CS 207 Systems Development for Computational Science

Non-uniform Time and Lazy Functions

This will be graded along with the rest of your codebase at the course Milestones.

Non-uniform time series

To this point, your `TimeSeries` classes have dealt with either a single vector of information, or two, one for times and one for values. (`def __init__(self, values, times=None)`) But we have not done much with a two vectors implementation, preferring to keep the times as optional. We'll change this here.

Single vectors are useful for regularly sampled time series data, like temperatures over the course of a year or stock market prices per day. But not all time series data is this nice, and today we'll be reworking our code to be a little more realistic.

One `values` vector *implicit* samples by default: you assume that the values are observations at regular intervals of some arbitrary unit. Then, last time you switched to *explicit* samples, recording a value and a time point for every observation.

The changes you'll be making today are *interface-breaking* changes. In other words, we're asking you to change your code such that it will be incompatible with your previous version. You'll need to write or rewrite new tests to exercise your new code.

Back to top ↑

Modifying your storage

We'd like you to rewrite your `ArrayTimeSeries` class, with two goals:

- We want you to store two arrays internally, even if you only have a single vector of values. The first should represent a list of time points, and the other should be a set of values.
- Go ahead and use `numpy` arrays for the internal storage.

Please change your class constructor to take two sequences as input. The call signature should look something like this now:

#5 (Lu Shen)

issue

```
#order of arguments has changed because no default
def __init__(self, times, values): #notice no default for times now
  ...
```

The goal of your new arrays is to work something like a cross between an array and a dictionary. Your class is an *ordered* container, where each time point should be monotonically increasing (you don't need to enforce this right now, but you're welcome to if you want). It's like a sequence or array in this way.

Make sure you implement the `__iter__`, `itertimes`, `iteritems`, `__len__`, `__getitem__` and `__setitem__` (the latter two take an index and get a value and set a value correspoinding to that index respectively..note i say index, not time)

Back to top ↑

Linear interpolation #7 (Sarah Andre)

Now that we've got a shiny new class, let's make it a little more useful.

Random sampling is a fairly common way to collect time series data, but it suffers from the issue that the domain between two independent experiments is almost certainly not the same. We'll fix that through by introducing a simple interpolation function.

Let's start with an example:

```
a = TimeSeries([0,5,10], [1,2,3])
b = TimeSeries([2.5,7.5], [100, -100])
# Simple cases
a.interpolate([1]) == TimeSeries([1],[1.2])
a.interpolate(b.itertimes()) == TimeSeries([2.5,7.5], [1.5, 2.5])
# Boundary conditions
a.interpolate([-100,100]) == TimeSeries([-100,100],[1,3])
```

The idea is simply that for every new time point passed to the `interpolate` method, we'd like you to compute a value for the TimeSeries class assuming that it is a piecewise-linear function. In other words, take the nearest two time points, draw a line between them, and pick the value at the new time point. Use stationary boundary conditions: so if a new time point is smaller than the first existing time point, just use the first value; likewise for larger time points.

Your `interpolate` function should return a *new* `TimeSeries` instance; it should **not** modify the existing one.

As a smoke test, you might want to try this: create a time series instance with values for a couple time points between 0 and 1, the try your function on `np.random.random(1000)` and plot the new time series. You should be able to see your piecewise linear approximation show up. (This is not required, and you don't need to turn anything in for this.)

Back to top ↑

Lazy evaluation

Create a new file called `lazy.py`. (issue #8, Andrew)

Inside, create a new class called `LazyOperation` , which will be our thunk. The constructor should take one required argument `function` and then arbitrary positional and keyword arguments (this is the `*args` and `**kwargs` syntax, if you remember). The constructor doesn't need to do anything but store them internally for now.

This follows what we did in the lecture.

Write a `@lazy` decorator now.

Add in a `lazy_add` and a `lazy_mul` and use these to write tests for `lazy.py` .

If you've done it correctly, you should be able to run something like this:

```
isinstance( lazy_add(1,2), LazyOperation ) == True
```

Back to top ↑

Lazy TimeSeries (issue #9, Andrew)

The last step is to hook this up to your `TimeSeries` class.

Technically, we don't actually need to do this:

```
@lazy
def check_length(a,b):
  return len(a)==len(b)
thunk = check_length(TimeSeries(range(0,4),range(1,5)), TimeSeries(range(1,5),range(2,6)))
assert thunk.eval()==True
```

We'd like you to create a `lazy` property method in your `TimeSeries` class. All this method does is return a new `LazyOperation` instance using an identity function (a function with one argument that just returns the argument) and `self` as the only argument. This wraps up the `TimeSeries` instance and a function which does nothing and saves them both for later.

This example should give identical results: `python x = TimeSeries([1,2,3,4],[1,4,9,16]) print(x) print(x.lazy.eval())` (Recall that properties don't need to be called, so `x.lazy` returns the result of calling the `TimeSeries.lazy(self)` function, which was decorated with `@property` .)

This adds a single extra layer of laziness indirection.

You should probably check that running your `lazy` -fied `TimeSeries` object works with the `check_length` example above.

Back to top ↑

A final note:

In order to debug this, it might be useful to inject a `print` statement or two into your `@lazy` -decorated and undecorated functions. You should be able to see the difference in when these functions get called: the undecorated versions should print immediately when the expression is evaluated, whereas the decorated versions won't print until after you call `eval` on the thunk they produced.

Good luck!