Kpop Documentation

Release 0.1.0

F1bio MacMendes

Sep 24, 2017

Contents

Warning: Beta software

You are using a software that has not reached a stable version yet. Please beware that interfaces might change, APIs might disappear and general breakage can occur before *1.0*.

If you plan to use this software for something important, please read the roadmap, and the issue tracker in Github. If you are unsure about the future of this project, please talk to the developers, or (better yet) get involved with the development of Kpop!

Kpop is a Python package concerned with population genetics analysis. It features:

- Interacts with common population genetic packages (e.g., Structure, ADMIXTURE).
- Clusterization and computation of admixture coefficients.
- Integrates machine learning methods from Sklearn and Tensorflow.
- Easy graphics.

CHAPTER 1

Table of contents

Overview

Kpop is a Python package to perform population genetics analysis that integrates traditional methods such as PCA and LDA (Latent Dirichilet Analysis, a.k.a the algorithm introduced by Pritchard in the Structure program) with machine learning.

Command line interface

Kpop can be used either as a library or as a command line interface application. Although the former is more flexible and powerful, some simple transformations and analysis can be done directly on the command line without touching any Python code.

Conversion from kpop to other common formats. Export to Structure database and param files:

```
$ kpop export pop.csv -f structure
```

Simple graphics. Show the result of a PCA analysis:

```
$ kpop show pop.csv -m pca
```

Clusterization/sharp estimate of parental affliations:

```
$ kpop cluster pop.csv -m kmeans -k 2
```

Soft cluster/computation of admixture coefficients:

```
$ kpop admix pop.csv -m kmeans -k 2
```

Basic statistics:

```
$ kpop stats pop.csv
```

Creation of random synthetic populations:

```
$ kpop create pop.csv --size 100 --num-loci 200
```

Start shell loading all of kpop symbols and all population files and in the current directory:

```
$ kpop shell
```

We encourage you to explore more options by using the builtin help utility:

```
$ kpop --help           # global help
$ kpop show --help      # help for a specific command
```

Kpop data formats

Kpop defines two file formats that represents populations. Pickle is a binary format that can be used to store the full state of a population, including any additional fields and meta data you may have created. Pickle is used internally by Python to serialize objects and is the fastest and most flexible format.

A major drawback of using Pickle is that other programs will not understand it. It can also change across Python versions, so if you create a database with a later Python version, it might not work when loaded from older Python interpreters. Pickle may also break after a major version upgrade of Kpop itself.

Pickle is fast and convenient, but it is not an archival and data exchange format. For that, Kpop uses simple CSV files. CSV is limited and is not the most efficient format both in terms of loading speed and disk usage. It is however easy to produce and you can even use your favorite spreadsheet program to create/edit an CSV file.

Kpop expects that CSV files should have a certain structure. The `kpop.load_csv()` function can adapt to different formats, but the command line interface expects a more or less rigid configuration. Your CSV file must have a single header line for which Kpop understands a few column names:

id: id for a single individual.

index: a numeric index. Kpop will ignore this column if id is given or otherwise use it as a id.

population: a id or index for the population that each individual belongs to.

gender: arbitrary gender id string. Not restricted to male/female.

age: a numeric (float) value representing age. You decide if this number means years, days, minutes, simulation ticks, etc.

phenotype: arbitrary string describing phenotype.

meta information: any column named as "#some-name" will be treated as arbitrary meta information attached to each individual. This data is stored, but does not influence any analysis performed by Kpop.

All other columns are treated as genetic marker names. The content of each marker is a string of N characters in which each character represents an specific allele. Kpop prefers numeric identifiers (e.g.: 12, 11, 22) vs letters (eg.: aA, AB, aa, etc), but it also accepts the later. By default, it treats `0`, the dash character (`-`) and empty cells as missing data.

A typical Kpop CSV file will be like the following:

```
id,population,rs123,rs1234,rs42
john,uk,12,02,11
paul,uk,22,22,
psy,korea,--,21,22
```

In example above, "john" has a missing allele in the second locus and "paul" and "psy" have no data for an entire locus (the third and the first, respectively).

Kpop python interface

In order to enjoy the full power of Kpop, it is necessary to use it from Python. Kpop can be used as a library by importing it in python code:

```
import kpop

pop = kpop.Population.random(10, 100)
...
```

If you are just exploring, it might be more useful to just open the Python shell or a Jupyter notebook using one of the commands:

```
$ kpop shell
$ kpop shell --notebook
```

It will start a Jupyter shell (or notebook) that already loads all symbols in the Kpop namespace and

Users are refered to the *API Reference*

Installation instructions

kpop can be installed using pip:

```
$ python3 -m pip install kpop
```

This command will fetch the archive and its dependencies from the internet and install them.

If you've downloaded the tarball, unpack it, and execute:

```
$ python3 setup.py install --user
```

You might prefer to install it system-wide. In this case, skip the `--user` option and execute as superuser by prepending the command with `sudo`.

Troubleshoot

Windows users may find that these command will only works if typed from Python's installation directory.

Some Linux distributions (e.g. Ubuntu) install Python without installing pip. Please install it before. If you don't have root privileges, download the get-pip.py script at https://bootstrap.pypa.io/get-pip.py and execute it as `python3 get-pip.py --user`.

Tutorial

Getting started

We are assuming you already installed Kpop. The easiest way to start is to simply type the following command on the terminal:

```
$ kpop shell
```

This will start a Python session with all basic Kpop functionality available. It also also tries to load all population files in the current directory, so they become easily available.

The kpop shell is just a convenience method of starting a IPython shell with a few useful imports:

This is useful for interactive and exploratory work. However, for more serious jobs you probably should make those imports manually and avoid the start import in the first line.

Basic Kpop concepts

You notice that most of Kpop interactions go through two main object types: *kpop.Individual* and *kpop.Population*. Let us start with the first of these two, *kpop.Individual*, which represent single individuals by their corresponding genotypes.

Individual

An *kpop.Individual* instance behaves basically as a list of genotype values. Kpop represents genotypes by numbers, where zero is used to encode missing data and numbers above one represent each allele. We can start a new individual by constructing it from a list of pairs of numbers:

```
>>> ind = Individual([[1, 1], [1, 2], [2, 2], [1, 2]])
```

This is a genotype with 4 loci of biallelic data. You might expect it behave just as a list of genotypes for each locus. It accepts Python indexing, slicing and iteration:

```
>>> ind[0]
array([1, 1], dtype=uint8)
```

```
>>> [(1 in locus) for locus in ind]
[True, True, False, True]
```

kpop.Individual objects can also be inspected in several ways.

```
>>> ind.num_loci, ind.ploidy, ind.is_biallelic
(4, 2, True)
```

You should use the autocomplete feature of Kpop's shell to discover more attributes. Just type `ind.` and hit the <tab> key to see a list of completions. Some of those options are methods (you will notice it by the open-close parens at the end of their names). In order to get help on the methods behavior and signature, just use the `?` helper as bellow

```
>>> ind.breed?
Signature: ind.breed(other, id=None, **kwargs)
   doctest:
Breeds with other individual.
<NEWLINE>
Creates a new genotype in which features are selected from both
parents.
File:      ~/git/bio/kpop/src/kpop/individual.py
Type:      method
```

You will notice that if you print an Individual in the terminal it will shown with the following notation

```
>>> ind
Individual('ind: 11 12 22 12')
```

This is actually a different way to construct *kpop.Individual* instances. The first part in the string before the column is a label used to identify the given individual and everything on the right hand side is its genotype.

Let us create a second individual to interact with the first.

```
>>> ind2 = Individual('ind2: 22 11 12 12')
>>> ind2.breed(ind)
Individual('ind2_: 21 12 12 12')
```

Of course, handling a handful of individuals is not very useful. Let us create a list of individuals by drawing samples from an specific probability. First, define a list of probabilities for each allele in each loci

```
>>> freqs = [[0.1, 0.9], [0.5, 0.5], [0.9, 0.1], [0.5, 0.5]]
```

Now we can create a random individual using the `from_freqs` method of the Individual class

```
>>> random_ind = Individual.from_freqs(freqs)
```

... and now we create a bunch:

```
>>> list_of_individuals = []
>>> for _ in range(10):
...     new_ind = Individual.from_freqs(freqs)
...     list_of_individuals.append(new_ind)
```

Population

Now that we have a bunch of individuals, we can make a population. Of course we could use the list of individuals directly, but Kpop provides the much more convenient *kpop.Population* type to represent a group of individuals.

```
>>> popA = Population(list_of_individuals, id='A')
>>> popA
  ind1: 22 21 12 22
  ind2: 22 11 11 21
  ind3: 22 11 11 21
  ind4: 22 11 11 21
  ind5: 22 11 11 12
  ind6: 22 22 11 21
  ind7: 22 11 11 21
  ind8: 22 21 11 22
  ind9: 22 12 11 12
 ind10: 22 22 11 21
```

We created the Population object from a list of individuals and gave it an optional label. The label is used to identify the population in several different contexts such as clustering, plotting, etc.

Just like *kpop.Individual* instances, *kpop.Population* objects have many associated methods and attributes. You can explore it by typing `popA.` and hitting the <tab> key (you will notice it is way more complex than Individual instances).

In population genetics we are usually interested in comparing different populations rather than different individuals in the same population. We can easily create a new random population using the Population.make_random function:

```
>>> popB = Population.random(10, num_loci=4, id='B')
```

This will create a new population with 10 individuals and 4 loci. Now, let us compose this population with the previous one by creating a new generation that breeds individuals from the first population with the second

```
>>> popC = popA.simulation.breed(popB, size=15, id='C')
```

We can combine all sub-populations into a single population containing all individuals by simply adding the population objects together

```
>>> pop_all = popA + popB + popC
```

This creates a *kpop.MultiPopulation* object which behaves essentially as a Population, but keeps track of sub-structuring.

Visualization

Kpop implements a few visualization methods through the Population.plot attribute. The `population.plot.?` namespace has methods for dimensionality reduction (such as PCA),

Statistics

Admixture

Admixture analysis is the task of estimating the admixture coefficients of each individual in a population. This is the main concern of programs such as Structure and ADMIXTURE.

Projections

All dimensionality reduction methods from the above section are implemented in the `population.projection` namespace. Those methods provide the raw data for dimensionality reduction and may be useful in contexts other than data visualization.

TODO.

Clusterization

Clusterization is the task of spliting data into separate groups without providing a training set on correct classifications. This is often refered as "unsupervised learning". Notice here that "unsupervised" does not mean "completely independent of human intervention" since almost all clustering algorithms requires some sort of tuning.

Kpop provides a few methods for performing clustering of individuals. They are all implemented under the `population.cluster` namespace.

TODO.

Classification

Differently from clustering, a supervised classification task learns from a dataset in which all items are classified with a corresponding label. A classification task is useful when it can generalize this mapping to data points outside of the training set.

In Population genetics this often maps to the sittuation in which we have a group of individuals with known parental populations and we want to classify additional specimens into one of those populations. Notice it is different from admixture analysis that tries to infer the fractions of DNA belonging to each parental population. Here the classification is sharp: the individual is said to belong to a single parental population.

All classification methods live under the `population.classification` namespace.

TODO

Classification

Kpop population objects have builtin tools for building classifiers from population objects. As with any classification task, you must provide a labeled training data set and the classifier algorithm will be trained to replicate those labels and generalize to new sample points. A very basic classification task can start with a population object and a list of labels:

```
>>> from kpop import Population
>>> pop = Population.random(5, 10)
>>> classifier = pop.classification(['A', 'A', 'B', 'B', 'A'])
```

The method returns a trained classifier object that associates each individual in the population with the given labels. Notice that we created a random population with 5 individuals and we had to provide the same number of labels.

Classifier objects are used a callable that receive a single population argument. It then returns a list of labels corresponding to the assigned classification of each individual. When we classify the training set, there is a fair chance of obtaining the original labels:

```
>>> classifier(pop)
['A', 'A', 'B', 'B', 'A']
```

The classifier exposes different classification algorithms that can be accessed either using the `pop.classification(labels, <method>)` method or using the corresponding attribute `pop.classification.<method>(labels)`. For instance, we could try different classifiers

```
>>> labels = ['A', 'A', 'B', 'B', 'A']
>>> nb = pop.classification.naive_bayes(labels)
>>> svm = pop.classification.svm(labels)
```

You can check the **:cls:‘kpop.population.classification.Classification‘** to see all available classifiers.

Easy labels

The default procedure for training a classifier involves passing a list of labels for the training algorithms. Sometimes those labels can be stored as meta data in the population object or can be derived from the population somehow. If the labels argument is a string, kpop will try to obtain the label list by using the first option valid option:

- Use population.meta[<label>], if it exists.
- If label equals ‘ancestry’, it creates a list of labels assigning the

id of each sub-population to all its individuals. * If label is the empty string or None, it looks for a ‘labels’ column in the meta information and then returns it.

This interface makes it very convenient to train classifiers to infer population ancestry. Remember that this is not an admixture analysis since we are assuming that all individuals belong to a single population.

```
>>> popA = Population.random(5, 20, id='A')
>>> popB = Population.random(5, 20, id='B')
>>> pop = popA + popB
>>> classifier = pop.classification(labels='ancestry')
>>> classifier(popA)
```

```
['A', 'A', 'A', 'A', 'A']
>>> classifier(popB)
['B', 'B', 'B', 'B', 'B']
```

Probabilistic classifiers

Some classifiers allow for probabilistic classification. That is, instead of assigning a single label per individual, it assigns a probability distribution with the probability that each individual belongs to each label. This is accomplished by the `.prob_*` methods of the classifier. Each method represents the probability distribution in a different way.

```
>>> probs = classifier.prob_list(popA)
>>> probs[0]
Prob({'A': 0.951, 'B': 0.049})
```

API docs

class kpop.population.classification.**Classification**

Implements the population.classification attribute.

naive_bayes(*labels=None*, *data='count'*, *prior='uniform'*, *alpha=0.5*)

Classify objects using the naive_bayes classifier.

Parameters

- **labels** – List of labels or a string with the metadata column used as label. Optionally, the 'ancestry' string classify using the sub-populations as labels.
- **alpha** – Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
- **prior** – The prior probability for each label. Must be either a Prob() object, the string 'uniform' or None. The default value is 'uniform' that assigns a fixed uniform prior. If prior is None, it learns priors from data. Finally, it can also be specified as a Prob() object or a mapping from labels to probabilities.

sklearn(*classifier*, *labels=None*, *data='count'*, ***kwargs*)

Uses a scikit learn classifier to classify population.

Parameters

- **classifier** – A scikit learn classifier class (e.g., sklearn.naive_bayes.BernoulliNB)
- **labels** – A sequence of labels used to train the classifier.
- **data** (*str*) – The method used to convert the population to a usable data set. It uses the same options as in the `Population.as_array()` method.

svm(*labels=None*, *data='count'*, ***kwargs*)

Classify objects using the Support Vector Machine (SVM) classifier.

Examples

Simulating synthetic populations

API Reference

API documentation for the `kpop` module.

Individual

Each element of a population is an instance of *kpop.Individual*. An *kpop.Individual* behave similarly as a list of genetypes or as an 2D array of genotypes.

class `kpop.`**Individual**(*data*, *id=None*, *population=None*, *allele_names=None*, *dtype=None*, *meta=None*, *admixture_q=None*, *num_alleles=None*)

Represents a single individual genotype.

A genotype data must be an integer array of shape (num_loci, ploidy).

Parameters

- **data** – Can be either a string of values or a list of raw genotype values represented as integers.
- **population** – Population to which individual belongs to.

num_loci
Number of loci in the raw genotype data

ploidy
Genotype's ploidy

data
A numpy array of integers with genotype data. Allele types are represented sequentially by 1, 2, 3, etc. Missing data is represented by zero. By default, data is stored in uint8 form. This supports up to 255 different allele types plus zero.

allele_names
A list of mappings between allele integer values to a character representation. If not given, it inherits from parent population.

breed(*other*, *id=None*, ***kwargs*)
Breeds with other individual.

Creates a new genotype in which features are selected from both parents.

copy(*data=None*, ***, *meta=<object object>*, ***kwargs*)
Creates a copy of individual.

count(*value*) → integer – return number of occurrences of value

classmethod **from_freqs**(*freqs*, *ploidy=2*, ***kwargs*)
Returns a random individual from the given frequency distribution.

Parameters

- **freqs** – A frequency distribution. Can be a sequence of Prob() elements or an square array of frequencies.
- **ploidy** – Individuals ploidy.

- ****kwargs** – Additional keyword arguments passed to the constructor.

Returns A new `Individual` instance.

haplotypes()
Return a sequence of ploidy arrays with each haplotype.

This operation is a simple transpose of genotype data.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

render(*id_align=None*, *max_loci=None*)
Renders individual genotype.

render_csv(*sep=', '*)
Render individual in CSV.

render_ped(*family_id='FAM001'*, *individual_id=0*, *paternal_id=0*, *maternal_id=0*, *sex=0*, *phenotype=0*, *memo=None*)
Render individual as a line in a plink's .ped file.

Parameters

- **family_id** – A string or number representing the individual's family.
- **individual_id** – A number representing the individual's id.
- **maternal_id** (*paternal_id,*) – A number representing the individuals father/mother's id.
- **sex** – The sex (1=male, 2=female, other=unknown).
- **phenotype** – A number representing the optional phenotype.

Population objects

The main type in the kpop package is *kpop.Population*. A population is basically a list of individuals. It has a similar interface as a Python's list or a Numpy array.

class kpop.**Population**(*data=()*, *id=None*, *individual_ids=None*, ***kwargs*)
A Population is a collection of individuals.

as_array(*which='raw'*)
Convert to a numpy data array using the requested conversion method. This is a basic pre-processing step in many dimensionality reduction algorithms.

Genotypes are categorical data and usually it doesn't make sense to treat the integer encoding used in kpop as ordinal data (there is no ordering implied when treating say, allele 1 vs allele 2 vs allele 3).

Conversion methods:

- **raw:** An 3 dimensional array of (size, num_loci, ploidy) for raw genotype data. Each component represents the value of a single allele.
- **flat:** Like raw, but flatten the last dimension into a (size, num_loci * ploidy) array. This creates a new feature per loci for each degree of ploidy in the data.
- **rflat:** Flatten data, but first shuffle the positions of alleles at each loci. This is recommended if data does not carry reliable haplotype information.
- **raw-norm, flat-norm, rflat-norm:** Normalized versions of "raw", "flat", and "rflat" methods. All components are rescaled with zero mean and unity variance.

- **count:** Force conversion to biallelic data and counts the number of occurrences of the first allele. Most methdds will require normalization, so you probably should consider an specific method such as count-unity, count-snp, etc
- **count-norm:** Normalized version of count scaled to zero mean and unity variance.
- **count-snp:** Normalizes each feature using the standard deviation expected under the assumption of Hardy-Weinberg equilibrium. This procedure is described at Patterson et. al., "Population Structure and Eigenanalysis" and is recommended for SNPs subject to genetic drift.
- **count-center:** Instead of normalizing, simply center data by subtracting half the ploidy to place it into a symmetric range. This normalization puts data into a cube with a predictable origin and range. For diploid data, the components will be either -1, 0, or 1.

Returns An ndarray with transformed data.

count(*value*) → integer – return number of occurrences of value

drop_individuals(*indexes*, ***kwargs*)
Creates new population removing the individuals in the given indexes.

drop_loci(*indexes*, ***kwargs*)
Create a new population with all loci in the given indexes removed.

drop_missing_data(*axis=0*, *thresh=0.0*, ***kwargs*)
Drop all individuals or loci that have a proportion of missing data higher than the given threshold.

Parameters

- **axis** (*0 or 1*) – If axis=0 or 'individuals' (default), it will scan individuals with a minimum amount of missing data values. If axis=1 or 'loci', it will drop all loci with the minimum ammount of missing data.
- **thresh** (*float, between 0 and 1*) – The maximum proportion of missing data tolerated.

Returns A new population.

drop_non_biallelic(***kwargs*)
Creates a new population removing all non-biallelic loci.

find_missing_data(*axis=0*, *thresh=0.0*)
Return the indexes for all all individuals or loci that have a proportion of missing data higher than the given threshold.

Parameters

- **axis** (*0 or 1*) – If axis=0 or 'individuals' (default), it will scan individuals with a minimum amount of missing data values. If axis=1 or 'loci', it will drop all loci with the minimum ammount of missing data.
- **thresh** (*float, between 0 and 1*) – The maximum proportion of missing data tolerated.

Returns An array of indexes.

find_non_biallelic()
Finds all non-biallelic loci in population.

force_biallelic(***kwargs*)
Return a new population with forced biallelic data.

If a locus has more than 2 alleles, the most common allele is picked as allele 1 and the alternate allele 2 comprises all the other alleles.

freqs
Return a list of Prob instances representing the frequencies in each locus.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

random(*size=0*, *num_loci=0*, *alleles=2*, *ploidy=2*, *id=None*, *seed=None*)
Creates a new random population.

Parameters

- **size** – Number of individuals. If a list of numbers is given, creates a Multipopulation object with sub-populations of the assigned sizes.
- **num_loci** – Number of loci in the genotype.
- **alleles** – Number of alleles for all loci.
- **ploidy** – Ploidy of genotype.
- **min_prob** – Minimum value for a frequency probability.

Returns A new population object.

shuffle_loci(***kwargs*)
Return a copy with shuffled contents of each locus.

size
Return the number of items in a container.

sort_by_allele_freq(***kwargs*)
Return a new population in which the index attributed to each allele in each locus is sorted by the frequency in the population. After that, allele 1 will be the most common, allele 2 is the second most common and so on.

Population vs Multipopulation

Kpop uses two classes to represent populations that have basically the same interface. A MultiPopulation is basically a population structured with many sub-populations.

class kpop.**MultiPopulation**(*populations=()*, *freqs=None*, ***kwargs*)
A population formed by several sub-populations.

add_population(*population*)
Adds a new sub-population.

Parameters **population** – A Population instance.

as_array(*which='raw'*)
Convert to a numpy data array using the requested conversion method. This is a basic pre-processing step in many dimensionality reduction algorithms.

Genotypes are categorical data and usually it doesn't make sense to treat the integer encoding used in kpop as ordinal data (there is no ordering implied when treating say, allele 1 vs allele 2 vs allele 3).

Conversion methods:

- **raw:** An 3 dimensional array of (size, num_loci, ploidy) for raw genotype data. Each component represents the value of a single allele.

- **flat:** Like raw, but flatten the last dimension into a (size, num_loci * ploidy) array. This creates a new feature per loci for each degree of ploidy in the data.
- **rflat:** Flatten data, but first shuffle the positions of alleles at each loci. This is recommended if data does not carry reliable haplotype information.
- **raw-norm, flat-norm, rflat-norm:** Normalized versions of "raw", "flat", and "rflat" methods. All components are rescaled with zero mean and unity variance.
- **count:** Force conversion to biallelic data and counts the number of occurrences of the first allele. Most methdds will require normalization, so you probably should consider an specific method such as count-unity, count-snp, etc
- **count-norm:** Normalized version of count scaled to zero mean and unity variance.
- **count-snp:** Normalizes each feature using the standard deviation expected under the assumption of Hardy-Weinberg equilibrium. This procedure is described at Patterson et. al., "Population Structure and Eigenanalysis" and is recommended for SNPs subject to genetic drift.
- **count-center:** Instead of normalizing, simply center data by subtracting half the ploidy to place it into a symmetric range. This normalization puts data into a cube with a predictable origin and range. For diploid data, the components will be either -1, 0, or 1.

Returns An ndarray with transformed data.

count(*value*) → integer – return number of occurrences of value

drop_individuals(*indexes*, ***kwargs*)
Creates new population removing the individuals in the given indexes.

drop_loci(*indexes*, ***kwargs*)
Create a new population with all loci in the given indexes removed.

drop_missing_data(*axis=0*, *thresh=0.0*, ***kwargs*)
Drop all individuals or loci that have a proportion of missing data higher than the given threshold.

Parameters

- **axis** (*0 or 1*) – If axis=0 or 'individuals' (default), it will scan individuals with a minimum amount of missing data values. If axis=1 or 'loci', it will drop all loci with the minimum ammount of missing data.
- **thresh** (*float, between 0 and 1*) – The maximum proportion of missing data tolerated.

Returns A new population.

drop_non_biallelic(***kwargs*)
Creates a new population removing all non-biallelic loci.

find_missing_data(*axis=0*, *thresh=0.0*)
Return the indexes for all all individuals or loci that have a proportion of missing data higher than the given threshold.

Parameters

- **axis** (*0 or 1*) – If axis=0 or 'individuals' (default), it will scan individuals with a minimum amount of missing data values. If axis=1 or 'loci', it will drop all loci with the minimum ammount of missing data.
- **thresh** (*float, between 0 and 1*) – The maximum proportion of missing data tolerated.

Returns An array of indexes.

find_non_biallelic()
Finds all non-biallelic loci in population.

force_biallelic(***kwargs*)
Return a new population with forced biallelic data.

If a locus has more than 2 alleles, the most common allele is picked as allele 1 and the alternate allele 2 comprises all the other alleles.

freqs
Return a list of Prob instances representing the frequencies in each locus.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

random(*size=0*, *num_loci=0*, *alleles=2*, *ploidy=2*, *id=None*, *seed=None*)
Creates a new random population.

Parameters

- **size** – Number of individuals. If a list of numbers is given, creates a Multipopulation object with sub-populations of the assigned sizes.
- **num_loci** – Number of loci in the genotype.
- **alleles** – Number of alleles for all loci.
- **ploidy** – Ploidy of genotype.
- **min_prob** – Minimum value for a frequency probability.

Returns A new population object.

shuffle_loci(***kwargs*)
Return a copy with shuffled contents of each locus.

size
Return the number of items in a container.

slice_indexes(*indexes*)
Map indexes to a list of indexes for each sub-population.

sort_by_allele_freq(***kwargs*)
Return a new population in which the index attributed to each allele in each locus is sorted by the frequency in the population. After that, allele 1 will be the most common, allele 2 is the second most common and so on.

The `.plot` attribute

Each *kpop.Population* or *kpop.MultiPopulation* instance have a `.plot` attribute that defines a namespace with many different plotting utilities.

Other utility types

Representing probabilities

class kpop.prob.**Prob**(*data*, *normalize=True*, *support=None*)
A dictionary-like object that behaves as a mapping between categories to their respective probabilities.

encode(*coding=None*)
Encode probability distribution as a vector.

Parameters **coding** – a sequence of ordered categories.

Example

```
>>> prob = Prob({'a': 0.75, 'b': 0.25})
>>> prob.encode(['b', 'a'])
[0.25, 0.75]
```

entropy()
Return the Shannon entropy for the probability distribution.

kl_divergence(*q: collections.abc.Mapping*)

Return the Kullback-Leibler divergence with probability distribution.

This is given by the formula:

$KL = sum_i p_i ln

rac {p_i} {q_i},$

in which p_i comes from the probability object and q_i comes from the argument.

max()
Return the value of maximum probability.

classmethod **mixture**(*coeffs*, *probs*)
Create a mixture probability from the given coeffs and list of Probs objects.

Parameters

- **coeffs** – Mixture coefficients. These coefficients do not have to be normalized.
- **probs** – List of Prob objects.

Returns A Prob object representing the mixture.

mode()
Return the element with the maximum probability.

If more than one element shares the maximum probability, return an arbitrary value within this set.

mode_set()
Return a set of elements that share the maximum probability.

random()
Returns a random element.

random_sequence(*size*)
Returns a sequence of random elements.

set_support(*support*)
Defines the support set of distribution.

If elements exist in support, they are forced to exist in distribution, possibly with zero probability. If element exists in the distribution but is not present in support, raises a ValueError.

sharp(*mode_set=True*)
Return a sharp version of the probability distribution.

All elements receive probability zero, except the mode which receives probability one.

update_support(*support*)
Force all elements in support to be explicitly present in distribution (possibly with null probability).

Parameters **support** – a list of elements in the support set for probability distribution.

Utility modules

Plotting

`kpop.plots` contains a few useful plotting functions based on matplotlib.

Loading objects

Functions from the `kpop.loaders` module are responsible for loading Population objects from files.

Frequently asked questions

Usage

Who is this for?

Kpop was created for people in the population genetics community.

License

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

S

U