latimes-calculate Documentation

Release 0.2

Los Angeles Times Data Desk

September 13, 2016

Contents

Some simple math we use to do journalism

CHAPTER 1

Getting started

Install the latest package from pypi.

```
$ pip install latimes-calculate
```

Note: For most functions, there are no additional requirements. The exception is the small number of geospatial functions, which require GeoDjango.

CHAPTER 2

Documentation

2.1 Basic functions

2.1.1 Adjusted-monthly value

adjusted_monthly_value(*value*, *datetime*)

Accepts a value and a datetime object, and then prorates the value to a 30-day figure depending on how many days are in the month. This can be useful for month-to-month comparisons in circumstances where fluctuations in the number of days per month may skew the analysis. For instance, February typically has only 28 days, in comparison to March, which has 31.

```
>>> import calculate
>>> calculate.adjusted_monthly_value(10, datetime.datetime(2009, 4, 1))
10.0
>>> calculate.adjusted_monthly_value(10, datetime.datetime(2009, 2, 17))
10.714285714285714
>>> calculate.adjusted_monthly_value(10, datetime.datetime(2009, 12, 31))
9.67741935483871
```

2.1.2 Age

age(*born*, *as_of=None*)

Returns the current age, in years, of a person born on the provided date.

First argument should be the birthdate and can be a datetime.date or datetime.datetime object, although datetimes will be converted to a date object and hours, minutes and seconds will not be part of the calculation.

The second argument is the *as_of* date that the person's age will be calculate at. By default, it is not provided and the age is returned as of the current date. But if you wanted to calculate someone's age at a past or future date, you could do that by providing the *as_of* date as the second argument.

```
>>> import calculate
>>> from datetime import date
>>> dob = date(1982, 7, 22)
>>> calculate.age(dob)
29 # As of the writing of this README, of course.
>>> as_of = date(1982, 7, 23)
>>> calculate.age(dob, as_of)
0
```

2.1.3 At percentile

at_percentile(*data_list*, *value*, *interpolation='fraction'*)

Accepts a list of values and a percentile for which to return the value. A percentile of, for example, 80 means that 80 percent of the scores in the sequence are below the given score. If the requested percentile falls between two values, the result can be interpolated using one of the following methods. The default is "fraction".

- `fraction`: The value proportionally between the pair of bordering values.
- `lower`: The lower of the two bordering values.
- `higher`: The higher of the two bordering values.

```
>>> import calculate
>>> calculate.at_percentile([1, 2, 3, 4], 75)
3.25
>>> calculate.at_percentile([1, 2, 3, 4], 75, interpolation='lower')
3.0
>>> calculate.at_percentile([1, 2, 3, 4], 75, interpolation='higher')
4.0
```

2.1.4 Benford's Law

benfords_law(*number_list*, *method='first_digit'*, *verbose=True*)

Accepts a list of numbers and applies a quick-and-dirty run against Benford's Law. Benford's Law makes statements about the occurance of leading digits in a dataset. It claims that a leading digit of 1 will occur about 30 percent of the time, and each number after it a little bit less, with the number 9 occuring the least. Datasets that greatly vary from the law are sometimes suspected of fraud.

The function returns the Pearson correlation coefficient, also known as Pearson's r, which reports how closely the two datasets are related. This function also includes a variation on the classic Benford analysis popularized by blogger Nate Silver, who conducted an analysis of the final digits of polling data. To use Silver's variation, provide the keyword argument *method* with the value 'last_digit'. To prevent the function from printing, set the optional keyword argument *verbose* to False. This function is based upon code from a variety of sources around the web, but owes a particular debt to the work of Christian S. Perone.

```
>>> import calculate
>>> calculate.benfords_law([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
BENFORD'S LAW: FIRST_DIGIT

Pearson's R: 0.86412304649

| Number | Count | Expected Percentage | Actual Percentage |
------------------------------------------------------------
| 1      | 2     | 30.1029995664       | 20.0              |
| 2      | 1     | 17.6091259056       | 10.0              |
| 3      | 1     | 12.4938736608       | 10.0              |
| 4      | 1     | 9.69100130081       | 10.0              |
| 5      | 1     | 7.91812460476       | 10.0              |
| 6      | 1     | 6.69467896306       | 10.0              |
| 7      | 1     | 5.79919469777       | 10.0              |
| 8      | 1     | 5.11525224474       | 10.0              |
| 9      | 1     | 4.57574905607       | 10.0              |

>>> calculate.benfords_law([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], verbose=False)
-0.863801937698704
```

2.1.5 Competition rank

competition_rank(*data_list*, *obj*, *order_by*, *direction='desc'*)

Accepts a list, an item plus the value and direction to order by. Then returns the supplied object's competition rank as an integer. In competition ranking equal numbers receive the same ranking and a gap is left before the next value (i.e. "1224"). You can submit a Django queryset, objects, or just a list of dictionaries.

```
>>> import calculate
>>> qs = Player.objects.all().order_by("-career_home_runs")
>>> ernie = Player.objects.get(first_name__iexact='Ernie', last_name__iexact='Banks')
>>> eddie = Player.objects.get(first_name__iexact='Eddie', last_name__iexact='Matthews')
>>> mel = Player.objects.get(first_name__iexact='Mel', last_name__iexact='Ott')
>>> calculate.competition_rank(qs, ernie, career_home_runs', direction='desc')
21
>>> calculate.competition_rank(qs, eddie, 'career_home_runs', direction='desc')
21
>>> calculate.competition_rank(qs, mel, 'career_home_runs', direction='desc')
23
```

2.1.6 Date range

date_range(*start_date*, *end_date*)

Returns a generator of all the days between two date objects. Results include the start and end dates. Arguments can be either datetime.datetime or date type objects.

```
>>> import datetime
>>> import calculate
>>> dr = calculate.date_range(datetime.date(2009,1,1), datetime.date(2009,1,3))
>>> dr
<generator object at 0x718e90>
>>> list(dr)
[datetime.date(2009, 1, 1), datetime.date(2009, 1, 2), datetime.date(2009, 1, 3)]
```

2.1.7 Decile

decile(*data_list*, *score*, *kind='weak'*)

Accepts a sample of values and a single number to add to it and determine the decile equivilent of its percentile rank.

By default, the method used to negotiate gaps and ties is "weak" because it returns the percentile of all values at or below the provided value. For an explanation of alternative methods, refer to the `percentile` function.

```
>>> import calculate
>>> calculate.decile([1, 2, 3, 3, 4], 3)
9
```

2.1.8 Ethnolinguistic Fractionalization Index

elfi(*data_list*)

The ELFI is a simplified method for calculating the Ethnolinguistic Fractionalization Index (ELFI). This is one form of what is commonly called a "diversity index." Accepts a list of decimal percentages, which are used to calculate the index. Returns a decimal value as a floating point number.

```
>>> import calculate
>>> calculate.elfi([0.2, 0.5, 0.05, 0.25])
0.64500000000000002
```

2.1.9 Equal-sized breakpoints

equal_sized_breakpoints(*data_list*, *classes*)

Returns break points for groups of equal size, known as quartiles, quintiles, etc. Provide a list of data values and the number of classes you'd like the list broken up into. No flashy math, just sorts them in order and makes the cuts.

```
>>> import calculate
>>> calculate.equal_sized_breakpoints(range(1,101), 5)
[1.0, 21.0, 41.0, 61.0, 81.0, 100]
```

2.1.10 Margin of victory

margin_of_victory(*data_list*)

Accepts a list of numbers and returns the difference between the first place and second place values.

This can be useful for covering elections as an easy to way to figure out the margin of victory for a leading candidate.

```
>>> import calculate
>>> # 2008 Iowa caucus results for [Edwards, Clinton, Obama]
>>> calculate.margin_of_victory([3285, 2804, 7170])
3885
```

2.1.11 Mean (Average)

mean(*data_list*)

Accepts a sample of values and returns their mean. The mean is the sum of all values in the sample divided by the number of members. It is also known as the average. Since the value is strongly influenced by outliers, median is generally a better indicator of central tendency.

```
>>> import calculate
>>> calculate.mean([1,2,3])
2.0
>>> calculate.mean([1, 99])
50.0
```

2.1.12 Median

median(*data_list*)

Accepts a list of numbers and returns the median value. The median is the number in the middle of a sequence, with 50 percent of the values above, and 50 percent below. In cases where the sequence contains an even number of values – and therefore no exact middle – the two values nearest the middle are averaged and the mean returned.

```
>>> import calculate
>>> calculate.median([1,2,3])
2.0
```

```
>> calculate.median((1,4,3,2))
2.5
```

2.1.13 Mode

mode (*data_list*)

Accepts a sample of numbers and returns the mode value. The mode is the most common value in a data set. If there is a tie for the highest count, no value is returned.

```
>>> import calculate
>>> calculate.mode([1,2,2,3])
2.0
>>> calculate.mode([1,2,3])
>>>
```

2.1.14 Ordinal rank

ordinal_rank (*sequence*, *item*, *order_by=None*, *direction='desc'*)

Accepts a list and an object. Returns the object's ordinal rank as an integer. Does not negiotiate ties.

```
>>> import calculate
>>> qs = Player.objects.all().order_by("-career_home_runs")
>>> barry = Player.objects.get(first_name__iexact='Barry', last_name__iexact='Bonds')
>>> calculate.ordinal_rank(qs, barry)
1
```

2.1.15 Pearson's r

pearson (*list_one*, *list_two*)

Accepts paired lists and returns a number between -1 and 1, known as Pearson's r, that indicates of how closely correlated the two datasets are. A score of close to one indicates a high positive correlation. That means that X tends to be big when Y is big. A score close to negative one indicates a high negative correlation. That means X tends to be small when Y is big. A score close to zero indicates little correlation between the two datasets.

A warning, though, correlation does not equal causation. Just because the two datasets are closely related doesn't not mean that one causes the other to be the way it is.

```
>>> import calculate
>>> calculate.pearson([6,5,2], [2,5,6])
-0.8461538461538467
```

2.1.16 Per capita

per_capita (*value*, *population*, *per=10000*, *fail_silently=True*)

Accepts two numbers, a value and population total, and returns the per capita rate. By default, the result is returned as a per 10,000 person figure. If you divide into zero – an illegal operation – a null value is returned by default. If you prefer for an error to be raised, set the kwarg 'fail_silently' to False.

```
>>> import calculate
>>> calculate.per_capita(12, 100000)
1.2
```

2.1.17 Per square mile

per_sqmi(*value*, *square_miles*, *fail_silently=True*)

Accepts two numbers, a value and an area, and returns the per square mile rate. Not much more going on here than a simple bit of division. If you divide into zero – an illegal operation – a null value is returned by default. If you prefer for an error to be raised, set the kwarg 'fail_silently' to False.

```
>>> import calculate
>>> calculate.per_sqmi(20, 10)
2.0
```

2.1.18 Percentage

percentage(*value*, *total*, *multiply=True*, *fail_silently=True*)

Accepts two integers, a value and a total. The value is divided into the total and then multiplied by 100, returning its percentage as a float. If you don't want the number multiplied by 100, set the 'multiply' kwarg to False. If you divide into zero – an illegal operation – a null value is returned by default. If you prefer for an error to be raised, set the kwarg 'fail_silently' to False.

```
>>> import calculate
>>> calculate.percentage(2, 10)
20.0
>>> calculate.percentage(2,0, multiply=False)
0.20000000000000001
>>> calculate.percentage(2,0)
```

2.1.19 Percentage change

percentage_change(*old_value*, *new_value*, *multiply=True*, *fail_silently=True*)

Accepts two integers, an old and a new number, and then measures the percent change between them. The change between the two numbers is determined and then divided into the original figure. By default, it is then multiplied by 100, and returning as a float. If you don't want the number multiplied by 100, set the 'multiply' kwarg to False. If you divide into zero – an illegal operation – a null value is returned by default. If you prefer for an error to be raised, set the kwarg 'fail_silently' to False.

```
>>> import calculate
>>> calculate.percentage_change(2, 10)
400.0
```

2.1.20 Percentile

percentile(*data_list*, *value*, *kind='weak'*)

Accepts a sample of values and a single number to add to it and determine its percentile rank. A percentile of, for example, 80 percent means that 80 percent of the scores in the sequence are below the given score. In the case of gaps or ties, the exact definition depends on the type of the calculation stipulated by the "kind" keyword argument. There are three kinds of percentile calculations provided here. The default is "weak".

- `weak`: Corresponds to the definition of a cumulative distribution function, with the result generated by returning the percentage of values at or equal to the the provided value.
- `strict`: Similar to "weak", except that only values that are less than the given score are counted. This can often produce a result much lower than "weak" when the provided score is occurs many times in the sample.

- `mean`: The average of the "weak" and "strict" scores.

```
>>> import calculate
>>> calculate.percentile([1, 2, 3, 4], 3)
75.0
>>> calculate.percentile([1, 2, 3, 3, 4], 3, kind='strict')
40.0
>>> calculate.percentile([1, 2, 3, 3, 4], 3, kind='weak')
80.0
>>> calculate.percentile([1, 2, 3, 3, 4], 3, kind='mean')
60.0
```

2.1.21 Range

`range` (*data_list*)

Accepts a sample of values and return the range. The range is the difference between the maximum and minimum values of a data set.

```
>>> import calculate
>>> calculate.range([1,2,3])
2
>>> calculate.range([2,2])
0
```

2.1.22 Split at breakpoints

`split_at_breakpoints` (*data_list*, *breakpoint_list*)

Splits up a list at the provided breakpoints. First argument is a list of data values. Second is a list of the breakpoints you'd like it to be split up with. Returns a list of lists, in order by breakpoint.

Useful for splitting up a list after you've determined breakpoints using another method like calculate.equal_sized_breakpoints.

```
>>> import calculate
>>> l = range(1,101)
>>> bp = calculate.equal_sized_breakpoints(l, 5)
>>> print bp
[1.0, 21.0, 41.0, 61.0, 81.0, 100]
>>> print calculate.split_at_breakpoints(l, bp)
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20], [21, 22, 23, 24, 25...
```

2.1.23 Standard deviation

`standard_deviation` (*data_list*)

Accepts a sample of values and returns the standard deviation. Standard deviation measures how widely dispersed the values are from the mean. A lower value means the data tend to be bunched close to the average. A higher value means they tend to be further away. This is a "population" calculation that assumes that you are submitting all of the values, not a sample.

```
>>> import calculate
>>> calculate.standard_deviation([2,3,3,4])
0.70710678118654757
>>> calculate.standard_deviation([-2,3,3,40])
16.867127793432999
```

2.1.24 Summary statistics

summary_stats (*data_list*)

Accepts a sample of numbers and returns a pretty print out of a variety of descriptive statistics.

```
>>> import calculate
>>> calculate.summary_stats(range(1,101))
| Statistic               | Value         |
-----------------------------------------|
| n                       | 100           |
| mean                    | 50.5          |
| median                  | 50.5          |
| mode                    | None          |
| maximum                 | 100           |
| minimum                 | 1             |
| range                   | 99.0          |
| standard deviation      | 28.8660700477 |
| variation coefficient   | 0.57160534748 |
```

2.1.25 Variation coefficient

variation_coefficient (*data_list*)

Accepts a list of values and returns the variation coefficient, which is a normalized measure of the distribution.

This is the sort of thing you can use to compare the standard deviation of sets that are measured in different units.

Note that it uses our "population" standard deviation as part of the calculation, not a "sample" standard deviation.

```
>>> import calculate
>>> calculate.variation_coefficient(range(1, 1000))
0.5767726299562651
```

2.2 Geospatial functions

2.2.1 Mean center

mean_center (*obj_list*, *point_attribute_name='point'*)

Accepts a geoqueryset, list of objects or list of dictionaries, expected to contain GeoDjango Point objects as one of their attributes. Returns a Point object with the mean center of the provided points. The mean center is the average x and y of all those points. By default, the function expects the Point field on your model to be called 'point'. If the point field is called something else, change the kwarg 'point_attribute_name' to whatever your field might be called.

```
>>> import calculate
>>> calculate.mean_center(qs)
<Point object at 0x77a1694>
```

2.2.2 Nudge points

nudge_points (*geoqueryset*, *point_attribute_name='point'*, *radius=0.0001*)

A utility that accepts a GeoDjango QuerySet and nudges slightly apart any identical points. Nothing is returned. By default, the distance of the move is 0.0001 decimal degrees. I'm not sure if this will go wrong if your data is in a different unit of measurement. This can be useful for running certain geospatial statistics, or even for presentation issues, like spacing out markers on a Google Map for instance.

```
>>> import calculate
>>> calculate.nudge_points(qs)
>>>
```

2.2.3 Random point

random_point(*extent*)

A utility that accepts the extent of a polygon and returns a random point from within its boundaries. The extent is a four-point tuple with (xmin, ymin, xmax, ymax).

```
>>> polygon = Model.objects.get(pk=1).polygon
>>> import calculate
>>> calculate.random_point(polygon.extent)
```

2.2.4 Standard-deviation distance

standard_deviation_distance(*obj_list*, *point_attribute_name='point'*)

Accepts a geoqueryset, list of objects or list of dictionaries, expected to contain objects with Point properties, and returns a float with the standard deviation distance of the provided points. The standard deviation distance is the average variation in the distance of points from the mean center. By default, the function expects the Point field on your model to be called `point`. If the point field is called something else, change the kwarg `point_attribute_name` to whatever your field might be called.

```
>>> import calculate
>>> calculate.standard_deviation_distance(qs)
0.046301584704149731
```

CHAPTER 3

Contributing

- Code repository: https://github.com/datadesk/latimes-calculate
- Issues: https://github.com/datadesk/latimes-calculate/issues
- Packaging: https://pypi.python.org/pypi/latimes-calculate
- Testing: https://travis-ci.org/datadesk/latimes-calculate
- Coverage: https://coveralls.io/r/datadesk/latimes-calculate

Index