in `simuUtil.py`)

3.14.4 function (Python) `loadFstat` (in `simuUtil.py`)

3.15 Visualization

There is no special visualizer (there was indeed a matlabPlotter before ver 0.5.9 but I decide to remove it since matlab is not universally available.) Since everything is exposed dynamically, all you need to do is plotting variables in whatever way you prefer. The basic steps are:

- find an appropriate tool. I prefer R/Rpy to any other tools since I am familiar with R. You can make your own choice.
- write a function to plot variable. If you would like to plot history of a variable, you can use the `Aggregator` object defined in `simuUtil.py`.
- wrap this function as an operator.

`simuRPy.py` provides a pure Python operator `varPlotter`. It is defined in `simuSciPy` and `simuMatPlt.py` as well but they are lack of subplot capacity (so the usages are different) due to the limit of SciPy/gplt and MatPlotLib's plotting capacities. Also note as of Apr, 2006, the development of gplt in scipy was stopped so support of simuPOP/simuSciPy was stopped as well.

3.15.1 Operator (Python) `varPlotter (simuRPy.py)`

The use of `varPlotter` is easy, if you would like to

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.

Plotting without history

- use option `history=False`. Parameters `byVal, varDim` etc. will be ignored.

Other options are

1. `title, xtitle, ytitle:` title of your figure(s). `title` is default to your expression, `xtitle` is defaulted to `generation`.
2. `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.
3. `update:` update figure after `update` generations. This is used when you do not want to update the figure at every generation.
4. `saveAs:` save figures in files `saveAs#gen.eps`. For example, if `saveAs='demo'`, you will get files `demo1.eps`, `demo2.eps` etc.
5. `separate:` plot data lines in separate panels.
6. `image:` use R image function to plot image, instead of lines.
7. `level:` level of image colors (default to 20).
8. `leaveOpen:` whether or not leave the plot open when plotting is done. Default to `True`.

Here is an example:

3.16 Terminator

3.16.1 Class `terminator`

terminate the evolution

Details

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

Initialization

create a terminator, default to be always active

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

Member Functions

`x.clone()` deep copy of a terminator

`x.message()` return the message to print when terminated???

3.16.2 Class `terminateIf`

terminate according to a condition

Details

This operator terminates the evolution under certain conditions. For example, `terminateIf(condition='alleleFreq[0][1]<0.05', begin=100)` terminates the evolution if the allele frequency of allele `1` at locus `0` is less than 0.05. Of course, to make this opertor work, you will need to use a `stat` operator before it so that variable `alleleFreq` exists in the local namespace.

When the condition is true, a shared variable `var="terminate"` will be set to the current generation.

Initialization

create a `terminateIf` terminator

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

Member Functions

`x.apply(pop)` apply the `terminateIf` terminator

`x.clone()` deep copy of a `terminateIf` terminator

3.16.3 Class `continueIf`

terminate according to a condition failure

Details

The same as `terminateIf` but continue if the condition is `True`.

Initialization

create a `continueIf` terminator

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

Member Functions

`x.apply(pop)` apply the `continueIf` terminator???

`x.clone()` deep copy of a `continueIf` terminator

3.17 Conditional operator

3.17.1 Class `ifElse`

conditional operator

Details

This operator accepts

- an expression that will be evaluated when this operator is applied;
- an operator that will be applied if the expression is `True` (default to null);
- an operator that will be applied if the expression is `False` (default to null).

When this operator is applied to a population, it will evaluate the expression and depending on its value, apply the supplied operator. Note that the `begin`, `at`, `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.

Initialization

simuPOP::ifElse::ifElse

```
ifElse(cond, *ifOp=NULL, *elseOp=NULL, output=">", outputExpr="",
stage=PostMating, begin=0, end=-1, step=1, at=[], rep=REP_ALL,
grp=GRP_ALL, infoFields=[])
```

cond expression that will be treated as a bool variable

elseOp an operator that will be applied when `cond` is `False`

ifOp an operator that will be applied when `cond` is `True`

Member Functions

`x.apply(pop)` apply the `ifElse` operator to one population

`x.clone()` deep copy of an `ifElse` operator

The following example uses some advanced operators of simuPOP:

- set affected status using `maPenetrance` as a `DuringMating` operator (penetrance can be used at other stages).
- count the number of affected individuals. Note that this has to be done after the penetrance operator is applied.
- if no one is affected, inject some mutations into the population. Note the use of `ifElse` operator.
- expose individual affectedness to the local namespaces. Note the use of `exposePop` option. With this, you can call any population member function.
- plot affectedness, use image.
- use dryrun to exam simulator first.

Example 3.14: Conditional operator

```
>>> from simuRPy import *
>>> from simuUtil import *
>>> numRep=4
>>> popSize=100
>>> endGen=50
>>>
>>> simu = simulator(population(size=popSize, loci=[1]),
...     randomMating(), rep=numRep)
>>> simu.evolve(
...     preOps = [ initByValue([1,1])],
...     ops = [
...         # penetrance, additve penetrance
...         maPenetrance(locus=0, wildtype=[1], penetrance=[0,0.5,1]),
...         # count number of affected
...         stat(numOfAffected=True),
...         # introduce disease if no one is affected
...         ifElse(cond='numOfAffected==0',
...             ifOp=kamMutator(rate=0.01, maxAllele=2)),
...         # expose affected status
...         pyExec('pop.exposeAffectedness()', exposePop=True),
...         # plot affected status
...         varPlotter(expr='affected',plotType="image", byRep=1, update=endGen,
...             varDim=popSize, win=endGen, numRep=numRep,
...             title='affected status', saveAs="ifElse")
...     ],
...     end=endGen,
...     dryrun=False
... )
Traceback (most recent call last):
  File "<embed>", line 1, in ?
AttributeError: 'population' object has no attribute 'exposeAffectedness'
PostMating operator <simuPOP::pyExec > throws an exception.

Traceback (most recent call last):
  File "refManual.py", line 19, in ?
    try:
SystemError: Evalulation of statements failed
>>>
```

3.18 Miscellaneous

3.18.1 Class noneOp

none operator

Initialization

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

This operator does nothing.

Member Functions

`x.apply(pop)` apply the `noneOp` operator to one population

`x.clone()` deep copy of a `noneOp` operator

This operator is used like follows:

```
if savePop :
  saveOp = savePopulation(output='a.txt')
else:
  saveOp = noneOp()
simu.evolve( [ ... saveOp ])
```

3.18.2 Class `pause`

pause a simulator

Details

This operator pauses the evolution of a simulator at given generations or at a key stroke, using `stopOnKeyStroke=True` option. Users can use `'q'` to stop an evolution. When a simulator is stopped, press any other key to resume the simulation or escape to a Python shell to examine the status of the simulation by press `'s'`.

There are two ways to use this operator, the first one is to pause the simulation at specified generations, using the usual operator parameters such as `at`. Another way is to pause a simulation with any key stroke, using the `stopOnKeyStroke` parameter. This feature is useful for a presentation or an interactive simulation. When `'s'` is pressed, this operator expose the current population to the main Python dictionary as variable `pop` and enter an interactive Python session. The way current population is exposed can be controlled by parameter `exposePop` and `popName`. This feature is useful when you want to examine the properties of a population during evolution.

Initialization

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

```
pause(prompt=True, stopOnKeyStroke=False, exposePop=True,
popName="pop", output=">", outputExpr="", stage=PostMating, begin=0,
end=-1, step=1, at=[], rep=REP_LAST, grp=GRP_ALL, infoFields=[])
```

exposePop whether or not expose `pop` to user namespace, only useful when user choose `'s'` at pause. Default to `True`.

popName by which name the population is exposed. Default to `pop`.

prompt if `True` (default), print prompt message

stopOnKeyStroke if `True`, stop only when a key was pressed

Member Functions

`x.apply(pop)` apply the `pause` operator to one population

`x.clone()` deep copy of a `pause` operator

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

timer operator

Details

This operator, when called, output the difference between current and the last called clock time. This can be used to estimate execution time of each generation. Similar information can also be obtained from `turnOnDebug(DBG_PROFILE)`, but this operator has the advantage of measuring the duration between several generations by setting `step` parameter.

Initialization

create a timer

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

Member Functions

`x.apply(pop)` apply the `ticToc` operator to one population

`x.clone()` deep copy of a `ticToc` operator

3.18.4 Class `setAncestralDepth`

set ancestral depth

Details

This operator set the number of ancestral generations to keep in a population. It is usually called like `setAncestral(at=[-2])` to start recording ancestral generations to a population at the end of the evolution. This is useful when constructing pedigree trees from a population.

Initialization

create a `setAncestralDepth` operator

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

Member Functions

`x.apply(pop)` apply the `setAncestralDepth` operator to one population

`x.clone()` deep copy of a `setAncestralDepth` operator

3.19 Debug-related operators/functions

3.19.1 Class `turnOnDebug` (Function form: `TurnOnDebug`)

set debug on

Details

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

- set environment variable `SIMUDEBUG`
- use `simuOpt.setOptions(debug)` function, or
- use `TurnOnDebug` or `TurnOnDebugByName` function
- use this `turnOnDebug` operator

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

Initialization

simuPOP::turnOnDebug::turnOnDebug

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

Member Functions

`x.apply(pop)` apply the `turnOnDebug` operator to one population

`x.clone()` deep copy of a `turnOnDebug` operator

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

set debug off

Details

Turn off debug.

Initialization

simuPOP::turnOffDebug::turnOffDebug

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

Member Functions

`x.apply(pop)` apply the `turnOffDebug` operator to one population

`x.clone()` deep copy of a `turnOffDebug` operator

Standard `simuPOP` library 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:

```
>>> 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.

CHAPTER

FOUR

Global and Python Utility functions

4.1 Option Handling

4.1.1 Conventions of simuPOP scripts

A simuPOP script is usually composed of the following parts:

1. First line:

   ```
   #!/usr/bin/env Python
   ```

2. Introduction to the whole script:

   ```
   '''
   This script simulates ....
   '''
   ```

   These comments can be accessed as module `__doc__` and will be displayed as help message.

3. Options: (see the next section)

   ```
   options = [
   ... a dictionary of all user input parameters ...
   ]
   ```

   These parameters will be handled by simuPOP automatically. Users will be able to set them through command line, configuration file, Tkinter- or wxPython-based GUI.

4. Auxiliary functions

5. Evolution function

   ```
   def simulation(....)
   ```

6. Executable part:

   ```
   if __name__ == '__main__':
     allParam = simuOpt.getParam(options,
       ''' A short description ''', __doc__)
     # if user press cancel,
     if len(allParam) == 0:
       sys.exit(1)
     # -h or --help
     if allParam[0]:
   ```

```
    print simuOpt.usage(options, __doc__)
    sys.exit(0)
# sace configuration, something like
if allParam[-2] != None:
   simuOpt.saveConfig(options, allParam[-2]+'.cfg', allParam)
# get the parameters, something like
N = allParam[1]
# run the simulation
simulation(N)
```

You will notice that `simuOpt` does all the housekeeping things for you, including parameter reading, conversion, validation, printing usage, saving the configuration file. Since most of the parts are pretty standard, you can actually copy any of the scripts under the `scripts` directory as a template for your new script.

Note that these scripts, if proper written, can also be imported. Other scripts (or interactive session) can import a script and call its simulation function directly.

4.1.2 Parameter handling and user input

Although simuPOP scripts, simply Python scripts, can be in any valid Python style, it is highly recommended that all simuPOP scripts follow the same writing style and provide a uniform interface to users. From a user's point of view, a simuPOP script `cmd.py` should

1. Start a Tk/wxPython dialog to accept a user's input when `--noDialog` is not specified.
2. List all command-line/config file options through `-h` or `--help` option.
3. Accept `-c` or `--config` parameter to read a configuration file and set parameters.
4. Be able to use command line arguments to set parameters if `--useDefault` is not specified.
5. When `--noDialog` and `--useDefault` is specified, use default values for all parameters, if they can not be obtained from command-line parameters, configuration file, and have default values.
6. Accept `--saveconfig file` to save current configuration into `file`.
7. Be able to make use of optimized libraries through the use of command line parameter (`--optimized`), config file entry (`optimized=True`) or environment variable (`SIMUOPTIMIZED`).
8. Be able to make use of long-allele libraries through the use of command line parameter (`--longallele`), config file entry (`longallele=True`) or environment variable (`SIMULONGALLELE`).

To alleviate the trouble of doing all these, `simuPOP` has provided a set of functions. Here is how parameters should be handled. The first step is describing each parameter in details. This includes (but not all of these are necessary) short and long argument names, entries in a configuration file, prompting when asking for user's input, default values, the description that will be shown in usage, allowed types of parameters, the function to validate the inputted values. All these should be put in a list of dictionaries like follows:

```
options = [
  { 'arg':'h', 'longarg':'help', 'default':False,
    'allowedTypes':[IntType],
    'description':'print this message',
    'jump':-1 },
  { 'longarg':'saveconfig=', 'default':'', 'allowedTypes':[StringType],
```

```
    'description':'Save current configuration in a file.'},
  { 'arg':'m', 'longarg':'mu', 'label'='mutationRate',
    'default':0.005,
    'validate': simuOpt.valueBetween(0,1),
    'description':'mutation rate (a number or an array of numbers) at each loci'
  } ]
```

The entries:

- `arg` and `longarg` are in command line argument format. For example,
  - `arg:'h'` checks the presence of argument `-h`, returns `True` if succeeds
  - `arg:'f:'` checks the presence of argument pair `-f something`, returns `something` if succeeds
  - `longarg:'help'` checks the presence of argument `--longarg`, returns `True` if succeeds
  - `longarg:'mu='` checks the presence of argument pair `--mu number`, returns `number` if succeeds.
- `label` will be used as the label of the input field in a parameter dialog, and as the prompt for user input.
- `default` is used when prompt is empty, or when user press 'Enter' directly.
- `useDefault` use default value without asking, if the value can not be determined from GUI, command line option or config file. This is useful for options that rarely need to be changed. Setting them to `useDfault` allows short command lines, and easy user input.
- `description` is the description of this parameter, will be put into the usage information. (`-h` or help button in parameter dialog).
- `allowedTypes` is the accepted types. If `allowedTypes` is `types.ListType` or `types.TupleType` and the user's input is a scalar, the input will be converted to a list automatically.
- `validate` is a function to validate the parameter. You can define your own functions or use the following from `simuOpt`:
  - `valueGT(a), valueLT(a), valueGE(a), valueLE(a)`: check if the value is greater than, less than, greater than or equal to, less than or equal to a value `a`.
  - `valueBetween(a,b), valueOneOf(list)`: check if the value is between `a` and `b` or is one from `list`.
  - `valueValidFile(), valueValidDir()`: check if the parameter is a valid file/directory name.
  - `valueIsNum()`: check if the parameter is a number.
  - `valueListOf()`: check if the parameter is a list of a given type, in a list of types, or just pass a validator. For example, you can use `valueListOf(types.IntType)`, `valueListOf([types.IntType, types.LongType])` or `valueListOf( valueValidFile())`. As you can see, validators can be nested.
  - `valueOr(validator), valueAnd(val1, val2), valueOr(val1,val2)`: accept other validators and perform respective logical calculations. For example

    ```
    valueOr( valueGT(0), valueListOf( valueGT(0) ))
    ```

    accepts a positive number, or a list of positive numbers.
- `chooseOneOf`: if specified, `simuOpt` will choose one from a list of values using a listbox (Tk) or a combo box (wxPython) .
- `chooseFrom`: if specified, `simuOpt` will choose one or more items from a list of values using a listbox (tk) or a combo box (wxPython) .

- `separator`: if specified, a blue label will be used to separate groups of parameters.
- `jump`: it is used to skip some parameters when doing the interactive user input. For example, `getParam` will skip the rest of the parameters if `-h` is specified since parameter `-h` has item `'jump':-1` which means that jump 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.
- `jumpIfFalse:` The same as `jump` but jump if current parameter is `False`.

With all these information at hand, the rest is routine, if you follow the coding conventions.

4.2 Gene Mapping

4.3 Save / Write in other formats

4.4 Random Number Generator

Random number generator is a tricky business. Reliable and fast RNGs are hard to find and everyone seems to trust/distrust certain RNGs. To avoid such arguments, I have included all RNGs from GNU Scientific Library and you can choose any of the 61 RNGs, if you really know what the differences between them. (I do not, except that some of them are really bad but fast.) Note that RNG that can not generate a full range of integers are removed.

Example 4.1: Random number generator

```
>>> print ListAllRNG()
('gfsr4', 'mt19937', 'mt19937_1999', 'mt19937_1998', 'r250', 'rand', 'rand48', 'random12
>>> print rng().name()
mt19937
>>> SetRNG("taus2", seed=10)
>>> print rng().name()
taus2
>>>
```

If you need to use a random number generator in your `pyEval` operator, you can either use Python random module (`import random`) or use `rng()` function to get the random number generator of simuPOP. Note that `rng()` does not have many member functions and it might be tricky to use them correctly. (This object is not designed to be used at Python level. For a full list of member functions, check `src/utility.h`)

Example 4.2: Random number generator

```
>>> r=rng()
>>> #help(RNG)
>>> for n in range(1,10):
...     print r.randBinomial(10, .7),
... #end
...
8 7 7 8 7 6 7 7 8
>>>
>>>
>>>
```

Since simuPOP 0.7.1, RNGs are seeded in the following order:

- use random number from `/dev/urandom` if it is available
- use random number from `/dev/random` if it is available
- use Python expression `(random.randint(0, sys.maxint) + int(time.time())) % sys.maxint`. This method is used only if simuPOP is first loaded and you are going to set random number generator by yourself. The relevant codes in simuPOP.py are recommended in this case.

The seed can also be retrieved using `rng().seed()` function, which should be saved for serious simulations.

CHAPTER

FIVE

Extending simuPOP

simuPOP can be extended easily using Python programming language. Because almost all data are exposed to the Python interface, your ability of extending simuPOP is *unlimited*. However, because Python is slower than C++ and the exchange of data between internal C++ data structure and Python interface may be costly, it is not recommended to write frequently used operators in Python. Appropriate pure Python operators are visualizers, statistics calculators, file outputers etc.

To write simuPOP extension, you will have to know more about data structures and member functions of population. Note that for efficiency and implementation reasons, many of the following functions do not provide keyword parameters.

5.1 Genotypic structure

The genotypes of an individual are organized as a single array. For example, if you have an diploid individual with two chromosomes, having 2 and 3 loci respectively. The genotypes should be in the order of

```
0-0-0, 1-0-0, 0-1-0, 1-1-0, 2-1-0, 0-0-1, 1-0-1, 0-1-1, 1-1-1, 2-1-1,
```

where X-X-X are locus-chromosome-ploidy indices. An important consequence of this arrangement is that 'locus location' + 'the total number of loci' is the location of the locus on the other set of chromosomes.

Several functions are provided to retrieve genotypic information:

- `ploidy()`, ploidy
- `numChrom()`, the number of chromosomes
- `numLoci(chrom)`, the number of loci on chromosome `chrom`
- `totNumLoci()`, the total number of loci
- `genoSize()`, the size of genotype. Equals to `totNumLoci()*ploidy()`.
- `alleleName()`, allele name given by parameter `alleleNames`. Otherwise the allele number is returned.
- `locusPos(loc)`, the locus position on chromosome (Distance to the beginning of chromosome)
- `arrlociPos()`, returns an `carray` of the locus distances.

The last function is very interesting. It actually returns the reference of the internal locus distance array. If you change the values of the returned array, the internal locus distance will be changed! All functions with this property will be named `arrFunc()`.

The following example shows how to change the locus distance through this function.

Example 5.1: geno stru

```
>>> pop = population(1, loci=[2,3,4])
>>> print pop.numLoci(1)
3
>>> print pop.locusPos(2)
1.0
>>> dis = pop.arrLociPos()
>>> print dis
[1.0, 2.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0, 4.0]
>>> dis[2] = 0.5
>>> print pop.locusPos(2)
0.5
>>> print pop.arrLociPos()
[1.0, 2.0, 0.5, 2.0, 3.0, 1.0, 2.0, 3.0, 4.0]
>>>
```

5.2 Accessing genotype and other info

Genotype of an individual can be retrieved through the following functions:

- `ind.allele(index, p=0)`,
- `ind.setAllele(value, index, p=0)`,
- `ind.arrGenotype(p=0, ch=0)`,

where `p` means ploidy. I.e., the index of the copy of chromosomes. `ch` means chromosome. For example

```
pop.individual(1).arrGenotype(1, 2)
```

returns an array of alleles on the third chromosome of the second copy of chromosomes, of the second individual in the population `pop`.

Example 5.2: genotype

```
>>> InitByFreq(pop, [.2,.8])
>>> Dump(pop, alleleOnly=1)
individual info:
sub population 0:
   0: FU   1  0   1  1  0   1  1  1  1 |   0  1   1  1  1   1  1  1  0
End of individual info.


No ancenstral population recorded.
>>> ind = pop.individual(0)
>>> print ind.allele(1,1)
1
>>> ind.setAllele(3,1,1)
>>> Dump(pop, alleleOnly=1)
individual info:
sub population 0:
   0: FU   1  0   1  1  0   1  1  1  1 |   0  3   1  1  1   1  1  1  0
End of individual info.
```

```
No ancenstral population recorded.
>>> a = ind.arrGenotype()
>>> print a
[1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 3, 1, 1, 1, 1, 1, 1, 0]
>>> a = ind.arrGenotype(1)
>>> print a
[0, 3, 1, 1, 1, 1, 1, 1, 0]
>>> a = ind.arrGenotype(1,2)
>>> print a
[1, 1, 1, 0]
>>> a[2]=4
>>> # the allele on the third chromosome has been changed
>>> Dump(pop, alleleOnly=1)
individual info:
sub population 0:
   0: FU   1  0   1  1  0   1  1  1  1 |   0  3   1  1  1   1  1  4  0
End of individual info.


No ancenstral population recorded.
>>>
```

Sex, affected status can be accessed through `sex, setSex, affected, setAffected` functions.

Example 5.3: genotype

```
>>> print ind.sex()
2
>>> print ind.sexChar()
F
>>> ind.setSex(Female)
>>> ind.setAffected(True)
>>> print ind.tag()
Traceback (most recent call last):
  File "refManual.py", line 1, in ?
    #
AttributeError: 'individual' object has no attribute 'tag'
>>> ind.setTag([1,2])
Traceback (most recent call last):
  File "refManual.py", line 1, in ?
    #
AttributeError: 'individual' object has no attribute 'setTag'
>>> Dump(pop)
Ploidy:                2
Number of chrom:       3
Number of loci:        2 3 4
Maximum allele state:  65535
Loci positions:
               1 2
               0.5 2 3
               1 2 3 4
Loci names:
               loc1-1 loc1-2
               loc2-1 loc2-2 loc2-3
```

```
                loc3-1 loc3-2 loc3-3 loc3-4
population size:        1
Number of subPop:       1
Subpop sizes:           1
Number of ancestral populations:        0
individual info:
sub population 0:
   0: FA   1  0   1  1  0   1  1  1  1 |   0  3   1  1  1   1  1  4  0
End of individual info.


No ancenstral population recorded.
>>>
```

5.2.1 Direct population manipulation

FIXME

5.3 Writing pure Python operator

Now we know how to access information for individuals in a population, but how can we use them in reality? Namely, how can you write an pure Python operator?

5.3.1 Use `pyOperator`

There are two kinds of pure Python operators. The first one is easy: define a function and wrap it with a `pyOperator` operator. This method is highly recommended because of its simplicity. Many user scripts will use this kind of pure Python operator. You can find such examples in scripts directory. A good one may be simuCDCV.py where a pure Python operator is used to calculate and visualize special statistics.

For example, if you would like to record a silly statistics, namely the genotype of the m individual at locu n, you can do:

```
def sillyStat(pop, para):
  # para can be used to pass any number of parameters
  (filename, m, n) = para # unpack parameter
  f = open(filename)
  f.write('%d ' % pop.individual(m).allele(n) )
  f.close()
# then in the evole function
evolve(...
  ops=[ # other operators
    pyOperator(func=sillyStat, param=('file.txt', 2, 1) )
  ]
)
```

`pyOperator` is by default a post-mating operator, you can redefine its stage by `stage` parameter.

5.3.2 Use Python `eval` function

This kind of pure Python operators acts more like an ordinary operator. They are usually `pyEval` or `pyExec` operators returned by a wrapper function. For example, the following function defines a `tab` operator:

Example 5.4: Tab operator

```
>>> def tab(**kwargs):
...     parm = ''
...     for (k,v) in kwargs.items():
...         parm += ' , ' + str(k) + '=' + str(v)
...     cmd = r'output( """\t""" ' + parm + ')'
...     # print cmd
...     return eval(cmd)
... #end
...
```

This function actually returns an operator

```
output(r"\t", rep=REP_LAST, begin=500)
```

This kind of operators have some advantages, namely

- it acts more like ordinary operator.
- it is more efficient since it is handled (at least the first layer) by a C/C++ operator.

However, because of its complexity, such operators can only be found in system libraries. You can ignore the rest of this section if `pyOperator` is enough to you.

To define a pure Python operator, here are what you will generally do:

- write a function that acts on a population. This function should be able to be called like `func(simu.population(0))`.
- wrap this function as an operator.

For example, function `saveInFstatFormat(pop, output, outputExpr, dict)` saves a population in FSTAT format. Its definition is (first 15 lines)

Example 5.5: genotype

```
>>> print ind.sex()
2
>>> print ind.sexChar()
F
>>> ind.setSex(Female)
>>> ind.setAffected(True)
>>> print ind.tag()
Traceback (most recent call last):
  File "refManual.py", line 1, in ?
    #
AttributeError: 'individual' object has no attribute 'tag'
>>> ind.setTag([1,2])
Traceback (most recent call last):
  File "refManual.py", line 1, in ?
    #
```

```
AttributeError: 'individual' object has no attribute 'setTag'
>>> Dump(pop)
Ploidy:                 2
Number of chrom:        3
Number of loci:         2 3 4
Maximum allele state:   65535
Loci positions:
                1 2
                0.5 2 3
                1 2 3 4
Loci names:
                loc1-1 loc1-2
                loc2-1 loc2-2 loc2-3
                loc3-1 loc3-2 loc3-3 loc3-4
population size:        1
Number of subPop:       1
Subpop sizes:           1
Number of ancestral populations:        0
individual info:
sub population 0:
   0: FA   1  0   1  1  0   1  1  1  1 |   0  3   1  1  1   1  1  4  0
End of individual info.


No ancenstral population recorded.
>>>
```

Note that

- you can use this function independently like

```
saveInFstatFormat(simu.population(1),'a.txt')
```

- `pop.vars()` is used to evaluate `outputExpr`.

Then you can wrap this function by an operator, actually a function that returns a `pyEval` operator:

Example 5.6: save fstat

```
>>> def saveFstat(output='', outputExpr='', **kwargs):
...     # deal with additional arguments
...     parm = ''
...     for (k,v) in kwargs.items():
...         parm += str(k) + '=' + str(v) + ', '
...     # pyEval( exposePop=1, param?, stmts="""
...     # saveInFSTATFormat( pop, rep=rep?, output=output?, outputExpr=outputExpr?)
...     # """)
...     opt = '''pyEval(exposePop=1, %s
...         stmts=r\'\'\'saveInFstatFormat(pop, rep=rep, output=r"""%s""",
...         outputExpr=r"""%s""" )\'\'\')''' % ( parm, output, outputExpr)
...     # print opt
...     return eval(opt)
... #end
...
>>>
```

This function takes all parameters of an ordinary operator:

```
saveFstat(at=[-1], outputExpr=r'a'+str(rep)+'.txt')
```

and generates a `pyEval` operator (use above example).

```
pyEval(exposePop=1, at=[-1],
  stmts=r"""saveInFSTATFormat(pop,
    output='''''', outputExpr=r''' 'a'+str(rep)+'.txt' """
  )
```

In this example,

- `pyEval` works in the local namespace of each replicate. To access that replicate of population, you should use the magic parameter `exposePop` of `pyEval`. When set `True`, `pyEval` will automatically set a variable `pop` in the current local namespace before any statement is executed. This is why we can call `saveInFSTATFormat(pop...)`
- `"'a"'` quotes are used to avoid conflicts with quotes in `outputExpr` etc.

.

5.4 Ultimate extension: working in C++

It is sometimes desired to write simuPOP extension in C++. For example,

- when you need some other mating scheme.
- when you need certain operator that a pure Python implementation would be too slow.
- if some aspect of simuPOP is too limited (like the number of maximum alleles).

It is not difficult to write simuPOP extension in C++, once you know how simuPOP is organized. The general procedure is

- install the latest version of SWIG (>1.3.28)
- check out simuPOP source using subversion
- build from source and see if your programming environment works well
- to add an operator, make changes in appropriate `.h` file. Check `simuPOP_common.i` if your operator can not be used.

The source code is reasonably well commented with full doxygen based documentation. Please post to the simuPOP forum if you encounter any problem while writing operators in C++.

5.5 Debugging

5.5.1 Test scripts

There are many test scripts under the `test` directory. It is recommended that you run the test scripts after you installed simuPOP. 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
```

Please report any failed test.

5.5.2 Memory leak detection

Python extensions tend to have memory leak problem, caused by the refcount mechanism. If your simuPOP script uses more and more RAM without population size increase, you may have this problem. You may try to disable individual operators and find out the offending operator if the problem persist.

Potential simuPOP developers can make use of simuPOP's built-in refcount detection mechanism. To use it,

- compile Python with configure option – with - `pydebug` . This will enable `sys.totalrefcount()` etc.

- compile simuPOP with - `DPy_REF_DEBUG`. This can be done in setup.py, or better in SConstruct.

`simulator.evolve` will check reference counts at the end of each generation and report any increased reference count. Some operators may create Python objects (like ascertainment operators) but if you see repeated warnings at each generation, there is definitely a memory leak.

BIBLIOGRAPHY

INDEX