Functional Programming: Activities
Statistics 650/750
Week 8 Tuesday

Alex Reinhart and Christopher Genovese

17 Oct 2017

Announcements

- You should have project drafts Thursday for peer review!
  - You don't need to open a pull request yet; we'll handle that Thursday
- Final submissions for TA approval will be **Thursday, October 26**
- You will have a chance to revise your projects after TA grading
- Status emails with important dates going out today

Map, Reduce, and Filter

Last Thursday we talked about functional programming and its basic principles: first-class functions, immutable data, and closures.

With first-class functions, we can write functions that take other functions as *arguments*, or even return new functions to do new things. With immutable data, we make operations easier to understand and to run simultaneously. With closures, functions can have private state, letting them encapsulate data.

(You can even build an object-oriented programming system in a language without objects by being sufficiently clever with closures.)

Today let's look at a few key higher-order functions – functions which take functions as arguments – we can use for many things.

Map

The `map` operation operation takes a function and a collection and calls the function for each successive element of the collection, producing a new collection out of the results. For example, in R:

```r
square <- function(x) x^2
Map(square, 1:10)   #=>  list(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

or in Python:

```python
map(lambda x: x*x, range(1,11))  #=> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

using an *anonymous* function, declared with the `lambda` keyword, that has no name.

`map` can usually do more than this. In most languages, it can take *multiple sequences* and a function that takes multiple arguments. It then calls the function with successive elements from each sequence as its arguments, producing a sequence of the results. For example, in R

```
Map(`-`, 1:10, 10:1)   #=>  list(-9, -7, -5, -3, -1, 1, 3, 5, 7, 9)
```

and in Python

```
map(lambda x, y: x - y, range(1, 11), range(10, 0, -1))
   #=> [-9, -7, -5, -3, -1, 1, 3, 5, 7, 9]
```

We can use `map` to replace a common pattern we see over and over in code processing data:

```
transformed_data <- c()

for (ii in 1:length(data)) {
    transformed_data <- c(transformed_data, do_stuff_to(data[ii]))
}
```

The `map` operation expresses our meaning more clearly and concisely, and is more efficient to boot (since it does not repeatedly copy the `transformed_data`).

Filter

The `filter` operation takes a predicate (essentially a boolean function) and a collection. It calls the predicate for each element, and returns a new collection containing only those elements for which the predicate returns true. For example, in R.

```
is_odd <- function(x) { (x %% 2 != 0) }

Filter(is_odd, 1:10) #=>  c(1, 3, 5, 7, 9)
Filter(is_odd, as.list(1:10)) #=>  list(1, 3, 5, 7, 9)
```

Notice that each result is the same type as the collection `Filter` was given. In Python,

```
filter(lambda x: x % 2 != 0, range(1, 11)) #=> [1, 3, 5, 7, 9]
```

We can always combine a `map` with a `filter`, applying a function to only those elements matching a predicate. The composability of these operations is a major advantage, and much easier to deal with than building complicated loops over data manually.

Reduce

The `reduce` operation, also sometimes called `fold`, is a much more general way to process the elements of a sequence. We could use it to build `map`, `filter`, or many other interesting operations.

`reduce` takes as arguments a function, an *accumulator*, and a sequence. The function takes two arguments – the accumulator and one element of the sequence – and returns an updated accumulator. The function is first passed with the initial accumulator and the first element of the sequence, returning a new accumulator which is passed to the function again with the second element of the sequence, and so on. The final value of the accumulator is returned by `reduce`.

For example, in R:

```r
parity_sum <- function(accum, element) {
    if ( element %% 2 == 0 ) {
        list(accum[[1]] + element, accum[[2]])
    } else {
        list(accum[[1]], accum[[2]] + element)
    }
}

Reduce(parity_sum, 1:10, list(0,0))  #=> list(30, 25)
```

In Python:

```python
def parity_sum(acc, x):
  even, odd = acc
  if x % 2 == 0:
      return [even + x, odd]
  else:
      return [even, odd + x]

reduce(parity_sum, range(1,11), [0,0])  #=> [30, 25]
```

Activities

Let's do some activities to get used to the idea of using higher-order functions like `map`, `filter`, and `reduce`.

You can work through these activities (or some subset) in any order that interests you. You can work together if you'd like.

However, you may **not** use loops of any form: no `for` or `while` loops can appear anywhere in your code. Anywhere you'd want to loop over a sequence, use `map`, `filter`, or `reduce` instead.

Write a few tests for each function. These activities are best done in a language with first-class functions, like R, Python, any Lisp variant, or Haskell. They are possible in C++ and Java, but not nearly as convenient or simple.

These lecture notes are on GitHub (`documents/Lectures/Week8T`) for your reference throughout the activity.

Tasks:

1. Write a function which sums a list (or vector) of numbers.

2. Write a function that takes a list/vector of integers and returns a list containing *only* those elements for which the *preceding* integer is negative.

3. Write a function that takes a string and returns true if all square `[ ]`, round `( )`, and curly `{ }` delimiters are properly paired and legally nested, or returns false otherwise. Other characters should be ignored.

   For example, `[(a)]{[b]}` is legally nested, but `{a}([b)]` is not.

4. Write a function `roman` that parses a Roman numeral string and returns the number it represents. You can assume that the input is well-formed, in upper case, and adheres to the "subtractive principle". You need only handle positive integers up to MMMCMXCIX (3999), the largest number representable with ordinary letters.

   A reference on Roman numerals, including the "subtractive principle": `http://www.numericana.com/answer/roman.htm#valid`

   Be sure to test carefully – Roman numerals are tricky.

5. Write a function `chain` which takes as its argument a list of functions, and returns a new function which applies each function in turn to its argument.

   For example,

```
import math

positive_root = chain([abs, math.sqrt])

positive_root(-4)    #=> 2.0
```

6. Write a function `partial` that takes a function and several arguments and returns a function that takes additional arguments and calls the original function with all the arguments in order. For example,

```
foo <- function(x, y, z) { x + y + z }
bar <- partial(foo, 2)

bar(3, 4) #=> 2 + 3 + 4 = 9
```

7. Write a function `count-repeats` (or `count_repeats` or `countRepeats` as you like) and a function `run-length`, that, respectively, takes a sequence of strings and returns the number of strings that are repeated at least twice in a row (at successive indices) and takes a sequence of strings and returns the maximum number of consecutive indices for which a string is repeated. So, for `["h" "h" "t" "h" "t" "t" "t"]`, `count-repeats` would return 2 and `run-length` would return 3. Each of these functions can be implemented directly using a common third function that you write.

8. Write a function `mavg` that takes a vector of numbers, a window size, and an optional vector of weights equal in length to the window size, and returns a vector of (backward) moving averages, using the weights in the window or the reciprocal of the window size if no weights are supplied. The ith element of the result is the weighted average (using the weights) of the data at indices i, i+1, . . . , i+$\text{window}_{\text{size}}$-1. Note that the result is shorter than the data vector by window size.

   Thus, `[1, 2, 3, 4, 5, 6]`, `3`, and `[1 2 1]` would return the vector

```
[1/5 + 2*2/5 + 3/5, 2/5 + 2*3/5 + 4/5, ..., 4/5 + 2*5/5+ 6/5].
```

9. Write a function `nnk` that takes a collection of p-dimensional vectors (as a data matrix, data frame, list of vectors as you prefer), a target point, and a positive integer k, and returns the k nearest neighbors among the data to the target point.

10. Write a function `bow` that takes an input source for some document (e.g., a file, a stream/connection, a string, a url) and returns a bag-of-words for that document. (A bag, or multiset, of words is a set of the words in the document along with associated count for how many times each word appears. It is usually represented as a vector of counts, where each element of the vector represents a word, with the words in a fixed and pre-determined order.)

11. Generalize the function `bow` from the previous activity to accept another parameter `ngram`, which is a positive integer determining how many consecutive words to consider at once. The original `bow` function corresponds to `ngram = 1`. If `ngram = 2`, then document "every good student does fine every day" is converted into "words" (actually 2-grams): "every good", "good student", "student does", "does "fine", "fine every", and "every day". If `ngram = 3` in this example, we would have "every good student", "good student does", "student does fine", "does fine every", and "fine every day".

12. Write a function `reverse_map` (or `reverse-map` or `reverseMap` as you like) that takes a hash table and returns the "reversed" hash table. The keys of the reversed hash table are the values of the original hash table, and the values of the reversed hash table are either the keys of the original, or when there are multiple keys with the same value, a list of such keys.