

# Documentation

Peter Occil

This version of the document is dated 2026-02-15.

problem in /home/peter/Documents/SharpDevelopProjects/peteroupc.github.io/betadist.py - SyntaxError: in  
Help on module fixed:

## NAME

fixed

## CLASSES

builtins.object  
Fixed

```
class Fixed(builtins.object)
|   Fixed(i)
|
|   Fixed-point numbers, represented using integers that store multiples
|   of 2n-BITS. They are not necessarily faster than floating-point numbers, nor
|   do they necessarily have the same precision or resolution of floating-point
|   numbers. The main benefit of fixed-point numbers is that they improve
|   determinism for applications that rely on noninteger real numbers (notably
|   simulations and machine learning applications), in the sense that the operations
|   given here deliver the same answer for the same input across computers,
|   whereas floating-point numbers have a host of problems that make repeatable
|   results difficult, including differences in their implementation, rounding
|   behavior, and order of operations, as well as nonassociativity of
|   floating-point numbers.
|
|   The operations given here are not guaranteed to be "constant-time"
|   (nondata-dependent and branchless) for every relevant input.
|
|   Any copyright to this file is released to the Public Domain. In case this is not
|   possible, this file is also licensed under Creative Commons Zero version 1.0.
|
|   Methods defined here:
|
|   __abs__(self)
|
|   __add__(a, b)
|
|   __cmp__(self, other)
|
|   __div__(a, b)
```

```

|  __eq__(self, other)
|      Return self==value.
|
|  __float__(a)
|
|  __floordiv__(a, b)
|
|  __ge__(self, other)
|      Return self>=value.
|
|  __gt__(self, other)
|      Return self>value.
|
|  __init__(self, i)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __int__(a)
|
|  __le__(self, other)
|      Return self<=value.
|
|  __lt__(self, other)
|      Return self<value.
|
|  __mod__(a, b)
|
|  __mul__(a, b)
|
|  __ne__(self, other)
|      Return self!=value.
|
|  __neg__(self)
|
|  __pos__(self)
|
|  __rdiv__(a, b)
|
|  __repr__(self)
|      Return repr(self).
|
|  __rtruediv__(a, b)
|
|  __str__(self)
|      Return str(self).
|
|  __sub__(a, b)
|
|  __truediv__(a, b)
|
|  acos(a)
|      Calculates an approximation of the inverse cosine of the specified number.

```

```

|
| asin(a)
|     Calculates an approximation of the inverse sine of the specified number.
|
| atan2(y, x)
|     Calculates the approximate measure, in radians, of the angle formed by the
|     x-axis and a line determined by the origin and the specified coordinates of a 2D
|     point. This is also known as the inverse tangent.
|
| cos(a)
|     Calculates the approximate cosine of the specified angle; the angle is in radians.
|     For the fraction size used by this class, this method is accurate to within
|     1 unit in the last place of the correctly rounded result for every input
|     in the range  $[-\pi*2, \pi*2]$ .
|     This method's accuracy decreases beyond that range.
|
| exp(a)
|     Calculates an approximation of e (base of natural logarithms) raised
|     to the power of this number. May raise an error if this number
|     is extremely high.
|
| floor(a)
|
| log(a)
|     Calculates an approximation of the natural logarithm of this number.
|
| pow(a, b)
|     Calculates an approximation of this number raised to the power of another number.
|
| round(a)
|
| sin(a)
|     Calculates the approximate sine of the specified angle; the angle is in radians.
|     For the fraction size used by this class, this method is accurate to within
|     1 unit in the last place of the correctly rounded result for every input
|     in the range  $[-\pi*2, \pi*2]$ .
|     This method's accuracy decreases beyond that range.
|
| sqrt(a)
|     Calculates an approximation of the square root of the specified number.
|
| tan(a)
|     Calculates the approximate tangent of the specified angle; the angle is in radians.
|     For the fraction size used by this class, this method is accurate to within
|     2 units in the last place of the correctly rounded result for every input
|     in the range  $[-\pi*2, \pi*2]$ .
|     This method's accuracy decreases beyond that range.
|
| -----
| Static methods defined here:
|
| v(i)

```

```

|     Converts a string, integer, Decimal, or other number type into
|     a fixed-point number.  If the parameter is a Fixed, returns itself.
|     If the specified number is a noninteger, returns the closest value to
|     a Fixed after rounding using the round-to-nearest-ties-to-even
|     rounding mode.  The parameter is recommended to be a string
|     or integer, and is not recommended to be a `float`.
|
|     -----
|     Data descriptors defined here:
|
|     __dict__
|         dictionary for instance variables
|
|     __weakref__
|         list of weak references to the object
|
|     -----
|     Data and other attributes defined here:
|
|     ArcTanBitDiff = 9
|
|     ArcTanFrac = 29
|
|     ArcTanHTable = [0, 294906490, 137123709, 67461703, 33598225, 16782680,...
|
|     ArcTanTable = [421657428, 248918914, 131521918, 66762579, 33510843, 16...
|
|     BITS = 20
|
|     ExpK = 648270061
|
|     HALF = 524288
|
|     HalfPiArcTanBits = 843314856
|
|     HalfPiBits = 1647099
|
|     HalfPiHighRes = 130496653328243011213339889301986179
|
|     HighResFrac = 116
|
|     Ln2ArcTanBits = 372130559
|
|     Log2Bits = 726817
|
|     LogMin = 157286
|
|     MASK = 1048575
|
|     PiAndHalfHighRes = 391489959984729033640019667905958538
|
|     PiArcTanBits = 1686629713

```

```
|
| PiBits = 3294199
|
| PiHighRes = 260993306656486022426679778603972359
|
| QuarterPiArcTanBits = 421657428
|
| SinCosK = 326016435
|
| TwoTimesPiArcTanBits = 3373259426
|
| TwoTimesPiBits = 6588397
|
| TwoTimesPiHighRes = 521986613312972044853359557207944718
|
| __hash__ = None
```

#### FILE

```
/home/peter/Documents/SharpDevelopProjects/peteroupc.github.io/fixed.py
```

Help on module bernoulli:

#### NAME

```
bernoulli
```

#### CLASSES

```
builtins.object
    Bernoulli
    DiceEnterprise
    PolynomialSim
```

```
class Bernoulli(builtins.object)
| This class contains methods that generate Bernoulli random numbers,
| (either 1 or heads with a given probability, or 0 or tails otherwise).
| This class also includes implementations of so-called "Bernoulli factories", algorithms
| that sample a new probability given a coin that shows heads with an unknown probability.
| Written by Peter O.
|
| References:
| - Flajolet, P., Pelletier, M., Soria, M., "On Buffon machines and numbers",
| arXiv:0906.5560v2 [math.PR], 2010.
| - Huber, M., "Designing perfect simulation algorithms using local correctness",
| arXiv:1907.06748v1 [cs.DS], 2019.
| - Huber, M., "Nearly optimal Bernoulli factories for linear functions",
| arXiv:1308.1562v2 [math.PR], 2014.
| - Huber, M., "Optimal linear Bernoulli factories for small mean problems",
| arXiv:1507.00843v2 [math.PR], 2016.
| - Łatuszyński, K., Kosmidis, I., Papaspiliopoulos, O., Roberts, G.O., "Simulating
| events of unknown probabilities via reverse time martingales", arXiv:0907.4018v2
| [stat.CO], 2009/2011.
| - Goyal, V. and Sigman, K. 2012. On simulating a class of Bernstein
| polynomials. ACM Transactions on Modeling and Computer Simulation 22(2),
```

```

| Article 12 (March 2012), 5 pages.
| - Giulio Morina. Krzysztof Łatuszyński. Piotr Nayar. Alex Wendland. "From the Bernoulli factory
|   doi.org/10.1214/21-AAP1679
| - Dughmi, Shaddin, Jason Hartline, Robert D. Kleinberg, and Rad Niazadeh. "Bernoulli factories
| - Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. (2017). Exact Monte
| Carlo likelihood-based inference for jump-diffusion processes.
| - Vats, D., Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. Efficient
| Bernoulli factory MCMC for intractable posteriors, Biometrika 109(2), June 2022.
| - Mendo, Luis. "An asymptotically optimal Bernoulli factory for certain
| functions that can be expressed as power series." Stochastic Processes and their
| Applications 129, no. 11 (2019): 4366-4384.
| - Canonne, C., Kamath, G., Steinke, T., "The Discrete Gaussian
| for Differential Privacy", arXiv:2004.00010 [cs.DS], 2020.
| - Lee, A., Doucet, A. and Łatuszyński, K., 2014. Perfect simulation using
| atomic regeneration with application to Sequential Monte Carlo,
| arXiv:1407.5770v1 [stat.CO]
|
| Methods defined here:
|
| __init__(self)
|     Creates a new instance of the Bernoulli class.
|
| a_bag_div_b_bag(self, numerator, numbag, intpart, bag)
|     Simulates (numerator+numbag)/(intpart+bag).
|
| a_div_b_bag(self, numerator, intpart, bag)
|     Simulates numerator/(intpart+bag).
|
| add(self, f1, f2, eps=Fraction(1, 20))
|     Addition Bernoulli factory:  $B(p), B(q) \Rightarrow B(p+q)$  (Dughmi et al. 2021)
|     - f1, f2: Functions that return 1 if heads and 0 if tails.
|     - eps: A Fraction in (0, 1). eps must be chosen so that  $p+q \leq 1 - \text{eps}$ ,
|       where p and q are the probability of heads for f1 and f2, respectively.
|
| alt_series(self, f, series)
|     Alternating-series Bernoulli factory:  $B(p) \rightarrow B(s[0] - s[1]*p + s[2]*p^2 - \dots)$ 
|     (Łatuszyński et al. 2011).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - series: Object that generates each coefficient of the series starting with the first.
|       Each coefficient must be less than or equal to the previous and all of them must
|       be 1 or less.
|     Implements the following two methods: reset() resets the object to the first
|     coefficient; and next() generates the next coefficient.
|
| arctan_n_div_n(self, f)
|     Arctan div N:  $B(p) \rightarrow B(\arctan(p)/p)$ . Uses a uniformly-fast special case of
|     the two-coin Bernoulli factory, rather than the even-parity construction in
|     Flajolet's paper, which does not have bounded expected running time for all heads probabilities.
|     Reference: Flajolet et al. 2010.
|     - f: Function that returns 1 if heads and 0 if tails.
|
| bernoulli_x(self, f, x)

```

```

| Bernoulli factory with a given probability:  $B(p) \Rightarrow B(x)$  (Mendo 2019).
|     Mendo calls Bernoulli factories "nonrandomized" if their randomness
|     is based entirely on the underlying coin.
|     - f: Function that returns 1 if heads and 0 if tails.
|     - x: Desired probability, in  $[0, 1]$ .
|
| bernstein(self, f, alpha)
|     Polynomial Bernoulli factory:  $B(p) \Rightarrow B(\text{Bernstein}(\alpha))$ 
|     (Goyal and Sigman 2012).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - alpha: List of Bernstein coefficients for the polynomial (when written
|       in Bernstein form),
|       whose degree is this list's length minus 1.
|       For this to work, each coefficient must be in  $[0, 1]$ .
|
| coin(self, c)
|     Convenience method to generate a function that returns
|     1 (heads) with the specified probability c (which must be in  $[0, 1]$ )
|     and 0 (tails) otherwise.
|
| complement(self, f)
|     Complement (NOT):  $B(p) \Rightarrow B(1-p)$  (Flajolet et al. 2010)
|     - f: Function that returns 1 if heads and 0 if tails.
|
| conditional(self, f1, f2, f3)
|     Conditional:  $B(p), B(q), B(r) \Rightarrow B((1-r)*q+r*p)$  (Flajolet et al. 2010)
|     - f1, f2, f3: Functions that return 1 if heads and 0 if tails.
|
| cos(self, f)
|     Cosine Bernoulli factory:  $B(p) \Rightarrow B(\cos(p))$ . Special
|     case of Algorithm3 of reverse-time martingale paper.
|
| disjunction(self, f1, f2)
|     Disjunction (OR):  $B(p), B(q) \Rightarrow B(p+q-p*q)$  (Flajolet et al. 2010)
|     - f1, f2: Functions that return 1 if heads and 0 if tails.
|
| divoneplus(self, f)
|     Divided by one plus p:  $B(p) \Rightarrow B(1/(1+p))$ , implemented
|     as a special case of the two-coin construction. Prefer over even-parity
|     for having bounded expected running time for all heads probabilities.
|     - f: Function that returns 1 if heads and 0 if tails.
|     Note that this function is slow as the probability of heads approaches 1.
|
| eps_div(self, f, eps)
|     Bernoulli factory as follows:  $B(p) \rightarrow B(\text{eps}/p)$  (Lee et al. 2014).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - eps: Fraction in  $(0, 1)$ , must be chosen so that  $\text{eps} < p$ , where p is
|       the probability of heads.
|
| evenparity(self, f)
|     Even parity:  $B(p) \Rightarrow B(1/(1+p))$  (Flajolet et al. 2010)
|     - f: Function that returns 1 if heads and 0 if tails.

```

```

|     Note that this function is slow as the probability of heads approaches 1.
|
| exp_minus(self, f)
|     Exp-minus Bernoulli factory:  $B(p) \rightarrow B(\exp(-p))$  (Łatuszyński et al. 2011).
|     - f: Function that returns 1 if heads and 0 if tails.
|
| exp_minus_ext(self, f, c=0)
|     Extension to the exp-minus Bernoulli factory of (Łatuszyński et al. 2011):
|      $B(p) \rightarrow B(\exp(-p - c))$ 
|     To the best of my knowledge, I am not aware
|         of any article or paper that presents this particular
|         Bernoulli factory (before my articles presenting
|         accurate beta and exponential generators).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - c: Integer part of exp-minus. Default is 0.
|
| fill_geometric_bag(self, bag, precision=53)
|
| geometric_bag(self, u)
|     Bernoulli factory for a uniformly-distributed random number in (0, 1)
|     (Flajolet et al. 2010).
|     - u: List that holds the binary expansion, from left to right, of the uniformly-
|         distributed random number. Each element of the list is 0, 1, or None (meaning
|         the digit is not yet known). The list may be expanded as necessary to put
|         a new digit in the appropriate place in the binary expansion.
|
| linear(self, f, cx, cy=1, eps=Fraction(1, 20))
|     Linear Bernoulli factory:  $B(p) \Rightarrow B((cx/cy)*p)$  (Huber 2016).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - cx, cy: numerator and denominator of c; the probability of heads (p) is multiplied
|         by c. c must be 0 or greater. If  $c > 1$ , c must be chosen so that  $c*p \leq 1 - \text{eps}$ .
|     - eps: A Fraction in (0, 1). If  $c > 1$ , eps must be chosen so that  $c*p \leq 1 - \text{eps}$ .
|
| linear_lowprob(self, f, cx, cy=1, m=Fraction(249, 500))
|     Linear Bernoulli factory which is faster if the probability of heads is known
|         to be less than half:  $B(p) \Rightarrow B((cx/cy)*p)$  (Huber 2016).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - cx, cy: numerator and denominator of c; the probability of heads (p) is multiplied
|         by c. c must be 0 or greater. If  $c > 1$ , c must be chosen so that  $c*p \leq m < 1/2$ .
|     - m: A Fraction in (0, 1/2). If  $c > 1$ , m must be chosen so that  $c*p \leq m < 1/2$ .
|
| linear_power(self, f, cx, cy=1, i=1, eps=Fraction(1, 20))
|     Linear-and-power Bernoulli factory:  $B(p) \Rightarrow B((p*cx/cy)^i)$  (Huber 2019).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - cx, cy: numerator and denominator of c; the probability of heads (p) is multiplied
|         by c. c must be 0 or greater. If  $c > 1$ , c must be chosen so that  $c*p \leq 1 - \text{eps}$ .
|     - i: The exponent. Must be an integer and 0 or greater.
|     - eps: A Fraction in (0, 1). If  $c > 1$ , eps must be chosen so that  $c*p \leq 1 - \text{eps}$ .
|
| logistic(self, f, cx=1, cy=1)
|     Logistic Bernoulli factory:  $B(p) \rightarrow B(cx*p/(cy+cx*p))$  or
|          $B(p) \rightarrow B((cx/cy)*p/(1+(cx/cy)*p))$  (Morina et al. 2019)

```



```

|     - f: Function that returns 1 if heads and 0 if tails. Note that this function can
|       be slow as the probability of heads approaches 0.
|     - cx, cy: numerator and denominator of c; the probability of heads (p) is multiplied
|       by c. c must be in (0, 1).
|
| martingale(self, coin, coeff)
|     General martingale algorithm for alternating power
|     series.
|     'coin' is the coin to be flipped; 'coeff' is a function
|     that takes an index 'i' and calculates the coefficient
|     for index 'i'. Indices start at 0.
|
| mean(self, f1, f2)
|     Mean:  $B(p)$ ,  $B(q) \Rightarrow B((p+q)/2)$  (Flajolet et al. 2010)
|     - f1, f2: Functions that return 1 if heads and 0 if tails.
|
| old_linear(self, f, cx, cy=1, eps=Fraction(1, 20))
|     Linear Bernoulli factory:  $B(p) \Rightarrow B((cx/cy)*p)$ . Older algorithm given in (Huber 2014).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - cx, cy: numerator and denominator of c; the probability of heads (p) is multiplied
|       by c. c must be 0 or greater. If  $c > 1$ , c must be chosen so that  $c*p < 1 - \text{eps}$ .
|     - eps: A Fraction in (0, 1). If  $c > 1$ , eps must be chosen so that  $c*p < 1 - \text{eps}$ .
|
| one_div_pi(self)
|     Generates 1 with probability  $1/\pi$ .
|     Reference: Flajolet et al. 2010.
|
| power(self, f, ax, ay=1)
|     Power Bernoulli factory:  $B(p) \Rightarrow B(p^{ax/ay})$ . (case of (0, 1) provided by
|     Mendo 2019).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - ax, ay: numerator and denominator of the desired power to raise the probability
|       of heads to. This power must be 0 or greater.
|
| powerseries(self, f)
|     Power series Bernoulli factory:  $B(p) \Rightarrow B(1 - c(0)*(1-p) + c(1)*(1-p)^2 +$ 
|        $c(2)*(1-p)^3 + \dots)$ , where  $c(i) = \text{c}[i]/\text{sum}(c)$  (Mendo 2019).
|     - f: Function that returns 1 if heads and 0 if tails.
|     - c: List of coefficients in the power series, all of which must be
|       nonnegative integers.
|
| probgenfunc(self, f, rng)
|     Probability generating function Bernoulli factory:  $B(p) \Rightarrow B(E[p^x])$ , where x is rng()
|     (Dughmi et al. 2021).  $E[p^x]$  is the expected value of  $p^x$  and is also known
|     as the probability generating function.
|     - f: Function that returns 1 if heads and 0 if tails.
|     - rng: Function that returns a nonnegative integer at random.
|     Example (Dughmi et al. 2021): if 'rng' is Poisson(lamda) we have
|     an "exponentiation" Bernoulli factory as follows:
|      $B(p) \Rightarrow B(\exp(p*\text{lamda}-\text{lamda}))$ 
|
| product(self, f1, f2)

```

```

|         Product (conjunction; AND):  $B(p), B(q) \Rightarrow B(p \cdot q)$  (Flajolet et al. 2010)
|         - f1, f2: Functions that return 1 if heads and 0 if tails.
|
|     randbit(self)
|         Generates a random bit that is 1 or 0 with equal probability.
|
|     rndint(self, maxInclusive)
|
|     rndintexc(self, maxexc)
|         Returns a random integer in  $[0, \text{maxexc})$ .
|
|     simulate(self, coin, fbelow, fabove, fbound, nextdegree=None)
|         Simulates a general factory function defined by two
|         sequences of polynomials that converge from above and below.
|         - coin(): Function that returns 1 or 0 with a fixed probability.
|         - fbelow(n, k): Calculates the kth Bernstein coefficient (not the value),
|           or a lower bound thereof, for the degree-n lower polynomial (k starts at 0).
|         - fabove(n, k): Calculates the kth Bernstein coefficient (not the value),
|           or an upper bound thereof, for the degree-n upper polynomial.
|         - fbound(n): Returns a tuple or list specifying a lower and upper bound
|           among the values of fbelow and fabove, respectively, for the specified n.
|         - nextdegree(n): Returns a lambda returning the next degree after the specified degree n. f
|           must return an integer greater than n.
|           Optional. If not given, the first degree is 1 and the next degree is  $n \cdot 2$ 
|           (so that for each power of 2 as well as 1, a polynomial of that degree
|           must be specified).
|
|     sin(self, f)
|         Sine Bernoulli factory:  $B(p) \Rightarrow B(\sin(p))$ . Special
|         case of Algorithm3 of reverse-time martingale paper.
|
|     square(self, f1, f2)
|         Square:  $B(p) \Rightarrow B(1-p)$ . (Flajolet et al. 2010)
|         - f1, f2: Functions that return 1 if heads and 0 if tails.
|
|     twocoin(self, f1, f2, c1=1, c2=1, beta=1)
|         Two-coin Bernoulli factory:  $B(p), B(q) \Rightarrow$ 
|            $B(c1 \cdot p \cdot \text{beta} / (\text{beta} * (c1 \cdot p + c2 \cdot q) - (\text{beta} - 1) \cdot (c1 + c2)))$ 
|           (Gonçalves et al. 2017, Vats et al. 2020; in Vats et al.,
|           C1,p1 corresponds to cy and C2,p2 corresponds to cx).
|           Logistic Bernoulli factory is a special case with  $q=1, c2=1, \text{beta}=1$ .
|         - f1, f2: Functions that return 1 if heads and 0 if tails.
|         - c1, c2: Factors to multiply the probabilities of heads for f1 and f2, respectively.
|         - beta: Early rejection parameter ("portkey" two-coin factory).
|           When  $\text{beta} = 1$ , the formula simplifies to  $B(c1 \cdot p / (c1 \cdot p + c2 \cdot q))$ .
|
|     twofacpower(self, fbase, fexponent)
|         Bernoulli factory  $B(p, q) \Rightarrow B(p^q)$ .
|         Based on algorithm from (Mendo 2019),
|         but changed to accept a Bernoulli factory
|         rather than a fixed value for the exponent.
|         To the best of my knowledge, I am not aware

```

```

|         of any article or paper that presents this particular
|         Bernoulli factory (before my articles presenting
|         accurate beta and exponential generators).
|         - fbase, fexponent: Functions that return 1 if heads and 0 if tails.
|           The first is the base, the second is the exponent.
|
|     zero_or_one(self, px, py)
|         Returns 1 at probability px/py, 0 otherwise.
|
|     zero_or_one_arctan_n_div_n(self, x, y=1)
|         Generates 1 with probability  $\arctan(x/y)*y/x$ ; 0 otherwise.
|         x/y must be in [0, 1]. Uses a uniformly-fast special case of
|         the two-coin Bernoulli factory, rather than the even-parity construction in
|         Flajolet's paper, which does not have bounded expected running time for all heads probabilities.
|         Reference: Flajolet et al. 2010.
|
|     zero_or_one_exp_minus(self, x, y)
|         Generates 1 with probability  $\exp(-x/y)$ ; 0 otherwise.
|         Reference: Canonne et al. 2020.
|
|     zero_or_one_log1p(self, x, y=1)
|         Generates 1 with probability  $\log(1+x/y)$ ; 0 otherwise.
|         Reference: Flajolet et al. 2010. Uses a uniformly-fast special case of
|         the two-coin Bernoulli factory, rather than the even-parity construction in
|         Flajolet's paper, which does not have bounded expected running time for all heads probabilities.
|
|     zero_or_one_pi_div_4(self)
|         Generates 1 with probability  $\pi/4$ .
|         Reference: Flajolet et al. 2010.
|
|     zero_or_one_power(self, px, py, n)
|         Generates 1 with probability  $(px/py)^n$  (where n can be
|         positive, negative, or zero); 0 otherwise.
|
|     zero_or_one_power_ratio(self, px, py, nx, ny)
|         Generates 1 with probability  $(px/py)^{(nx/ny)}$  (where nx/ny can be
|         positive, negative, or zero); 0 otherwise.
|
|     -----
|     Data descriptors defined here:
|
|     __dict__
|         dictionary for instance variables
|
|     __weakref__
|         list of weak references to the object
|
class DiceEnterprise(builtins.object)
|     Implements the Dice Enterprise algorithm for
|     turning loaded dice with unknown probability of heads into loaded dice
|     with a different probability of heads. Specifically, it supports specifying
|     the probability that the output die will land on a given

```

number, as a polynomial function of the input die's probability of heads.  
 The case of coins to coins is also called  
 the Bernoulli factory problem; this class allows the output  
 coin's probability of heads to be specified as a polynomial function of the  
 input coin's probability of heads.

Reference: Morina, G., Łatuszyński, K., et al., "From the  
 Bernoulli Factory to a Dice Enterprise via Perfect  
 Sampling of Markov Chains", arXiv:1912.09229v1 [math.PR], 2019.

Example:

```
>>> from bernoulli import DiceEnterprise
>>> import math
>>> import random
>>>
>>> ent=DiceEnterprise()
>>> # Example 3 from the paper
>>> ent.append_poly(1,[[math.sqrt(2),3]])
>>> ent.append_poly(0,[[[-5,3],[11,2],[-9,1],[3,0]])
>>> coin=lambda: 1 if random.random() < 0.60 else 0
>>> print([ent.next(coin) for i in range(100)])
```

Methods defined here:

```
__init__(self)
    Initialize self. See help(type(self)) for accurate signature.

append_poly(self, result, poly)
    Appends a probability that the output die will land on
    a given number, in the form of a polynomial.
    result - A number indicating the result (die roll or coin
    flip) that will be returned by the _output_coin or _output_
    die with the probability represented by this polynomial.
    Must be an integer 0 or greater. In the case of dice-to-coins
    or coins-to-coins, must be either 0 or 1, where 1 means
    heads and 0 means tails.
    poly - Polynomial expressed as a list of terms as follows:
    Each term is a list of two or more items that each express one of
    the polynomial's terms; the first item is the coefficient, and
    the remaining items are the powers of the input die's
    probabilities. The number of remaining items in each term
    is the number of faces the _input_die has. Specifically, the
    term has the following form:

    In the case of coins-to-dice or coins-to-coins (so the probabilities are 1-p and p,
    where the [unknown] probability that the _input_coin returns 0
    is 1 - p, or returns 1 is p):
        term[0] * p**term[1] * (1-p)**term[2].
    In the case of dice-to-dice or dice-to-coins (so the probabilities are p1, p2, etc.,
    where the [unknown] probability that the _input_die returns
    0 is p1, returns 1 is p2, etc.):
```

```

|         term[0] * p1**term[1] * p2**term[2] * ... * pn**term[n].
|
|     For example, [3, 4, 5] becomes:
|         3 * p**4 * (1-p)**5
|     As a special case, the term can contain two items and a zero is
|     squeezed between the first and second item.
|     For example, [3, 4] is the same as [3, 0, 4], which in turn becomes:
|         3 * p**4 * (1-p)**0 = 3 * p **4
|
|     For best results, the coefficient should be a rational number
|     (such as int or Python's Fraction).
|
|     Each term in the polynomial must have the same number of items (except
|     for the special case given earlier). For example, the following is not a valid
|     way to express this parameter:
|         [[1, 1, 0], [1, 3, 4, 5], [1, 1, 2], [2, 3, 4]]
|     Here, the second term has four items, not three like the rest.
|     Returns this object.
|
| augment(self, count=1)
|     Augments the degree of the function represented
|     by this object, which can improve performance in some cases
|     (for details, see the paper).
|     - count: Number of times to augment the ladder.
|     Returns this object.
|
| next(self, coin)
|     Returns the next result of the flip from a coin or die
|     that is transformed from the specified input coin or die by the function
|     represented by this Dice Enterprise object.
|     coin - In the case of coins-to-dice or coins-to-coins (see the "append_poly" method),
|           this specifies the _input coin_, which must be a function that
|           returns either 1 (heads) or 0 (tails). In the case of dice-to-dice or dice-to-coins,
|           this specifies an _input die_ with _m_ faces, which must be a
|           function that returns an integer in the interval [0, m), which
|           specifies which face the input die lands on.
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables
|
| __weakref__
|     list of weak references to the object
|
class PolynomialSim(builtins.object)
|     PolynomialSim(bern, powerCoeffs, coin1)
|
|     Bernoulli factory for a polynomial expressed in power coefficients.
|     powerCoeffs[0] is the coefficient of order 0; powerCoeffs[1], of order 1, etc.
|

```

```

| Methods defined here:
|
| __init__(self, bern, powerCoeffs, coin1)
|     Initialize self.  See help(type(self)) for accurate signature.
|
| simulate(self)
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables
|
| __weakref__
|     list of weak references to the object

```

#### FUNCTIONS

```

    berncofrompower(coeffs)

    berncomatrix(deg)

    bernsteinDiff(coeffs, diff=1)

    matmult(mat, vec)

```

#### DATA

```

    BERNCOMATRIX = {}

```

#### FILE

```

    /home/peter/Documents/SharpDevelopProjects/peteroupc.github.io/bernoulli.py
problem in /home/peter/Documents/SharpDevelopProjects/peteroupc.github.io/betadist.py - SyntaxError: in
problem in /home/peter/Documents/SharpDevelopProjects/peteroupc.github.io/betadist.py - SyntaxError: in
problem in /home/peter/Documents/SharpDevelopProjects/peteroupc.github.io/betadist.py - SyntaxError: in

```