Non-uniform Time and Lazy Functions

- Non-uniform time series
  - Modifying your storage
  - Fleshing out the interface
  - Linear interpolation
- Lazy evaluation
  - A thunk class
  - A lazy decorator
  - Recursive evaluation
  - Lazy TimeSeries

This is a group lab. Please work with your team.

There is no google form for this, as it 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 only dealt with a single vector of information. This is 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.

Your current code uses *implicit* samples: you assume that the values are observations at regular intervals of some arbitrary unit. Your first task will be to switch to *explicit* samples, recording a value and a time point for every observation.

One last note: 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 `TimeSeries` class, with two goals:

- We want you to store two arrays internally. 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. We'll be merging your two classes and only keeping this one.

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

```
def __init__(self, times, values):
  ...
```

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. However, indexing on your class will be done with time points, not integers. Another way to think about this is that a time series is really a map or discrete function: for a set of points in the domain, it assigns a value in its range. In this respect, your class behaves like a `dict` : given a time point, you can't implicitly know what the next time point is.

Let's implement that behavior now...

Back to top ↑

Fleshing out the interface

Recall that indexing is accomplished through `__getitem__` and `__setitem__` . We'd like you to update these two functions such that instead of just getting or setting the *nth* element, you retrieve or update the value in the `values` array corresponding to same location as wherever the user's key is found in the `times` array. It's a little easier with an example:

```
# Create a non-uniform TimeSeries instance:
a = TimeSeries([1, 1.5, 2, 2.5, 10], [0, 2, -1, 0.5, 0])
# Set the value at time 2.5
a[2.5] == 0.5
# Set the value at time 1.5
a[1.5] = 2.5
# This should return an error, because there is no time point at t=0
a[0]
```

You might be tempted to use a dictionary to do this, but remember that we asked you to implement the times and values storage as numpy arrays internally. This means that in order to get or set a particular element, you'll need to find the location user's time in the times array, then use that location to get the corresponding value from the values array.

Once you've got that done, we'd like you to implement the following special methods as well:

- `__contains__` , which takes a time and returns true if it is in the times array
- `__len__` , as it sounds, and you can assume (or assert) that both the times and values arrays are the same length
- `__iter__` , iterates over the *values* in your time series. So for instance, `[v for v in TimeSeries([0,1,2],[1,3,5])]` should return `[1,3,5]` .
- `__str__` , please make sure it is still an abbreviating version

And finally some extra, non-special methods:

- `values` , which takes no arguments and returns a sequence of the values
- `times` , which takes no arguments and returns a sequence of the times
- `items` , which takes no arguments and returns a sequence of (time, value) tuples

Please add docstrings to the initializer, `__str__` , and wherever you see fit.

Back to top ↑

Linear interpolation

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.times()) == TimeSeries([2.5,7.5], [1.5, 2.5])
# Boundary conditions
a.interpolate([-100,100]) == TimeSeries([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 a 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

Now we'll do something a little trickier: we're going to implement a form a lazy evaluation, as briefly covered in lecture.

Lazy evaluation is a deceptively simple concept: if I give you some expression like `(a+b)*c`, normal "eager" languages will compute the value of the expression and return it. A lazy evaluation of that expression doesn't return the value, it returns a thing that represents what the answer will be, should you ever ask for it. But *until* you actually ask that thing what the answer was, no computation is actually done.

Why is this useful? It can be for a number of reasons:

- If you're not sure that you're actually going to need the answer, lazy evaluation avoids unnecessary computation.
- It can be easier to express infinitely-large computations in lazy systems, since you can describe the computation without having to actually compute infinitely many values.
- You can modify the expression before you actually evaluate it. (We'll return to this idea later in the course.)

Back to top ↑

A thunk class

As we mentioned, our lazy expressions are not going to return normal values. For instance, adding two numbers won't return a number, it will return an object that represents the result of the computation, which you can ask for later.

This thing is sometimes called a thunk (https://en.wikipedia.org/wiki/Thunk) or future (https://en.wikipedia.org/wiki/Futures_and_promises). While they can be implemented a number of ways, we're going to use a class.

Create a new file called `lazy.py`. 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.

Eventually, when we do something like `(a+b)*c`, the value of the expression will end up being an instance of this thunk class.

Back to top ↑

A lazy decorator

Now comes the interesting part. We'd like to be able to turn *any* function lazy. So let's create a temporary test file with the following:

```
import timeseries as ts
import lazy

def add(a,b):
  return a+b
def mul(a,b):
  return a*b
```

We want to create a version of `add` which, instead of actually adding `a` and `b`, returns a `LazyOperation` thunk with represents what *would* happen if we added `a` and `b`. This probably sounds harder than it is, and it actually doesn't involve rewriting `add`.

The trick is to realize that the notion of what the answer *would* be is the same as saying "don't evaluate this function just yet; instead, save it and all of its arguments so I can evaluate it later if I want." This is one way of implementing lazy evaluation.

Let's be more clear: you created a class which (right now) just stores a function and a set of arbitrary positional and keywords arguments. This is exactly what you need to know if you want to evaluate that function later! So we want to write a decorator which turns `add` into a function which returns a `LazyOperation` with all of its information stored inside, similar to this:

```
@lazy
def lazy_add(a,b):
  return a+b

lazy_add(1,2) == LazyOperation( old_function, args=[a,b], kwargs={} )
```

Where `old_function` is the undecorated version of `lazy_add`. (Remember that decorators are just functions which take a function as an argument, return a new function, and bind that new function to the original name.)

Write a `@lazy` decorator now.

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 ↑

Recursive evaluation

You're probably wondering what use this is. How do we actually use these?

The answer is an `eval` method for `LazyOperation`. This is the mechanism for us to ask for an actual answer. In our example above:

```
lazy_add(1,2).eval() == 3
```

`eval` takes no arguments: it has all the information it needs. Its behavior is as follows:

- First, transform the arguments: for all the positional and keyword arguments stored in the `LazyOperation`, if the argument is an instance of a `LazyOperation`, call `eval` on *it*. If it's anything else, do nothing.
- Second, call the stored function on the transformed arguments and return the value.

That's it!

So what's actually going on here? Why do we need to recursively call `eval`? And why only on some arguments?

Let's use an example:

```
thunk = lazy_mul( lazy_add(1,2), 4)
thunk.eval()
```

In the first line we are crafting a chain of `LazyOperation`s. The expression `lazy_add(1,2)` returns an instance of `LazyOperation`. (1 and 2 are stored as its arguments.) Then we create another thunk in `lazy_mul`, and the two stored arguments are the first `LazyOperation` instance and the number 4. So something like this:

```
thunk = LazyOperation( body_of_lazy_mul, LazyOperation( body_of_lazy_add, 1, 2 ), 4 )
```

Now we call `eval` on `thunk`. `eval` will first recursively call `eval` on its `LazyOperation`-typed arguments (of which there is only one: the result of `lazy_add`). This recursive invocation does the same, but it has no `LazyOperation` arguments, so it just calls its stored function (the body of the original `lazy_add` function: `a+b`) on its arguments 1 and 2. The result of this is a number: 3. Now the first `eval` function can complete, and it evaluates *its* stored function (the body of the original `lazy_mul` function) on its new, transformed arguments: 3 and 4. This returns 7.

If you've done everything right, you should be able to run this example.

Back to top ↑

Lazy TimeSeries

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
```

But it turns out that later in the course we'll need a different mechanism.

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. Again, it's not strictly required now, but it will be useful later.

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!