Sage Reference Manual: Quantitative Finance

Release 8.8

The Sage Development Team

Jun 27, 2019

CONTENTS

CHAPTER

ONE

TIME SERIES

This is a module for working with discrete floating point time series. It is designed so that every operation is very fast, typically much faster than with other generic code, e.g., Python lists of doubles or even NumPy arrays. The semantics of time series is more similar to Python lists of doubles than Sage real double vectors or NumPy 1-D arrays. In particular, time series are not endowed with much algebraic structure and are always mutable.

Note: NumPy arrays are faster at slicing, since slices return references, and NumPy arrays have strides. However, this speed at slicing makes NumPy slower at certain other operations.

EXAMPLES:

```
sage: set_random_seed(1)
sage: t = finance.TimeSeries([random()-0.5 for _ in range(10)]); t
[0.3294, 0.0959, -0.0706, -0.4646, 0.4311, 0.2275, -0.3840, -0.3528, -0.4119, -0.2933]
sage: t.sums()
[0.3294, 0.4253, 0.3547, -0.1099, 0.3212, 0.5487, 0.1647, -0.1882, -0.6001, -0.8933]
sage: t.exponential_moving_average(0.7)
[0.0000, 0.3294, 0.1660, 0.0003, -0.3251, 0.2042, 0.2205, -0.2027, -0.3078, -0.3807]
sage: t.standard_deviation()
0.33729638212891383
sage: t.mean()
-0.08933425506929439
sage: t.variance()
0.1137688493972542...
```

AUTHOR:

- William Stein

class sage.finance.time_series.**TimeSeries**

Bases: object

Initialize new time series.

INPUT:

- values – integer (number of values) or an iterable of floats.
- initialize – bool (default: True); if False, do not bother to zero out the entries of the new time series. For large series that you are going to just fill in, this can be way faster.

EXAMPLES:

This implicitly calls init:

```
sage: finance.TimeSeries([pi, 3, 18.2])
[3.1416, 3.0000, 18.2000]
```

Conversion from a NumPy 1-D array, which is very fast:

```
sage: v = finance.TimeSeries([1..5])
sage: w = v.numpy()
sage: finance.TimeSeries(w)
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000]
```

Conversion from an n-dimensional NumPy array also works:

```
sage: import numpy
sage: v = numpy.array([[1,2], [3,4]], dtype=float); v
array([[1., 2.],
       [3., 4.]])
sage: finance.TimeSeries(v)
[1.0000, 2.0000, 3.0000, 4.0000]
sage: finance.TimeSeries(v[:,0])
[1.0000, 3.0000]
sage: u = numpy.array([[1,2],[3,4]])
sage: finance.TimeSeries(u)
[1.0000, 2.0000, 3.0000, 4.0000]
```

For speed purposes we don't initialize (so value is garbage):

```
sage: t = finance.TimeSeries(10, initialize=False)
```

abs()

Return new time series got by replacing all entries of `self` by their absolute value.

OUTPUT:

A new time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,3.1908439,-4,5.93932])
sage: v
[1.0000, 3.1908, -4.0000, 5.9393]
sage: v.abs()
[1.0000, 3.1908, 4.0000, 5.9393]
```

add_entries(*t*)

Add corresponding entries of `self` and `t` together, extending either `self` or `t` by 0's if they do not have the same length.

Note: To add a single number to the entries of a time series, use the `add_scalar` method.

INPUT:

- `t` – a time series.

OUTPUT:

A time series with length the maxima of the lengths of `self` and `t`.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,2,-5]); v
[1.0000, 2.0000, -5.0000]
sage: v.add_entries([3,4])
[4.0000, 6.0000, -5.0000]
sage: v.add_entries(v)
[2.0000, 4.0000, -10.0000]
sage: v.add_entries([3,4,7,18.5])
[4.0000, 6.0000, 2.0000, 18.5000]
```

add_scalar(*s*)

Return new time series obtained by adding a scalar to every value in the series.

Note: To add componentwise, use the `add_entries` method.

INPUT:

- `s` – a float.

OUTPUT:

A new time series with `s` added to all values.

EXAMPLES:

```
sage: v = finance.TimeSeries([5,4,1.3,2,8,10,3,-5]); v
[5.0000, 4.0000, 1.3000, 2.0000, 8.0000, 10.0000, 3.0000, -5.0000]
sage: v.add_scalar(0.5)
[5.5000, 4.5000, 1.8000, 2.5000, 8.5000, 10.5000, 3.5000, -4.5000]
```

autocorrelation(*k=1*)

Return the k-th sample autocorrelation of this time series x_i.

Let μ be the sample mean. Then the sample autocorrelation function is

$$\frac{\sum_{t=0}^{n-k-1}(x_t-\mu)(x_{t+k}-\mu)}{\sum_{t=0}^{n-1}(x_t-\mu)^2}.$$

Note that the variance must be nonzero or you will get a `ZeroDivisionError`.

INPUT:

- `k` – a nonnegative integer (default: 1)

OUTPUT:

A time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([13,8,15,4,4,12,11,7,14,12])
sage: v.autocorrelation()
-0.1875
sage: v.autocorrelation(1)
-0.1875
sage: v.autocorrelation(0)
1.0
sage: v.autocorrelation(2)
-0.20138888888888887
sage: v.autocorrelation(3)
```

(continues on next page)

(continued from previous page)

```
0.18055555555555555

sage: finance.TimeSeries([1..1000]).autocorrelation()
0.997
```

autocovariance (*k=0*)

Return the k-th autocovariance function $\gamma(k)$ of `self`. This is the covariance of `self` with `self` shifted by k, i.e.,

$$\left(\sum_{t=0}^{n-k-1}(x_t-\mu)(x_{t+k}-\mu)\right)\Big/ n,$$

where n is the length of `self`.

Note the denominator of n, which gives a "better" sample estimator.

INPUT:

- `k` – a nonnegative integer (default: 0)

OUTPUT:

A float.

EXAMPLES:

```
sage: v = finance.TimeSeries([13,8,15,4,4,12,11,7,14,12])
sage: v.autocovariance(0)
14.4
sage: mu = v.mean(); sum([(a-mu)^2 for a in v])/len(v)
14.4
sage: v.autocovariance(1)
-2.7
sage: mu = v.mean(); sum([(v[i]-mu)*(v[i+1]-mu) for i in range(len(v)-1)])/
↪len(v)
-2.7
sage: v.autocovariance(1)
-2.7
```

We illustrate with a random sample that an independently and identically distributed distribution with zero mean and variance σ^2 has autocovariance function $\gamma(h)$ with $\gamma(0)=\sigma^2$ and $\gamma(h)=0$ for $h\neq 0$.

```
sage: set_random_seed(0)
sage: v = finance.TimeSeries(10^6)
sage: v.randomize('normal', 0, 5)
[3.3835, -2.0055, 1.7882, -2.9319, -4.6827 ... -5.1868, 9.2613, 0.9274, -6.
↪2282, -8.7652]
sage: v.autocovariance(0)
24.95410689...
sage: v.autocovariance(1)
-0.00508390...
sage: v.autocovariance(2)
0.022056325...
sage: v.autocovariance(3)
-0.01902000...
```

autoregressive_fit (*M*)

This method fits the time series to an autoregressive process of order `M`. That is, we assume the process is

given by $X_t - \mu = a_1(X_{t-1} - \mu) + a_2(X_{t-1} - \mu) + \cdots + a_M(X_{t-M} - \mu) + Z_t$ where μ is the mean of the process and Z_t is noise. This method returns estimates for $a_1, \ldots, a_M$.

The method works by solving the Yule-Walker equations $\Gamma a = \gamma$, where $\gamma = (\gamma(1), \ldots, \gamma(M))$, $a = (a_1, \ldots, a_M)$ with $\gamma(i)$ the autocovariance of lag i and $\Gamma_{ij} = \gamma(i - j)$.

Warning: The input sequence is assumed to be stationary, which means that the autocovariance $\langle y_j y_k \rangle$ depends only on the difference $|j - k|$.

INPUT:

- `M` – an integer.

OUTPUT:

A time series – the coefficients of the autoregressive process.

EXAMPLES:

```
sage: set_random_seed(0)
sage: v = finance.TimeSeries(10^4).randomize('normal').sums()
sage: F = v.autoregressive_fit(100)
sage: v
[0.6767, 0.2756, 0.6332, 0.0469, -0.8897 ... 87.6759, 87.6825, 87.4120, 87.
↪6639, 86.3194]
sage: v.autoregressive_forecast(F)
86.0177285042...
sage: F
[1.0148, -0.0029, -0.0105, 0.0067, -0.0232 ... -0.0106, -0.0068, 0.0085, -0.
↪0131, 0.0092]

sage: set_random_seed(0)
sage: t=finance.TimeSeries(2000)
sage: z=finance.TimeSeries(2000)
sage: z.randomize('normal',1)
[1.6767, 0.5989, 1.3576, 0.4136, 0.0635 ... 1.0057, -1.1467, 1.2809, 1.5705,␣
↪1.1095]
sage: t[0]=1
sage: t[1]=2
sage: for i in range(2,2000):
....:     t[i]=t[i-1]-0.5*t[i-2]+z[i]
sage: c=t[0:-1].autoregressive_fit(2)  #recovers recurrence relation
sage: c #should be close to [1,-0.5]
[1.0371, -0.5199]
```

autoregressive_forecast (*filter*)

Given the autoregression coefficients as outputted by the `autoregressive_fit` command, compute the forecast for the next term in the series.

INPUT:

- `filter` – a time series outputted by the `autoregressive_fit` command.

EXAMPLES:

```
sage: set_random_seed(0)
sage: v = finance.TimeSeries(100).randomize('normal').sums()
sage: F = v[:-1].autoregressive_fit(5); F
```

(continues on next page)

(continued from previous page)

```
[1.0019, -0.0524, -0.0643, 0.1323, -0.0539]
sage: v.autoregressive_forecast(F)
11.7820298611...
sage: v
[0.6767, 0.2756, 0.6332, 0.0469, -0.8897 ... 9.2447, 9.6709, 10.4037, 10.4836,
↪ 12.1960]
```

central_moment(*k*)

Return the k-th central moment of `self`, which is just the mean of the k-th powers of the differences `self[i] - mu`, where `mu` is the mean of `self`.

INPUT:

- `k` – a positive integer.

OUTPUT:

A double.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,2,3])
sage: v.central_moment(2)
0.666666666666666
```

Note that the central moment is different from the moment here, since the mean is not 0:

```
sage: v.moment(2)      # doesn't subtract off mean
4.666666666666667
```

We compute the central moment directly:

```
sage: mu = v.mean(); mu
2.0
sage: ((1-mu)^2 + (2-mu)^2 + (3-mu)^2) / 3
0.666666666666666
```

clip_remove(*min=None*, *max=None*)

Return new time series obtained from `self` by removing all values that are less than or equal to a certain minimum value or greater than or equal to a certain maximum.

INPUT:

- `min` – (default: `None`) `None` or double.
- `max` – (default: `None`) `None` or double.

OUTPUT:

A time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([1..10])
sage: v.clip_remove(3,7)
[3.0000, 4.0000, 5.0000, 6.0000, 7.0000]
sage: v.clip_remove(3)
[3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000, 9.0000, 10.0000]
sage: v.clip_remove(max=7)
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000]
```

correlation(*other*)

Return the correlation of `self` and `other`, which is the covariance of `self` and `other` divided by the product of their standard deviation.

INPUT:

- `self`, `other` – time series.

Whichever time series has more terms is truncated.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,-2,3]); w = finance.TimeSeries([4,5,-10])
sage: v.correlation(w)
-0.558041609...
sage: v.covariance(w)/(v.standard_deviation() * w.standard_deviation())
-0.558041609...
```

covariance(*other*)

Return the covariance of the time series `self` and `other`.

INPUT:

- `self`, `other` – time series.

Whichever time series has more terms is truncated.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,-2,3]); w = finance.TimeSeries([4,5,-10])
sage: v.covariance(w)
-11.777777777777779
```

diffs(*k=1*)

Return the new time series got by taking the differences of successive terms in the time series. So if `self` is the time series $X_0, X_1, X_2, \ldots$, then this function outputs the series $X_1 - X_0, X_2 - X_1, \ldots$. The output series has one less term than the input series. If the optional parameter k is given, return $X_k - X_0, X_{2k} - X_k, \ldots$.

INPUT:

- `k` – positive integer (default: 1)

OUTPUT:

A new time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([5,4,1.3,2,8]); v
[5.0000, 4.0000, 1.3000, 2.0000, 8.0000]
sage: v.diffs()
[-1.0000, -2.7000, 0.7000, 6.0000]
```

exp()

Return new time series got by applying the exponential map to all the terms in the time series.

OUTPUT:

A new time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([1..5]); v
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000]
sage: v.exp()
[2.7183, 7.3891, 20.0855, 54.5982, 148.4132]
sage: v.exp().log()
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000]
```

exponential_moving_average(*alpha*)

Return the exponential moving average time series. Assumes the input time series was constant with its starting value for negative time. The t-th step of the output is the sum of the previous k-1 steps of `self` and the k-th step divided by k.

The 0-th term is formally undefined, so we define it to be 0, and we define the first term to be `self[0]`.

INPUT:

- `alpha` – float; a smoothing factor with `0 <= alpha <= 1`.

OUTPUT:

A time series with the same number of steps as `self`.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.exponential_moving_average(0)
[0.0000, 1.0000, 1.0000, 1.0000, 1.0000]
sage: v.exponential_moving_average(1)
[0.0000, 1.0000, 1.0000, 1.0000, 2.0000]
sage: v.exponential_moving_average(0.5)
[0.0000, 1.0000, 1.0000, 1.0000, 1.5000]
```

Some more examples:

```
sage: v = finance.TimeSeries([1,2,3,4,5])
sage: v.exponential_moving_average(1)
[0.0000, 1.0000, 2.0000, 3.0000, 4.0000]
sage: v.exponential_moving_average(0)
[0.0000, 1.0000, 1.0000, 1.0000, 1.0000]
```

extend(*right*)

Extend this time series by appending elements from the iterable `right`.

INPUT:

- `right` – iterable that can be converted to a time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,2,-5]); v
[1.0000, 2.0000, -5.0000]
sage: v.extend([-3.5, 2])
sage: v
[1.0000, 2.0000, -5.0000, -3.5000, 2.0000]
```

fft(*overwrite=False*)

Return the real discrete fast Fourier transform of `self`, as a real time series:

$$[y(0), \Re(y(1)), \Im(y(1)), \ldots, \Re(y(n/2)), \Im(y(n/2))] \text{ if } n \text{ is even}$$
$$[y(0), \Re(y(1)), \Im(y(1)), \ldots, \Re(y(n/2)), \Im(y(n/2))] \text{ if } n \text{ is odd}$$

where

$$y(j) = \sum_{k=0}^{n-1} x[k] \cdot \exp(-\sqrt{-1} \cdot jk \cdot 2\pi/n)$$

for $j = 0, \ldots, n-1$. Note that $y(-j) = y(n-j)$.

EXAMPLES:

```
sage: v = finance.TimeSeries([1..9]); v
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000, 9.0000]
sage: w = v.fft(); w
[45.0000, -4.5000, 12.3636, -4.5000, 5.3629, -4.5000, 2.5981, -4.5000, 0.7935]
```

We get just the series of real parts of

```
sage: finance.TimeSeries([w[0]]) + w[1:].scale_time(2)
[45.0000, -4.5000, -4.5000, -4.5000, -4.5000]
```

An example with an even number of terms:

```
sage: v = finance.TimeSeries([1..10]); v
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000, 9.0000, 10.
↪0000]
sage: w = v.fft(); w
[55.0000, -5.0000, 15.3884, -5.0000, 6.8819, -5.0000, 3.6327, -5.0000, 1.6246,
↪ -5.0000]
sage: v.fft().ifft()
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000, 9.0000, 10.
↪0000]
```

histogram(*bins=50*, *normalize=False*)

Return the frequency histogram of the values in this time series divided up into the given number of bins.

INPUT:

- `bins` – a positive integer (default: 50)
- `normalize` – (default: `False`) whether to normalize so the total area in the bars of the histogram is 1.

OUTPUT:

- counts – list of counts of numbers of elements in each bin.
- endpoints – list of 2-tuples (a,b) that give the endpoints of the bins.

EXAMPLES:

```
sage: v = finance.TimeSeries([5,4,1.3,2,8,10,3,-5]); v
[5.0000, 4.0000, 1.3000, 2.0000, 8.0000, 10.0000, 3.0000, -5.0000]
sage: v.histogram(3)
([1, 4, 3], [(-5.0, 0.0), (0.0, 5.0), (5.0, 10.0)])
```

hurst_exponent()

Returns an estimate of the Hurst exponent of this time series. We use the algorithm from pages 61 – 63 of [Peteres, Fractal Market Analysis (1994); see Google Books].

We define the Hurst exponent of a constant time series to be 1.

EXAMPLES:

The Hurst exponent of Brownian motion is 1/2. We approximate it with some specific samples. Note that the estimator is biased and slightly overestimates.

```
sage: set_random_seed(0)
sage: bm = finance.TimeSeries(10^5).randomize('normal').sums(); bm
[0.6767, 0.2756, 0.6332, 0.0469, -0.8897 ... 152.2437, 151.5327, 152.7629,␣
↪152.9169, 152.9084]
sage: bm.hurst_exponent()
0.527450972...
```

We compute the Hurst exponent of a simulated fractional Brownian motion with Hurst parameter 0.7. This function estimates the Hurst exponent as 0.706511951...

```
sage: set_random_seed(0)
sage: fbm = finance.fractional_brownian_motion_simulation(0.7,0.1,10^5,1)[0]
sage: fbm.hurst_exponent()
0.706511951...
```

Another example with small Hurst exponent (notice the overestimation).

```
sage: fbm = finance.fractional_brownian_motion_simulation(0.2,0.1,10^5,1)[0]
sage: fbm.hurst_exponent()
0.278997441...
```

We compute the mean Hurst exponent of 100 simulated multifractal cascade random walks:

```
sage: set_random_seed(0)
sage: y = finance.multifractal_cascade_random_walk_simulation(3700,0.02,0.01,
↪0.01,1000,100)
sage: finance.TimeSeries([z.hurst_exponent() for z in y]).mean()
0.57984822577934...
```

We compute the mean Hurst exponent of 100 simulated Markov switching multifractal time series. The Hurst exponent is quite small.

```
sage: set_random_seed(0)
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,0.5,0.95,3)
sage: y = msm.simulations(1000,100)
sage: finance.TimeSeries([z.hurst_exponent() for z in y]).mean()
0.286102325623705...
```

ifft(*overwrite=False*)

Return the real discrete inverse fast Fourier transform of `self`, which is also a real time series.

This is the inverse of `fft()`.

The returned real array contains

$$[y(0), y(1), \ldots, y(n-1)]$$

where for n is even

$$y(j) = 1/n \left(\sum_{k=1}^{n/2-1} (x[2k-1] + \sqrt{-1} \cdot x[2k]) \cdot \exp(\sqrt{-1} \cdot jk \cdot 2pi/n) + c.c. + x[0] + (-1)^j x[n-1] \right)$$

and for n is odd

$$y(j) = 1/n \left(\sum_{k=1}^{(n-1)/2} (x[2k-1] + \sqrt{-1} \cdot x[2k]) \cdot \exp(\sqrt{-1} \cdot jk \cdot 2pi/n) + c.c. + x[0] \right)$$

where *c.c.* denotes complex conjugate of preceding expression.

EXAMPLES:

```
sage: v = finance.TimeSeries([1..10]); v
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000, 9.0000, 10.
↪0000]
sage: v.ifft()
[5.1000, -5.6876, 1.4764, -1.0774, 0.4249, -0.1000, -0.2249, 0.6663, -1.2764,␣
↪1.6988]
sage: v.ifft().fft()
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000, 9.0000, 10.
↪0000]
```

list()

Return list of elements of `self`.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,-4,3,-2.5,-4,3])
sage: v.list()
[1.0, -4.0, 3.0, -2.5, -4.0, 3.0]
```

log()

Return new time series got by taking the logarithms of all the terms in the time series.

OUTPUT:

A new time series.

EXAMPLES:

We exponentiate then log a time series and get back the original series:

```
sage: v = finance.TimeSeries([1,-4,3,-2.5,-4,3]); v
[1.0000, -4.0000, 3.0000, -2.5000, -4.0000, 3.0000]
sage: v.exp()
[2.7183, 0.0183, 20.0855, 0.0821, 0.0183, 20.0855]
sage: v.exp().log()
[1.0000, -4.0000, 3.0000, -2.5000, -4.0000, 3.0000]
```

Log of 0 gives `-inf`:

```
sage: finance.TimeSeries([1,0,3]).log()[1]
-inf
```

max(*index=False*)

Return the largest value in this time series. If this series has length 0 we raise a `ValueError`.

INPUT:

- `index` – bool (default: `False`); if `True`, also return index of maximum entry.

OUTPUT:

- float – largest value.
- integer – index of largest value; only returned if `index=True`.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,-4,3,-2.5,-4,3])
sage: v.max()
3.0
sage: v.max(index=True)
(3.0, 2)
```

mean()

Return the mean (average) of the elements of `self`.

OUTPUT:

A double.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.mean()
1.6
```

min(*index=False*)

Return the smallest value in this time series. If this series has length 0 we raise a `ValueError`.

INPUT:

- `index` – bool (default: `False`); if `True`, also return index of minimal entry.

OUTPUT:

- float – smallest value.
- integer – index of smallest value; only returned if `index=True`.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,-4,3,-2.5,-4])
sage: v.min()
-4.0
sage: v.min(index=True)
(-4.0, 1)
```

moment(*k*)

Return the k-th moment of `self`, which is just the mean of the k-th powers of the elements of `self`.

INPUT:

- `k` – a positive integer.

OUTPUT:

A double.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.moment(1)
1.6
sage: v.moment(2)
3.2
```

numpy(*copy=True*)

Return a NumPy version of this time series.

Note: If copy is `False`, return a NumPy 1-D array reference to exactly the same block of memory as this time series. This is very, very fast and makes it easy to quickly use all NumPy/SciPy functionality on `self`. However, it is dangerous because when this time series goes out of scope and is garbage collected, the corresponding NumPy reference object will point to garbage.

INPUT:

- `copy` – bool (default: `True`)

OUTPUT:

A numpy 1-D array.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,-3,4.5,-2])
sage: w = v.numpy(copy=False); w
array([ 1. , -3. ,  4.5, -2. ])
sage: type(w)
<... 'numpy.ndarray'>
sage: w.shape
(4,)
```

Notice that changing `w` also changes `v` too!

```
sage: w[0] = 20
sage: w
array([20. , -3. ,  4.5, -2. ])
sage: v
[20.0000, -3.0000, 4.5000, -2.0000]
```

If you want a separate copy do not give the `copy=False` option.

```
sage: z = v.numpy(); z
array([20. , -3. ,  4.5, -2. ])
sage: z[0] = -10
sage: v
[20.0000, -3.0000, 4.5000, -2.0000]
```

plot (*plot_points=1000*, *points=False*, ***kwds*)

Return a plot of this time series as a line or points through $(i, T(i))$, where i ranges over nonnegative integers up to the length of `self`.

INPUT:

- `plot_points` – (default: 1000) 0 or positive integer. Only plot the given number of equally spaced points in the time series. If 0, plot all points.
- `points` – bool (default: `False`). If `True`, return just the points of the time series.
- `**kwds` – passed to the line or point command.

EXAMPLES:

```
sage: v = finance.TimeSeries([5,4,1.3,2,8,10,3,-5]); v
[5.0000, 4.0000, 1.3000, 2.0000, 8.0000, 10.0000, 3.0000, -5.0000]
sage: v.plot()
Graphics object consisting of 1 graphics primitive
sage: v.plot(points=True)
```

(continues on next page)

(continued from previous page)

```
Graphics object consisting of 1 graphics primitive
sage: v.plot() + v.plot(points=True, rgbcolor='red')
Graphics object consisting of 2 graphics primitives
sage: v.plot() + v.plot(points=True, rgbcolor='red', pointsize=50)
Graphics object consisting of 2 graphics primitives
```

plot_candlestick(*bins=30*)

Return a candlestick plot of this time series with the given number of bins.

A candlestick plot is a style of bar-chart used to view open, high, low, and close stock data. At each bin, the line represents the high / low range. The bar represents the open / close range. The interval is colored blue if the open for that bin is less than the close. If the close is less than the open, then that bin is colored red instead.

INPUT:

- `bins` – positive integer (default: 30), the number of bins or candles.

OUTPUT:

A candlestick plot.

EXAMPLES:

Here we look at the candlestick plot for Brownian motion:

```
sage: v = finance.TimeSeries(1000).randomize()
sage: v.plot_candlestick(bins=20)
Graphics object consisting of 40 graphics primitives
```

plot_histogram(*bins=50*, *normalize=True*, ***kwds*)

Return histogram plot of this time series with given number of bins.

INPUT:

- `bins` – positive integer (default: 50)
- `normalize` – (default: `True`) whether to normalize so the total area in the bars of the histogram is 1.

OUTPUT:

A histogram plot.

EXAMPLES:

```
sage: v = finance.TimeSeries([1..50])
sage: v.plot_histogram(bins=10)
Graphics object consisting of 10 graphics primitives
```

```
sage: v.plot_histogram(bins=3,normalize=False,aspect_ratio=1)
Graphics object consisting of 3 graphics primitives
```

pow(*k*)

Return a new time series with every elements of `self` raised to the k-th power.

INPUT:

- `k` – a float.

OUTPUT:

A time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.pow(2)
[1.0000, 1.0000, 1.0000, 4.0000, 9.0000]
```

prod()

Return the product of all the entries of `self`. If `self` has length 0, returns 1.

OUTPUT:

A double.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.prod()
6.0
```

randomize(*distribution='uniform'*, *loc=0*, *scale=1*, ***kwds*)

Randomize the entries in this time series, and return a reference to `self`. Thus this function both changes `self` in place, and returns a copy of `self`, for convenience.

INPUT:

- `distribution` – (default: `"uniform"`); supported values are:
  - `'uniform'` – from `loc` to `loc + scale`
  - `'normal'` – mean `loc` and standard deviation `scale`
  - `'semicircle'` – with center at `loc` (`scale` is ignored)
  - `'lognormal'` – mean `loc` and standard deviation `scale`
- `loc` – float (default: 0)
- `scale` – float (default: 1)

Note: All random numbers are generated using algorithms that build on the high quality GMP random number function gmp_urandomb_ui. Thus this function fully respects the Sage `set_random_state` command. It's not quite as fast as the C library random number generator, but is of course much better quality, and is platform independent.

EXAMPLES:

We generate 5 uniform random numbers in the interval [0,1]:

```
sage: set_random_seed(0)
sage: finance.TimeSeries(5).randomize()
[0.8685, 0.2816, 0.0229, 0.1456, 0.7314]
```

We generate 5 uniform random numbers from 5 to $5 + 2 = 7$:

```
sage: set_random_seed(0)
sage: finance.TimeSeries(10).randomize('uniform', 5, 2)
[6.7369, 5.5632, 5.0459, 5.2911, 6.4628, 5.2412, 5.2010, 5.2761, 5.5813, 5.
↪5439]
```

(continues on next page)

(continued from previous page)

We generate 5 normal random values with mean 0 and variance 1.

```
sage: set_random_seed(0)
sage: finance.TimeSeries(5).randomize('normal')
[0.6767, -0.4011, 0.3576, -0.5864, -0.9365]
```

We generate 10 normal random values with mean 5 and variance 2.

```
sage: set_random_seed(0)
sage: finance.TimeSeries(10).randomize('normal', 5, 2)
[6.3534, 4.1978, 5.7153, 3.8273, 3.1269, 2.9598, 3.7484, 6.7472, 3.8986, 4.
↪6271]
```

We generate 5 values using the semicircle distribution.

```
sage: set_random_seed(0)
sage: finance.TimeSeries(5).randomize('semicircle')
[0.7369, -0.9541, 0.4628, -0.7990, -0.4187]
```

We generate 1 million normal random values and create a frequency histogram.

```
sage: set_random_seed(0)
sage: a = finance.TimeSeries(10^6).randomize('normal')
sage: a.histogram(10)[0]
[36, 1148, 19604, 130699, 340054, 347870, 137953, 21290, 1311, 35]
```

We take the above values, and compute the proportion that lie within 1, 2, 3, 5, and 6 standard deviations of the mean (0):

```
sage: s = a.standard_deviation()
sage: len(a.clip_remove(-s,s))/float(len(a))
0.683094
sage: len(a.clip_remove(-2*s,2*s))/float(len(a))
0.954559
sage: len(a.clip_remove(-3*s,3*s))/float(len(a))
0.997228
sage: len(a.clip_remove(-5*s,5*s))/float(len(a))
0.999998
```

There were no "six sigma events":

```
sage: len(a.clip_remove(-6*s,6*s))/float(len(a))
1.0
```

range_statistic(*b=None*)

Return the rescaled range statistic R/S of `self`, which is defined as follows (see Hurst 1951). If the optional parameter `b` is given, return the average of R/S range statistics of disjoint blocks of size `b`.

Let σ be the standard deviation of the sequence of differences of `self`, and let Y_k be the k-th term of `self`. Let n be the number of terms of `self`, and set $Z_k = Y_k - ((k+1)/n) \cdot Y_n$. Then

$$R/S = \big(\max(Z_k) - \min(Z_k)\big)/\sigma$$

where the max and min are over all Z_k. Basically replacing Y_k by Z_k allows us to measure the difference from the line from the origin to (n, Y_n).

INPUT:

- `self` – a time series (*not* the series of differences).
- `b` – integer (default: `None`); if given instead divide the input time series up into `j = floor(n/b)` disjoint blocks, compute the R/S statistic for each block, and return the average of those R/S statistics.

OUTPUT:

A float.

EXAMPLES:

Notice that if we make a Brownian motion random walk, there is no difference if we change the standard deviation.

```
sage: set_random_seed(0); finance.TimeSeries(10^6).randomize('normal').sums().
↪range_statistic()
1897.8392602...
sage: set_random_seed(0); finance.TimeSeries(10^6).randomize('normal',0,100).
↪sums().range_statistic()
1897.8392602...
```

rescale(*s*)

Change `self` by multiplying every value in the series by `s`.

INPUT:

- `s` – a float.

EXAMPLES:

```
sage: v = finance.TimeSeries([5,4,1.3,2,8,10,3,-5]); v
[5.0000, 4.0000, 1.3000, 2.0000, 8.0000, 10.0000, 3.0000, -5.0000]
sage: v.rescale(0.5)
sage: v
[2.5000, 2.0000, 0.6500, 1.0000, 4.0000, 5.0000, 1.5000, -2.5000]
```

reversed()

Return new time series obtain from this time series by reversing the order of the entries in this time series.

OUTPUT:

A time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([1..5])
sage: v.reversed()
[5.0000, 4.0000, 3.0000, 2.0000, 1.0000]
```

scale(*s*)

Return new time series obtained by multiplying every value in the series by `s`.

INPUT:

- `s` – a float.

OUTPUT:

A new time series with all values multiplied by `s`.

EXAMPLES:

```
sage: v = finance.TimeSeries([5,4,1.3,2,8,10,3,-5]); v
[5.0000, 4.0000, 1.3000, 2.0000, 8.0000, 10.0000, 3.0000, -5.0000]
sage: v.scale(0.5)
[2.5000, 2.0000, 0.6500, 1.0000, 4.0000, 5.0000, 1.5000, -2.5000]
```

scale_time(*k*)

Return the new time series at scale `k`. If the input time series is $X_0, X_1, X_2, \ldots$, then this function returns the shorter time series $X_0, X_k, X_{2k}, \ldots$.

INPUT:

- `k` – a positive integer.

OUTPUT:

A new time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([5,4,1.3,2,8,10,3,-5]); v
[5.0000, 4.0000, 1.3000, 2.0000, 8.0000, 10.0000, 3.0000, -5.0000]
sage: v.scale_time(1)
[5.0000, 4.0000, 1.3000, 2.0000, 8.0000, 10.0000, 3.0000, -5.0000]
sage: v.scale_time(2)
[5.0000, 1.3000, 8.0000, 3.0000]
sage: v.scale_time(3)
[5.0000, 2.0000]
sage: v.scale_time(10)
[]
```

A series of odd length:

```
sage: v = finance.TimeSeries([1..5]); v
[1.0000, 2.0000, 3.0000, 4.0000, 5.0000]
sage: v.scale_time(2)
[1.0000, 3.0000, 5.0000]
```

show(**args*, ***kwds*)

Calls plot and passes all arguments onto the plot function. This is thus just an alias for plot.

EXAMPLES:

Draw a plot of a time series:

```
sage: finance.TimeSeries([1..10]).show()
Graphics object consisting of 1 graphics primitive
```

simple_moving_average(*k*)

Return the moving average time series over the last `k` time units. Assumes the input time series was constant with its starting value for negative time. The t-th step of the output is the sum of the previous `k - 1` steps of `self` and the `k`-th step divided by `k`. Thus `k` values are averaged at each point.

INPUT:

- `k` – positive integer.

OUTPUT:

A time series with the same number of steps as `self`.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.simple_moving_average(0)
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.simple_moving_average(1)
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.simple_moving_average(2)
[1.0000, 1.0000, 1.0000, 1.5000, 2.5000]
sage: v.simple_moving_average(3)
[1.0000, 1.0000, 1.0000, 1.3333, 2.0000]
```

standard_deviation(*bias=False*)

Return the standard deviation of the entries of `self`.

INPUT:

- `bias` – bool (default: `False`); if `False`, divide by `self.length() - 1` instead of `self.length()` to give a less biased estimator for the variance.

OUTPUT:

A double.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.standard_deviation()
0.8944271909...
sage: v.standard_deviation(bias=True)
0.8
```

sum()

Return the sum of all the entries of `self`. If `self` has length 0, returns 0.

OUTPUT:

A double.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.sum()
8.0
```

sums(*s=0*)

Return the new time series got by taking the running partial sums of the terms of this time series.

INPUT:

- `s` – starting value for partial sums.

OUTPUT:

A time series.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
```

(continues on next page)

(continued from previous page)

```
sage: v.sums()
[1.0000, 2.0000, 3.0000, 5.0000, 8.0000]
```

variance(*bias=False*)

Return the variance of the elements of `self`, which is the mean of the squares of the differences from the mean.

INPUT:

- `bias` – bool (default: `False`); if `False`, divide by `self.length() - 1` instead of `self.length()` to give a less biased estimator for the variance.

OUTPUT:

A double.

EXAMPLES:

```
sage: v = finance.TimeSeries([1,1,1,2,3]); v
[1.0000, 1.0000, 1.0000, 2.0000, 3.0000]
sage: v.variance()
0.8
sage: v.variance(bias=True)
0.64
```

vector()

Return real double vector whose entries are the values of this time series. This is useful since vectors have standard algebraic structure and play well with matrices.

OUTPUT:

A real double vector.

EXAMPLES:

```
sage: v = finance.TimeSeries([1..10])
sage: v.vector()
(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0)
```

sage.finance.time_series.**autoregressive_fit**(*acvs*)

Given a sequence of lagged autocovariances of length M produce $a_1, \ldots, a_p$ so that the first M autocovariance coefficients of the autoregressive processes $X_t = a_1 X_{t_1} + \cdots + a_p X_{t-p} + Z_t$ are the same as the input sequence.

The function works by solving the Yule-Walker equations $\Gamma a = \gamma$, where $\gamma = (\gamma(1), \ldots, \gamma(M))$, $a = (a_1, \ldots, a_M)$, with $\gamma(i)$ the autocovariance of lag i and $\Gamma_{ij} = \gamma(i-j)$.

EXAMPLES:

In this example we consider the multifractal cascade random walk of length 1000, and use simulations to estimate the expected first few autocovariance parameters for this model, then use them to construct a linear filter that works vastly better than a linear filter constructed from the same data but not using this model. The Monte-Carlo method illustrated below should work for predicting one "time step" into the future for any model that can be simulated. To predict k time steps into the future would require using a similar technique but would require scaling time by k.

We create 100 simulations of a multifractal random walk. This models the logarithms of a stock price sequence.

```
sage: set_random_seed(0)
sage: y = finance.multifractal_cascade_random_walk_simulation(3700,0.02,0.01,0.01,
↪1000,100)
```

For each walk below we replace the walk by the walk but where each step size is replaced by its absolute value – this is what we expect to be able to predict given the model, which is only a model for predicting volatility. We compute the first 200 autocovariance values for every random walk:

```
sage: c = [[a.diffs().abs().sums().autocovariance(i) for a in y] for i in␣
↪range(200)]
```

We make a time series out of the expected values of the autocovariances:

```
sage: ac = finance.TimeSeries([finance.TimeSeries(z).mean() for z in c])
sage: ac
[3.9962, 3.9842, 3.9722, 3.9601, 3.9481 ... 1.7144, 1.7033, 1.6922, 1.6812, 1.
↪6701]
```

Note: `ac` looks like a line – one could best fit it to yield a lot more approximate autocovariances.

We compute the autoregression coefficients matching the above autocovariances:

```
sage: F = finance.autoregressive_fit(ac); F
[0.9982, -0.0002, -0.0002, 0.0003, 0.0001 ... 0.0002, -0.0002, -0.0000, -0.0002, -
↪0.0014]
```

Note that the sum is close to 1:

```
sage: sum(F)
0.99593284089454...
```

Now we make up an 'out of sample' sequence:

```
sage: y2 = finance.multifractal_cascade_random_walk_simulation(3700,0.02,0.01,0.
↪01,1000,1)[0].diffs().abs().sums()
sage: y2
[0.0013, 0.0059, 0.0066, 0.0068, 0.0184 ... 6.8004, 6.8009, 6.8063, 6.8090, 6.
↪8339]
```

And we forecast the very last value using our linear filter; the forecast is close:

```
sage: y2[:-1].autoregressive_forecast(F)
6.7836741372407...
```

In fact it is closer than we would get by forecasting using a linear filter made from all the autocovariances of our sequence:

```
sage: y2[:-1].autoregressive_forecast(y2[:-1].autoregressive_fit(len(y2)))
6.770168705668...
```

We record the last 20 forecasts, always using all correct values up to the one we are forecasting:

```
sage: s1 = sum([(y2[:-i].autoregressive_forecast(F)-y2[-i])^2 for i in range(1,
↪20)])
```

We do the same, but using the autocovariances of the sample sequence:

```
sage: F2 = y2[:-100].autoregressive_fit(len(F))
sage: s2 = sum([(y2[:-i].autoregressive_forecast(F2)-y2[-i])^2 for i in range(1,
↪20)])
```

Our model gives us something that is 15 percent better in this case:

```
sage: s2/s1
1.15464636102...
```

How does it compare overall? To find out we do 100 simulations and for each we compute the percent that our model beats naively using the autocovariances of the sample:

```
sage: y_out = finance.multifractal_cascade_random_walk_simulation(3700,0.02,0.01,
↪0.01,1000,100)
sage: s1 = []; s2 = []
sage: for v in y_out:
....:     s1.append(sum([(v[:-i].autoregressive_forecast(F)-v[-i])^2 for i in␣
↪range(1,20)]))
....:     F2 = v[:-len(F)].autoregressive_fit(len(F))
....:     s2.append(sum([(v[:-i].autoregressive_forecast(F2)-v[-i])^2 for i in␣
↪range(1,20)]))
```

We find that overall the model beats naive linear forecasting by 35 percent!

```
sage: s = finance.TimeSeries([s2[i]/s1[i] for i in range(len(s1))])
sage: s.mean()
1.354073591877...
```

sage.finance.time_series.**unpickle_time_series_v1**(*v*, *n*)

Version 1 unpickle method.

INPUT:

- v – a raw char buffer

EXAMPLES:

```
sage: v = finance.TimeSeries([1,2,3])
sage: s = v.__reduce__()[1][0]
sage: type(s)  # py2
<type 'str'>
sage: type(s)  # py3
<type 'bytes'>
sage: sage.finance.time_series.unpickle_time_series_v1(s,3)
[1.0000, 2.0000, 3.0000]
sage: sage.finance.time_series.unpickle_time_series_v1(s+s,6)
[1.0000, 2.0000, 3.0000, 1.0000, 2.0000, 3.0000]
sage: sage.finance.time_series.unpickle_time_series_v1(b'',0)
[]
```

CHAPTER

TWO

STOCK MARKET PRICE SERIES

This module's main class is *Stock*. It defines the following methods:

market_value()	Return the current market value of this stock.
current_price_data()	Get Yahoo current price data for this stock.
history()	Return an immutable sequence of historical price data for this stock
open()	Return a time series containing historical opening prices for this stock.
close()	Return the time series of all historical closing prices for this stock.
load_from_file()	Load historical data from a local csv formatted data file.

Warning: The *Stock* class is currently broken due to the change in the Yahoo interface. See trac ticket #25473.

AUTHORS:

- William Stein, 2008
- Brett Nakayama, 2008
- Chris Swierczewski, 2008

2.1 Classes and methods

class sage.finance.stock.**OHLC**(*timestamp*, *open*, *high*, *low*, *close*, *volume*)

Open, high, low, and close information for a stock. Also stores a timestamp for that data along with the volume.

INPUT:

- `timestamp` – string
- `open`, `high`, `low`, `close` – float
- `volume` – int

EXAMPLES:

```
sage: from sage.finance.stock import OHLC
sage: OHLC('18-Aug-04', 100.01, 104.06, 95.96, 100.34, 22353092)
 18-Aug-04 100.01 104.06 95.96 100.34   22353092
```

class sage.finance.stock.**Stock**(*symbol*, *cid=''*)

Class for retrieval of stock market information.

close(**args*, ***kwds*)

Return the time series of all historical closing prices for this stock. If no arguments are given, will return last acquired historical data. Otherwise, data will be gotten from Google Finance.

INPUT:

- `startdate` – string, (default: `'Jan+1,+1900'`)
- `enddate` – string, (default: current date)
- `histperiod` – string, (`'daily'` or `'weekly'`)

OUTPUT:

A time series – close price data.

EXAMPLES:

You can directly obtain close data as so:

```
sage: finance.Stock('vmw').close(startdate='Jan+1,+2008', enddate='Feb+1,+2008
↪')                  # optional -- internet # known bug
[84.6000, 83.9500, 80.4900, 72.9900, ... 83.0000, 54.8700, 56.4200, 56.6700,␣
↪57.8500]
```

Or, you can initialize stock data first and then extract the Close data:

```
sage: c = finance.Stock('vmw')  # optional -- internet # known bug
sage: c.history(startdate='Feb+1,+2008', enddate='Mar+1,+2008')[:5]    #␣
↪optional -- internet # known bug
[
  1-Feb-08 56.98 58.14 55.06 57.85    2490481,
  4-Feb-08 58.00 60.47 56.91 58.05    1840709,
  5-Feb-08 57.60 59.30 57.17 59.30    1712179,
  6-Feb-08 60.32 62.00 59.50 61.52    2211775,
  7-Feb-08 60.50 62.75 59.56 60.80    1521651
]
sage: c.close()    # optional -- internet # known bug
[57.8500, 58.0500, 59.3000, 61.5200, ... 58.2900, 60.1800, 59.8600, 59.9500,␣
↪58.6700]
```

Otherwise, *history()* will be called with the default arguments returning a year's worth of data:

```
sage: finance.Stock('vmw').close()    # random; optional -- internet # known␣
↪bug
[57.7100, 56.9900, 55.5500, 57.3300, 65.9900 ... 84.9900, 84.6000, 83.9500,␣
↪80.4900, 72.9900]
```

current_price_data()

Get Yahoo current price data for this stock.

This method returns a dictionary with the following keys:

`'price'`	`'change'`	`'volume'`	`'avg_daily_volume'`
`'stock_exchange'`	`'market_cap'`	`'book_value'`	`'ebitda'`
`'dividend_per_share'`	`'dividend_yield'`	`'earnings_per_share'`	`'52_week_high'`
`'52_week_low'`	`'50day_moving_avg'`	`'200day_moving_avg'`	`'price_earnings_ratio'`
`'price_earnings_growth_ratio'`	`'price_sales_ratio'`	`'price_book_ratio'`	`'short_ratio'`.

EXAMPLES:

```
sage: finance.Stock('GOOG').current_price_data()  #  random; optional -␣
↪internet # known bug
{'200day_moving_avg': '536.57',
 '50day_moving_avg': '546.01',
 '52_week_high': '599.65',
 '52_week_low': '487.56',
 'avg_daily_volume': '1826450',
 'book_value': '153.64',
 'change': '+0.56',
 'dividend_per_share': 'N/A',
 'dividend_yield': 'N/A',
 'earnings_per_share': '20.99',
 'ebitda': '21.48B',
 'market_cap': '366.11B',
 'price': '537.90',
 'price_book_ratio': '3.50',
 'price_earnings_growth_ratio': '0.00',
 'price_earnings_ratio': '25.62',
 'price_sales_ratio': '5.54',
 'short_ratio': '1.50',
 'stock_exchange': '"NMS"',
 'volume': '1768181'}
```

google (**args*, ***kwds*)

Deprecated: Use `history()` instead. See trac ticket #18355 for details.

history (*startdate='Jan+1, +1900'*, *enddate=None*, *histperiod='daily'*)

Return an immutable sequence of historical price data for this stock, obtained from Google. OHLC data is stored internally as well. By default, returns the past year's daily OHLC data.

Dates `startdate` and `enddate` should be formatted `'Mon+d,+yyyy'`, where `'Mon'` is a three character abbreviation of the month's name.

Note: Google Finance returns the past year's financial data by default when `startdate` is set too low from the equity's date of going public. By default, this function only looks at the NASDAQ and NYSE markets. However, if you specified the market during initialization of the stock (i.e. `finance.Stock("OTC:NTDOY")`), this method will give correct results.

INPUT:

- `startdate` – string, (default: `'Jan+1,+1900'`)
- `enddate` – string, (default: current date)
- `histperiod` – string, (`'daily'` or `'weekly'`)

OUTPUT:

A sequence.

EXAMPLES:

We get the first five days of VMware's stock history:

```
sage: finance.Stock('vmw').history('Aug+13,+2007')[:5] # optional -- internet
↪# known bug
[
 14-Aug-07 50.00 55.50 48.00 51.00   38262850,
```

(continues on next page)

(continued from previous page)

```
 15-Aug-07 52.11 59.87 51.50 57.71   10689100,
 16-Aug-07 60.99 61.49 52.71 56.99    6919500,
 17-Aug-07 59.00 59.00 54.45 55.55    3087000,
 20-Aug-07 56.05 57.50 55.61 57.33    2141900
]
sage: finance.Stock('F').history('Aug+20,+1992', 'Jul+7,+2008')[:5] #␣
↪optional -- internet # known bug
[
 20-Aug-92 0.00 7.90 7.73 7.83    5492698,
 21-Aug-92 0.00 7.92 7.66 7.68    5345999,
 24-Aug-92 0.00 7.59 7.33 7.35   11056299,
 25-Aug-92 0.00 7.66 7.38 7.61    8875299,
 26-Aug-92 0.00 7.73 7.64 7.68    6447201
]
```

Note that when `startdate` is too far prior to a stock's actual start date, Google Finance defaults to a year's worth of stock history leading up to the specified end date. For example, Apple's (AAPL) stock history only dates back to September 7, 1984:

```
sage: finance.Stock('AAPL').history('Sep+1,+1900', 'Jan+1,+2000')[0:5] #␣
↪optional -- internet # known bug
[
  4-Jan-99 0.00 1.51 1.43 1.47  238221200,
  5-Jan-99 0.00 1.57 1.48 1.55  352522800,
  6-Jan-99 0.00 1.58 1.46 1.49  337125600,
  7-Jan-99 0.00 1.61 1.50 1.61  357254800,
  8-Jan-99 0.00 1.67 1.57 1.61  169680000
]
```

Here is an example where we create and get the history of a stock that is not in NASDAQ or NYSE:

```
sage: finance.Stock("OTC:NTDOY").history(startdate="Jan+1,+2007", enddate=
↪"Jan+1,+2008")[:5]  # optional -- internet # known bug
[
  3-Jan-07 32.44 32.75 32.30 32.44     156283,
  4-Jan-07 31.70 32.40 31.20 31.70     222643,
  5-Jan-07 30.15 30.50 30.15 30.15      65670,
  8-Jan-07 30.10 30.50 30.00 30.10     130765,
  9-Jan-07 29.90 30.05 29.60 29.90     103338
]
```

Here, we create a stock by cid, and get historical data. Note that when using historical, if a cid is specified, it will take precedence over the stock's symbol. So, if the symbol and cid do not match, the history based on the contract id will be returned.

```
sage: sage.finance.stock.Stock("AAPL", 22144).history(startdate='Jan+1,+1990
↪')[:5] #optional -- internet # known bug
[
  8-Jun-99 0.00 1.74 1.70 1.70   78414000,
  9-Jun-99 0.00 1.73 1.69 1.73   88446400,
 10-Jun-99 0.00 1.72 1.69 1.72   79262400,
 11-Jun-99 0.00 1.73 1.65 1.66   46261600,
 14-Jun-99 0.00 1.67 1.61 1.62   39270000
]
```

load_from_file(*file*)

Load historical data from a local csv formatted data file. Note that no symbol data is included in Google

Finance's csv data. The csv file must be formatted in the following way, just as on Google Finance:

```
Timestamp,Open,High,Low,Close,Volume
```

INPUT:

- `file` – local file with Google Finance formatted OHLC data.

OUTPUT:

A sequence – OHLC data.

EXAMPLES:

Suppose you have a file in your home directory containing Apple stock OHLC data, such as that from Google Finance, called `AAPL-minutely.csv`. One can load this information into a Stock object like so. Note that the path must be explicit:

```
sage: filename = tmp_filename(ext='.csv')
sage: with open(filename, 'w') as fobj:
....:     _ = fobj.write("Date,Open,High,Low,Close,Volume\n1212405780,187.80,
↪187.80,187.80,187.80,100\n1212407640,187.75,188.00,187.75,188.00,
↪2000\n1212407700,188.00,188.00,188.00,188.00,1000\n1212408000,188.00,188.11,
↪188.00,188.00,2877\n1212408060,188.00,188.00,188.00,188.00,687")
sage: finance.Stock('aapl').load_from_file(filename)[:5]
[
1212408060 188.00 188.00 188.00 188.00        687,
1212408000 188.00 188.11 188.00 188.00       2877,
1212407700 188.00 188.00 188.00 188.00       1000,
1212407640 187.75 188.00 187.75 188.00       2000,
1212405780 187.80 187.80 187.80 187.80        100
]
```

Note that since the source file doesn't contain information on which equity the information comes from, the symbol designated at initialization of Stock need not match the source of the data. For example, we can initialize a Stock object with the symbol `'goog'`, but load data from `'aapl'` stock prices:

```
sage: finance.Stock('goog').load_from_file(filename)[:5]
[
1212408060 188.00 188.00 188.00 188.00        687,
1212408000 188.00 188.11 188.00 188.00       2877,
1212407700 188.00 188.00 188.00 188.00       1000,
1212407640 187.75 188.00 187.75 188.00       2000,
1212405780 187.80 187.80 187.80 187.80        100
]
```

market_value()

Return the current market value of this stock.

OUTPUT:

A Python float.

EXAMPLES:

```
sage: finance.Stock('goog').market_value()    # random; optional - internet #␣
↪known bug
575.83000000000004
```

open(**args*, ***kwds*)

Return a time series containing historical opening prices for this stock. If no arguments are given, will

return last acquired historical data. Otherwise, data will be gotten from Google Finance.

INPUT:

- `startdate` – string, (default: `'Jan+1,+1900'`)
- `enddate` – string, (default: current date)
- `histperiod` – string, (`'daily'` or `'weekly'`)

OUTPUT:

A time series – close price data.

EXAMPLES:

You can directly obtain Open data as so:

```
sage: finance.Stock('vmw').open(startdate='Jan+1,+2008', enddate='Feb+1,+2008
↪')                  # optional -- internet # known bug
[85.4900, 84.9000, 82.0000, 81.2500, ... 82.0000, 58.2700, 54.4900, 55.6000,␣
↪56.9800]
```

Or, you can initialize stock data first and then extract the Open data:

```
sage: c = finance.Stock('vmw') # optional -- internet # known bug
sage: c.history(startdate='Feb+1,+2008', enddate='Mar+1,+2008')[:5]     #␣
↪optional -- internet # known bug
[
  1-Feb-08 56.98 58.14 55.06 57.85    2490481,
  4-Feb-08 58.00 60.47 56.91 58.05    1840709,
  5-Feb-08 57.60 59.30 57.17 59.30    1712179,
  6-Feb-08 60.32 62.00 59.50 61.52    2211775,
  7-Feb-08 60.50 62.75 59.56 60.80    1521651
]
sage: c.open()     # optional -- internet # known bug
[56.9800, 58.0000, 57.6000, 60.3200, ... 56.5500, 59.3000, 60.0000, 59.7900,␣
↪59.2600]
```

Otherwise, *history()* will be called with the default arguments returning a year's worth of data:

```
sage: finance.Stock('vmw').open()    # random; optional -- internet # known bug
[52.1100, 60.9900, 59.0000, 56.0500, 57.2500, ... 83.0500, 85.4900, 84.9000,␣
↪82.0000, 81.2500]
```

yahoo(**args*, ***kwds*)

Deprecated: Use *current_price_data()* instead. See trac ticket #18355 for details.

CHAPTER

THREE

TOOLS FOR WORKING WITH FINANCIAL OPTIONS

AUTHORS: - Brian Manion, 2013: initial version

sage.finance.option.**black_scholes**(*spot_price*, *strike_price*, *time_to_maturity*, *risk_free_rate*, *vol*, *opt_type*)

Calculates call/put price of European style options using Black-Scholes formula. See [**?**] for one of many standard references for this formula.

INPUT:

- `spot_price` – The current underlying asset price
- `strike_price` – The strike of the option
- `time_to_maturity` – The # of years until expiration
- `risk_free_rate` – The risk-free interest-rate
- `vol` – The volatility
- `opt_type` – string; The type of option, either `'put'` for put option or `'call'` for call option

OUTPUT:

The price of an option with the given parameters.

EXAMPLES:

```
sage: finance.black_scholes(42, 40, 0.5, 0.1, 0.2, 'call')       # abs tol 1e-10
4.759422392871532
sage: finance.black_scholes(42, 40, 0.5, 0.1, 0.2, 'put')        # abs tol 1e-10
0.8085993729000958
sage: finance.black_scholes(100, 95, 0.25, 0.1, 0.5, 'call')     # abs tol 1e-10
13.695272738608132
sage: finance.black_scholes(100, 95, 0.25, 0.1, 0.5, 'put')      # abs tol 1e-10
6.349714381299734
sage: finance.black_scholes(527.07, 520, 0.424563772, 0.0236734,0.15297,
↪'whichever makes me more money')
Traceback (most recent call last):
...
ValueError: 'whichever makes me more money' is not a valid string
```

CHAPTER

FOUR

MULTIFRACTAL RANDOM WALK

This module implements the fractal approach to understanding financial markets that was pioneered by Mandelbrot. In particular, it implements the multifractal random walk model of asset returns as developed by Bacry, Kozhemyak, and Muzy, 2006, *Continuous cascade models for asset returns* and many other papers by Bacry et al. See http://www.cmap.polytechnique.fr/~bacry/ftpPapers.html

See also Mandelbrot's *The Misbehavior of Markets* for a motivated introduction to the general idea of using a self-similar approach to modeling asset returns.

One of the main goals of this implementation is that everything is highly optimized and ready for real world high performance simulation work.

AUTHOR:

- William Stein (2008)

sage.finance.fractal.**fractional_brownian_motion_simulation**(*H*, *sigma2*, *N*, *n=1*)

Returns the partial sums of a fractional Gaussian noise simulation with the same input parameters.

INPUT:

- `H` – float; `0 < H < 1`; the Hurst parameter.
- `sigma2` - float; innovation variance (should be close to 0).
- `N` – positive integer.
- `n` – positive integer (default: 1).

OUTPUT:

List of `n` time series.

EXAMPLES:

```
sage: set_random_seed(0)
sage: finance.fractional_brownian_motion_simulation(0.8,0.1,8,1)
[[-0.0754, 0.1874, 0.2735, 0.5059, 0.6824, 0.6267, 0.6465, 0.6289]]
sage: set_random_seed(0)
sage: finance.fractional_brownian_motion_simulation(0.8,0.01,8,1)
[[-0.0239, 0.0593, 0.0865, 0.1600, 0.2158, 0.1982, 0.2044, 0.1989]]
sage: finance.fractional_brownian_motion_simulation(0.8,0.01,8,2)
[[-0.0167, 0.0342, 0.0261, 0.0856, 0.1735, 0.2541, 0.1409, 0.1692],
 [0.0244, -0.0153, 0.0125, -0.0363, 0.0764, 0.1009, 0.1598, 0.2133]]
```

sage.finance.fractal.**fractional_gaussian_noise_simulation**(*H*, *sigma2*, *N*, *n=1*)

Return `n` simulations with `N` steps each of fractional Gaussian noise with Hurst parameter `H` and innovations variance `sigma2`.

INPUT:

- H – float; 0 < H < 1; the Hurst parameter.
- sigma2 - positive float; innovation variance.
- N – positive integer; number of steps in simulation.
- n – positive integer (default: 1); number of simulations.

OUTPUT:

List of n time series.

EXAMPLES:

We simulate a fractional Gaussian noise:

```
sage: set_random_seed(0)
sage: finance.fractional_gaussian_noise_simulation(0.8,1,10,2)
[[-0.1157, 0.7025, 0.4949, 0.3324, 0.7110, 0.7248, -0.4048, 0.3103, -0.3465, 0.
↪2964],
 [-0.5981, -0.6932, 0.5947, -0.9995, -0.7726, -0.9070, -1.3538, -1.2221, -0.0290,␣
↪1.0077]]
```

The sums define a fractional Brownian motion process:

```
sage: set_random_seed(0)
sage: finance.fractional_gaussian_noise_simulation(0.8,1,10,1)[0].sums()
[-0.1157, 0.5868, 1.0818, 1.4142, 2.1252, 2.8500, 2.4452, 2.7555, 2.4090, 2.7054]
```

ALGORITHM:

See *Simulating a Class of Stationary Gaussian Processes using the Davies-Harte Algorithm, with Application to Long Meoryy Processes*, 2000, Peter F. Craigmile for a discussion and references for why the algorithm we give – which uses the `stationary_gaussian_simulation()` function.

sage.finance.fractal.**multifractal_cascade_random_walk_simulation**(*T*, *lambda2*, *ell*, *sigma2*, *N*, *n=1*)

Return a list of n simulations of a multifractal random walk using the log-normal cascade model of Bacry-Kozhemyak-Muzy 2008. This walk can be interpreted as the sequence of logarithms of a price series.

INPUT:

- T – positive real; the integral scale.
- lambda2 – positive real; the intermittency coefficient.
- ell – a small number – time step size.
- sigma2 – variance of the Gaussian white noise eps[n].
- N – number of steps in each simulation.
- n – the number of separate simulations to run.

OUTPUT:

List of time series.

EXAMPLES:

```
sage: set_random_seed(0)
sage: a = finance.multifractal_cascade_random_walk_simulation(3770,0.02,0.01,0.01,
↪10,3)
sage: a
```

(continues on next page)

(continued from previous page)

```
[[-0.0096, 0.0025, 0.0066, 0.0016, 0.0078, 0.0051, 0.0047, -0.0013, 0.0003, -0.
↪0043],
 [0.0003, 0.0035, 0.0257, 0.0358, 0.0377, 0.0563, 0.0661, 0.0746, 0.0749, 0.0689],
 [-0.0120, -0.0116, 0.0043, 0.0078, 0.0115, 0.0018, 0.0085, 0.0005, 0.0012, 0.
↪0060]]
```

The corresponding price series:

```
sage: a[0].exp()
[0.9905, 1.0025, 1.0067, 1.0016, 1.0078, 1.0051, 1.0047, 0.9987, 1.0003, 0.9957]
```

MORE DETAILS:

The random walk has n-th step $\text{eps}_n e^{\omega_n}$, where eps_n is gaussian white noise of variance σ^2 and ω_n is renormalized gaussian magnitude, which is given by a stationary gaussian simulation associated to a certain autocovariance sequence. See Bacry, Kozhemyak, Muzy, 2006, *Continuous cascade models for asset returns* for details.

sage.finance.fractal.**stationary_gaussian_simulation**(*s*, *N*, *n=1*)

Implementation of the Davies-Harte algorithm which given an autocovariance sequence (ACVS) `s` and an integer `N`, simulates `N` steps of the corresponding stationary Gaussian process with mean 0. We assume that a certain Fourier transform associated to `s` is nonnegative; if it isn't, this algorithm fails with a `NotImplementedError`.

INPUT:

- `s` – a list of real numbers that defines the ACVS. Optimally `s` should have length `N+1`; if not we pad it with extra 0's until it has length `N+1`.
- `N` – a positive integer.

OUTPUT:

A list of `n` time series.

EXAMPLES:

We define an autocovariance sequence:

```
sage: N = 2^15
sage: s = [1/math.sqrt(k+1) for k in [0..N]]
sage: s[:5]
[1.0, 0.7071067811865475, 0.5773502691896258, 0.5, 0.4472135954999579]
```

We run the simulation:

```
sage: set_random_seed(0)
sage: sim = finance.stationary_gaussian_simulation(s, N)[0]
```

Note that indeed the autocovariance sequence approximates `s` well:

```
sage: [sim.autocovariance(i) for i in [0..4]]
[0.98665816086255..., 0.69201577095377..., 0.56234006792017..., 0.48647965409871..
↪., 0.43667043322102...]
```

Warning: If you were to do the above computation with a small value of `N`, then the autocovariance sequence would not approximate `s` very well.

REFERENCES:

This is a standard algorithm that is described in several papers. It is summarized nicely with many applications at the beginning of *Simulating a Class of Stationary Gaussian Processes Using the Davies-Harte Algorithm, with Application to Long Memory Processes*, 2000, Peter F. Craigmile, which is easily found as a free PDF via a Google search. This paper also generalizes the algorithm to the case when all elements of `s` are nonpositive.

The book *Wavelet Methods for Time Series Analysis* by Percival and Walden also describes this algorithm, but has a typo in that they put a 2π instead of π a certain sum. That book describes exactly how to use Fourier transform. The description is in Section 7.8. Note that these pages are missing from the Google Books version of the book, but are in the Amazon.com preview of the book.

CHAPTER

FIVE

MARKOV SWITCHING MULTIFRACTAL MODEL

REFERENCE:

How to Forecast Long-Run Volatility: Regime Switching and the Estimation of Multifractal Processes, Calvet and Fisher, 2004.

AUTHOR:

- William Stein, 2008

class sage.finance.markov_multifractal.**MarkovSwitchingMultifractal**(*kbar*, *m0*, *sigma*, *gamma_kbar*, *b*)

INPUT:

- `kbar` – positive integer
- `m0` – float with `0 <= m0 <= 2`
- `sigma` – positive float
- `gamma_kbar` – float with `0 <= gamma_kbar < 1`
- `b` – float > 1

EXAMPLES:

```
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,0.5,0.95,3); msm
Markov switching multifractal model with m0 = 1.4, sigma = 0.5, b = 3.0, and␣
↪gamma_8 = 0.95
sage: yen_usd = finance.MarkovSwitchingMultifractal(10,1.448,0.461,0.998,3.76)
sage: cad_usd = finance.MarkovSwitchingMultifractal(10,1.278,0.262,0.644,2.11)
sage: dm = finance.MarkovSwitchingMultifractal(10,1.326,0.643,0.959,2.7)
```

b()

Return parameter b of Markov switching multifractal model.

EXAMPLES:

```
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,1,0.95,3)
sage: msm.b()
3.0
```

gamma()

Return the vector of the kbar transitional probabilities.

OUTPUT:

- gamma – a tuple of `self.kbar()` floats.

EXAMPLES:

```
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,1.0,0.95,3)
sage: msm.gamma()
(0.001368852970712986, 0.004100940201672509, 0.012252436441829..., 0.
↪03630878209190..., 0.10501923017634..., 0.28312883556311..., 0.
↪6315968501359..., 0.95000000000000...)
```

gamma_kbar()

Return parameter gamma_kbar of Markov switching multifractal model.

EXAMPLES:

```
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,0.01,0.95,3)
sage: msm.gamma_kbar()
0.95
```

kbar()

Return parameter kbar of Markov switching multifractal model.

EXAMPLES:

```
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,0.01,0.95,3)
sage: msm.kbar()
8
```

m0()

Return parameter m0 of Markov switching multifractal model.

EXAMPLES:

```
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,1,0.95,3)
sage: msm.m0()
1.4
```

sigma()

Return parameter sigma of Markov switching multifractal model.

EXAMPLES:

```
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,1,0.95,3)
sage: msm.sigma()
1.0
```

simulation(*n*)

Same as self.simulations, but run only 1 time, and returns a time series instead of a list of time series.

INPUT:

- n – a positive integer.

EXAMPLES:

```
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,1.0,0.95,3)
sage: msm.simulation(5)
[0.0059, -0.0097, -0.0101, -0.0110, -0.0067]
sage: msm.simulation(3)
[0.0055, -0.0084, 0.0141]
```

simulations(*n*, *k=1*)

Return k simulations of length n using this Markov switching multifractal model for n time steps.

INPUT:

- n – positive integer; number of steps.
- k – positive integer (default: 1); number of simulations.

OUTPUT:

list – a list of TimeSeries objects.

EXAMPLES:

```
sage: cad_usd = finance.MarkovSwitchingMultifractal(10,1.278,0.262,0.644,2.
↪11); cad_usd
Markov switching multifractal model with m0 = 1.278, sigma = 0.262, b = 2.11,␣
↪and gamma_10 = 0.644
```

CHAPTER

SIX

MARKOV SWITCHING MULTIFRACTAL MODEL

Cython code

sage.finance.markov_multifractal_cython.**simulations**(*n*, *k*, *m0*, *sigma*, *kbar*, *gamma*)

Return k simulations of length n using the Markov switching multifractal model.

INPUT: n, k – positive integers m0, sigma – floats kbar – integer gamma – list of floats

OUTPUT: list of lists

EXAMPLES:

```
sage: set_random_seed(0)
sage: msm = finance.MarkovSwitchingMultifractal(8,1.4,1.0,0.95,3)
sage: import sage.finance.markov_multifractal_cython
sage: sage.finance.markov_multifractal_cython.simulations(5,2,1.278,0.262,8,msm.
↪gamma())
[[0.0014, -0.0023, -0.0028, -0.0030, -0.0019], [0.0020, -0.0020, 0.0034, -0.0010,␣
↪-0.0004]]
```

CHAPTER

SEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

f

INDEX

H

I

K

L

M

N

O

P

R

S

T

U

V

Y