Numerical Methods with Functional Programming in Python*

Paul Downen

CIS425 – December 5, 2019

Many numerical methods are expressed as a series of approximations; each one a little more accurate than the one that came before. Since the approximations can usually always be made more accurate by some sort of improvement, these series are *infinite*. That way, an answer can be calculated to any desired level of accuracy, in principle, by searching far enough into the infinite series of approximations to find one that is close enough to the real answer.

Unfortunately, translating this kind of numerical method to a computer algorithm quickly runs into a snag: a computer with finite memory cannot store an infinite series. Because of this mismatch between the mathematics and the machine, the programmer must encode the method into some finite representation. In standard practice, the encodings will be far removed the original description of the method, involving (many several nested) loops with intricate control flow and termination conditions.

But there is another way! Instead of twisting the infinite series of approximations into some sort of loop, we can represent infinite series of approximations directly as an ordinary object, keeping the structure of the program much closer to that of the description. To do so, we will be using the following techniques from functional programming:

- Using on data and objects which are *constant* and do not change their value or behavior over time.
- Treating functions as data which can be stored in objects and passed as parameters to other functions.
- Manipulating data structures containing information that is computed on-demand, rather than at the time of creation.

*This exercise set is based on Section 4 from John Hughes' excellent paper "Why Functional Programming Matters" (`https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf`). Hughes' original paper expresses its programs in Miranda, a lazy, functional programming language. The exercises here (attempt to) preserve the main ideas as much as possible while translating the programs to Python, an eager, object-oriented language.

Remark 1. The code to follow will occasionally make use of *anonymous functions*. This is just another way to refer to a function without having to first give it a name. Anonymous functions in Python are written as:

```
lambda x, y, z: # some expression involving x, y, and z
```

which describes (in this case) a function of three parameters `x`, `y`, and `z`. For example, the following two definitions of the `square` function are the same:

```
def square(x):
    return x*x

square = lambda x: x*x
```

In the first case, a function named `square` is defined using the normal `def`inition syntax. In the second case, the variable `square` is assigned a value, and that value is the function which squares its argument.

1 Infinite Streams

An infinite stream $a_0, a_1, a_2, \ldots$ cannot be stored in its entirety as a list in a computer, since there is not enough space to hold the whole thing (let alone taking eternity to do so). So instead, we will have to come up with a type of object that somehow *hides* the stream itself, and gives us access to its information through a specific interface. The interface for streams includes two methods: the `head` of a stream is its first element, and the `tail` of a stream is the remaining stream with the head remove. In other words, the interface for a stream object $a_0, a_1, a_2, a_3, a_4, \ldots$ is expressed by the following two equations:

$$(a_0, a_1, a_2, a_3, a_4, \ldots).\texttt{head}() = a_0$$
$$(a_0, a_1, a_2, a_3, a_4, \ldots).\texttt{tail}() = a_1, a_2, a_3, a_4, \ldots$$

To represent these objects in Python, we can use an (abstract) class `Stream`. Every `Stream` object must respond to the above two `head` and `tail` methods.[1]

```
class Stream:
    # head: Stream -> elem
    # tail: Stream -> Stream
```

Notice how the `head` and `tail` methods—the most important part of the stream interface—are not given any definition in the generic `Stream` class! That's because each more specific type of `Stream` will define its own specialized responses to these two methods. The most basic type of stream is one that repeats the same element over and over, *ad nauseam*.

$$\texttt{Repeat}(a) = a, a, a, a, \ldots$$

[1] In the template script, the `Stream` class includes some extra methods that are defined in terms of `head` and `tail`. These extra methods it a little easier to work with `Stream`s in Python by enabling the use of special syntax (like `for` loops and indexing $s[i]$) and printing something actually useful in the interactive prompt.

To define what a `Repeat` stream is in Python only requires explaining what is the `head` and `tail` of `Repat`(a), which should obey the equations:

$$\begin{aligned} \texttt{Repeat}(a).\texttt{head}() &= a \\ \texttt{Repeat}(a).\texttt{tail}() &= \texttt{Repeat}(a) \end{aligned} \tag{1}$$

These equations can be translated to the following `Repeat` class, which is a subclass of `Streams`:

```
class Repeat(Stream):
    def __init__(self, a):
        self.state = a

    def head(self): return self.state
    def tail(self): return Repeat(self.state)
```

The parameter `self` refers to the object itself, and the element `a` of `Repeat`(a) is stored within the object as `self.state`, which is set during initialization in `__init__`. Therefore, the parameter a from the equations Eq. (1) are replaced by `self.state` in the class definition.

`Repeat` streams can be made more general by repeatedly applying some function `f` to generate different elements of the stream, as in:

$$\texttt{Repeat}(a, f) = a, f(a), f(f(a)), f(f(f(a))), \ldots$$

The i^{th} element of the `Repeat`(a, f) stream is the i^{th} iteration of f on a. This stream obeys the following equalities on the `head` and `tail` methods:

$$\begin{aligned} \texttt{Repeat}(a, f).\texttt{head}() &= a \\ \texttt{Repeat}(a, f).\texttt{tail}() &= \texttt{Repeat}(f(a), f) \end{aligned}$$

This is a generalization previous definition because the simpler `Repeat`(a) can be recovered by taking f to be the identity function `lambda` x: x which just returns its parameter unchanged

$$\texttt{Repeat}(a) = \texttt{Repeat}(a, \texttt{lambda}\ x\text{: } x)$$

and the previous definition of the `Repeat` class is updated as follows:

```
class Repeat(Stream):
    def __init__(self, a, f=lambda x: x):
        self.state = a
        self.step = f

    def head(self): return self.state
    def tail(self): return Repeat(self.step(self.state), self.step)
```

In the above method definition `def __init__(self, state, step=lambda x: x):`, the syntax `step=lambda x: x` gives a default value for the `step` parameter, which happens to be the identity function, if one is not provided.

Exercise 1. Two example streams are the constant stream of all zeroes, and the stream of all natural numbers:

$$\texttt{zeroes} = 0, 0, 0, 0, 0, \ldots$$
$$\texttt{nats} = 0, 1, 2, 3, 4, \ldots$$

Generate these two streams in Python using the `Repeat` class constructor.

Exercise 2. Another way to generate a stream is to modify an existing one by applying a function to each element. Given a stream a_0, a_1, a_2, a_3, a_4, `Map`ping the function f over it is:

$$\texttt{Map}((a_0, a_1, a_2, a_3, a_4, \ldots), f) = f(a_0), f(a_1), f(a_2), f(a_3), f(a_4), \ldots$$

According to the above definition, the `head` and `tail` methods of a `Map` stream behave as follows:

$$\texttt{Map}((a_0, a_1, a_2, a_3, \ldots), f).\texttt{head}() = f(a_0)$$
$$\texttt{Map}((a_0, a_1, a_2, a_3, \ldots), f).\texttt{tail}() = \texttt{Map}((a_1, a_2, a_3, \ldots), f)$$

Now, fill in the following class definition for `Map` in Python that obeys the above two equations:

```
class Map(Stream):
    def __init__(self, stream, f):
        self.stream = stream
        self.trans = f

    def head(self): # FILL IN
    def tail(self): # FILL IN
```

Example 1. Here is a use of `Map` to create a stream of squares; one for each natural number:

```
squares = Map(nats, lambda x: x*x)
```

The `squares` object represents the following infinite sequence:

$$\texttt{squares} = 0, 1, 4, 9, 16, \ldots$$

Exercise 3. Yet another way to generate a stream is to combine two streams, pointwise. Given two streams $a_0, a_1, a_2, a_3, \ldots$ and $b_0, b_1, b_2, b_3, \ldots$, `Zip`ping the two together is:

$$\texttt{Zip}((a_0, a_1, a_2, a_3, \ldots), (b_0, b_1, b_2, b_3, \ldots)) = (a_0, b_0), (a_1, b_1), (a_2, b_2), (a_3, b_3), \ldots$$

As with `Repeat`, `Zip` can be generalized to take an arbitrary binary function f used to combine the two elements, instead of pairing them together, as follows:

$$\texttt{Zip}((a_0, a_1, a_2, \ldots), (b_0, b_1, b_2, \ldots), f) = f(a_0, b_0), f(a_1, b_1), f(a_2, b_2), \ldots$$

The simpler behavior is recovered by setting f to be the function just returns the pair of its two arguments:

$$\texttt{Zip}((a_0, \dots), (b_0, \dots)) = \texttt{Zip}((a_0, \dots), (b_0, \dots), \texttt{lambda}\ x, y\text{: } (x, y))$$

According to the above definition of `Zip`, the `head` and `tail` methods of a `Zip` stream behave as follows:

$$\texttt{Zip}((a_0, a_1, a_2, \dots), (b_0, b_1, b_2, \dots), f).\texttt{head}() = f(a_0, b_0)$$
$$\texttt{Zip}((a_0, a_1, a_2, \dots), (b_0, b_1, b_2, \dots), f).\texttt{tail}() = \texttt{Zip}((a_1, a_2, \dots), (b_1, b_2, \dots), f)$$

Now, fill in the following class definition for `Zip` in Python that obeys the above two equations:

```
class Zip(Stream):
    def __init__(self, l, r, f=lambda x, y: (x,y)):
        self.left = l
        self.right = r
        self.combine = f

    def head(self): # FILL IN
    def tail(self): # FILL IN
```

2 Newton-Raphson Square Roots

Newton's method for finding the square root of n is to begin with some initial approximation a_0 of $\sqrt{n}$, and then generating increasingly better approximations as follows:

$$\begin{aligned} a_0 &= \text{some initial guess} \\ a_{i+1} &= \frac{a_i + \frac{n}{a_i}}{2} \end{aligned} \tag{2}$$

If this series of approximations converges to a limit a, so that $a = \frac{a+n/a}{2}$ then

$$\begin{aligned} 2a &= a + \frac{n}{a} \\ a &= \frac{n}{a} \\ a^2 &= n \\ a &= \sqrt{n} \end{aligned}$$

That is to say, the limit of the series $a_0, a_1, a_2, \dots$ described by Eq. (2) is the square root of n. Since the series converges on the limit, the difference between consecutive elements a_i and a_{i+1} represents the error: the smaller $|a_i - a_{i+1}|$ is, the more accurate a_{i+1} is with respect to the real answer $\sqrt{n}$. That means that an approximation of $\sqrt{n}$ can be calculated to any desired accuracy by going deep enough into the series until the error is within an acceptably small margin.

2.1 Generating approximations

Exercise 4. We can represent the infinite series described in Eq. (2) as a `Stream` object. First, define a Python function `next_root` that calculates the next number in the series given the current one a, so that:

$$\texttt{next\_root}(n, a) = \frac{a + \frac{n}{a}}{2}$$

```
def next_root(n, a): # FILL IN
```

Next, define a Python function `sqrts` that generates the series Eq. (2) by `Repeat`ing the `next_root` function, so that:

$$\texttt{sqrts}(a_0, n) = a_0, \texttt{next\_root}(n, a_0), \texttt{next\_root}(n, \texttt{next\_root}(n, a_0)), \dots$$

```
def sqrts(a0, n): # FILL IN
```

2.2 Selecting an answer

Exercise 5. To determine the error of approximations, we need to compare them side-by-side. So it would be convenient to organize the stream so that consecutive elements are given side-by-side. Define a Python function `by_twos` that groups together consecutive elements of a stream into pairs, like so:

$$\texttt{by\_twos}((a_0, a_1, a_2, a_3, a_4, \dots)) = (a_0, a_1), (a_1, a_2), (a_2, a_3), (a_3, a_4), \dots$$

```
def by_twos(stream): # FILL IN
```

HINT: `Zip`ping might be helpful here.

Next, use `by_twos` to find the first pair of elements a_i and a_{i+1} that are within a given error value ε and return the second (more accurate) one, as follows:

$$\texttt{within}(\varepsilon, (a_0, a_1, \dots)) = a_{i+1} \qquad \text{if } a_i - a_{i+1} \leq \varepsilon$$

```
def within(eps, stream): # FILL IN
```

HINT: If you use the special `__iter__` method of `Stream` from the template script, a `for` loop can inspect the elements of a stream one at a time as in:

```
for elem in stream:
    # do something with each elem
```

Exercise 6. Now, use `sqrts` and `within` to define a square root approximation function.

```
def sqrt(a0, eps, n): # FILL IN
```

2.3 Another selection criteria

Exercise 7. Another way to measure the error between two approximations is to compare their ratios, rather than their difference. That is to say, instead of trying to get $|a_i - a_{i+1}|$ as small as possible, we could instead try to get a_i/a_{i+1} as close to 1 as possible. This measure of error can give better results for very small numbers where the difference is small (because the numbers are already small) but the ratio might be huge.

Fortunately, since the generation of approximations is separate from the selection of answers, we only need to define a new selection function to replace `within`. First, define the ratio-based selection function `relative` similar to `within` that finds the first pair of elements a_i and a_{i+1} satisfying the following equation:

$$\texttt{relative}(\varepsilon, (a_0, a_1, \dots)) = a_{i+1} \qquad \text{if } \left|\frac{a_i}{a_{i+1}} - 1\right| \leq \varepsilon$$

```
def relative(eps, stream): # FILL IN
```

Next, write a relative square root function `rsqrt` that is the same as `sqrt` but uses `relative` in place of `within`.

```
def rsqrt(a0, eps, n): # FILL IN
```

3 Numerical Differentiation

Derivatives are another appropriate problem to solve with a series of approximations. The very definition of involves eliminating the error of an estimation to find the real answer!

$$\frac{d}{dx}f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3}$$

The derivative of f at x can be calculated numerically by picking a value for the distance h that is small, but not actually 0. The smaller the value for h ,the more accurate the calculation. So we can rephrase the definition of a derivative to be the limit of the calculation on a sequence H of ever decreasing distances.

$$H = h_0, h_1, h_2, \dots \qquad (\text{where } h_i > h_{i+1})$$
$$\frac{d}{dx}f(x) = \lim_{h \in H} \frac{f(x+h) - f(x)}{h}$$

3.1 Generating approximations

Exercise 8. An appropriate choice of distances to begin with some initial h, and then halve each successive distance. Write the Python function `halves` that generates this stream starting from an initial h.

$$\texttt{halves}(h) = h, \frac{h}{2}, \frac{h}{4}, \frac{h}{16}, \frac{h}{32}, \dots$$

```
def halves(h): # FILL IN
```

Next, write the `easy_diff` function which calculates a derivative estimate of $f(x)$ from Eq. (3) for a given h.

$$\texttt{easy\_diff}(h, x, f) = \frac{f(x+h) - f(x)}{h}$$

```
def easy_diff(h, x, f): # FILL IN
```

Finally, combine the above two functions together to generate a stream of derivative estimates for a distance shrinking by half each time.

$$\texttt{differentiate}(h_0, x, f) = \texttt{easy\_diff}(h_0, x, f), \texttt{easy\_diff}(h_0/2, x, f), \ldots$$

```
def differentiate(h0, x, f): # FILL IN
```

Example 2. Here are some numeric differential approximation streams that you can check out. Compare how fast each of the different streams converge on the right answer.

```
const = differentiate(1, 10, lambda x: 5)
lin   = differentiate(1, 10, lambda x: 12*x)
quad  = differentiate(1, 10, lambda x: x**2)
expn  = differentiate(1, 10, lambda x: 2**x)
dexpn = differentiate(1,  5, lambda x: 2**(2**x))
trig  = differentiate(1, 10, lambda x: sin(x))
```

HINT: The `sin` function can be imported from the `math` module.

3.2 Selecting an answer

Exercise 9. To calculate the derivative of a function, we have to pick one answer out of the many options that is within our tolerance for error. Thankfully, we have already written such a selection function: `within`! Just like in Exercise 7 where the selection criteria was swapped out, so too can the generating stream be swapped out. Combine `differentiate` and `within` to approximate the differential of `f` at `x` within an error of `eps` by beginning at a distance of `h0`.

```
def diff(h0, eps, x, f): # FILL IN
```

3.3 Improving convergence

One potential problem with this approach to numeric differentiation is that the approximations can converge on the real answer rather slowly, so that finding an answer within a desired tolerance for error can require too many calculations. Fortunately, some ingenuity can be applied to make the approximations converge

much faster. Each approximation is an alteration of the real answer with some error, and the error depends on h:

$$\text{the real answer} \;+\; \text{an error involving } h$$

It turns out that error term is approximately a multiple of h^n, for some n. Call the real answer A and the error multiplier B, so that for two consecutive approximations a_i and a_{i+1}:

$$a_i = A + (2h)^n B \qquad\qquad a_{i+1} = A + h^n B$$

Solving the system of equations gives:

$$2^n a_{i+1} - a_i = (2^n A - A) + (2^n h^n B - (2h)^n B) = (2^n - 1)A$$

This leaves us with the "real" answer A with the error term eliminated:

$$A = \frac{2^n a_{i+1} - a_i}{2^n - 1} \tag{4}$$

Exercise 10. Write a Python function `elim_error` that eliminates the error term of order n from a stream of approximations by using the "real" answer from Eq. (4).

$$\texttt{elim\_error}(n, (a_0, a_1, a_2, \dots)) = \frac{2^n a_1 - a_0}{2^n - 1}, \frac{2^n a_2 - a_1}{2^n - 1}, \dots$$

```
def elim_error(n, stream): # FILL IN
```

HINT: Notice that each element of the stream generated by `elim_error` refers to two consecutive elements of the given stream, like in `by_twos`.

Exercise 11. To actually eliminate the error from a stream, we need to know the order n of the error. This is hard to know ahead of time, but easy to measure on the stream with the following calculation:

$$\texttt{order}(a_0, a_1, a_2, \dots) \approx \log_2 \left| \frac{a_0 - a_2}{a_1 - a_2} - 1 \right| \tag{5}$$

Write the Python function `order` that measures the order n of a stream using the calculation from Eq. (5), as well as the function `improve` that eliminates the error of the appropriate order of a stream.

```
def order(stream): # FILL IN

def improve(stream): # FILL IN
```

Check the `improve`d versions of the derivative streams from Example 2 to see the impact of `improve`ment on approximations.

HINT1: The `log` function can be imported from the `math` module. The base-2 logarithm $\log_2(x)$ is $\texttt{log}(x, 2)$.

HINT2: A convenient way of taking apart a fixed-length list in python is to use a pattern-matching assignment like:

```
[a, b, c] = [1, 2, 3]
```

which assigns `1` to `a`, `2` to `b`, and `3` to `c`. The `take`(n) method of the `Stream` class in the template script returns a list of the first n elements. So the first three elements of a `stream` can be easily accessed with one line:

```
[a, b, c] = stream.take(3)
```

HINT3: It's possible that the first few elements in the given stream are so close together than their difference is effectively 0. In that case, there is no `order` since the calculation causes an exception, so the only possible "improvement" is to remove the first element (the worst approximation) from the stream.

3.4 Super improvement

The method in Eq. (4) for eliminating some error from a stream of approximations works because the variable h which controls the error in each approximation is half as big as the one that comes before it. But note that an `improve`d stream is also such a "halving" approximation series! That means we can `improve` on an already `improve`d stream (as in `improve`(`improve`(`s`))) to eliminate another error term and converge *even faster*. And this can be `improve`d again, and again.

How much `improve`ment is enough to effectively "jump" to the answer in only a few hops depends entirely on the approximation stream itself, and is difficult to predict ahead of time. Instead, we can use the following "`super_improve`" function that repeatedly `improve`s a stream, picking the second approximation from each `improve`ment.

```
def super_improve(stream):
    return Map(Repeat(stream, improve), lambda s: s.tail().head())
```

The i^{th} element of a `super_improve`d stream comes from the i^{th} `improve`ment of the original approximation stream (where s.`tail`().`head`() is $s[1]$).

$$\texttt{super\_improve}(s) = s[1], \texttt{improve}(s)[1], \texttt{improve}(\texttt{improve}(s))[1], \ldots$$

Exercise 12. Update your `diff` function from Exercise 9 to use a `super_improve`d stream of approximations. Check the `super_improve`d versions of the derivative streams from Example 2, and compare them with the original and singly-`improve`d versions. Which ones converge noticeably faster, and which ones are about the same as a single application of `improve`?