Functional Programming
Statistics 650/750
Week 7 Thursday

Christopher Genovese and Alex Reinhart

12 Oct 2017

Announcements

- You should be making progress on your challenge projects; let us know if you have questions.
- New Object-Oriented Programming and Functional Programming problems have been pushed to the problem bank, with more to come.
- `cow-proximity` has a nice functional programming solution; try it!
- Brief comment on driver/wrapper scripts

Goals for Today

- Describe the core concepts of functional programming (FP)
- Show you some of the mechanics of using FP
- Argue for the value of "immutability" and "pure functions"
- Give you a taste of what "thinking functionally" means
- Pointing you toward FP resources in R and other languages

What takes longer than one class: the "aha" moment.
Key point:
Whatever language you use, using the ideas and methods of FP will make you a better programmer.

Review and Contrast

A fundamental part of developing software is **building effective abstractions** to model and organize the functionality the code provides.

Last time, we looked at object-oriented programming (OOP), which models problems with a collection of inter-related *entities* which have designated *behaviors* and which manage (by mutating) their own *internal state.*

The core features of OOP are:

Encapsulation keeping things on a "need to know" basis

Polymorphism interchangeability of objects based on interface

Inheritance and Composition how to build new classes

Delegation of Responsibilities each class handles its own sphere of concerns

In OOP, **the class is the fundamental unit of abstraction**.
Why OOP?

- Modularity
- Separation of Concerns
- Ease of modification where interface is shared
- Ease of testing (maybe)
- Concrete modeling abstraction
- Can be easy to reason about

Issues:

- Overhead, boilerplate, verbosity
- Emergent complexity – many classes, complex relationships
- In practice, not as modular or reusable as hoped.
- Parallelism/Concurrency is very hard
- Fundamentally prioritizes nouns over verbs. Why?
- Can be hard to reason about

Today, we consider another programming paradigm called **Functional programming (FP)** that emphasizes *verbs* over nouns.
Functional design focuses on building and combining the basic parts of a computation, createing **composable abstractions**,
computation and
Notes:

1. OOP was the clearly dominant paradigm of the 1990s and 2000s and remains entrenched, but FP has been enjoying a striking renaissance this decade.
2. FP and OOP are not in strictly in opposition; they can usefully coexist. But they do lead to different ways of thinking about computation.
3. For reference, there are many common programming paradigms. These include: procedural, object-oriented, functional, logic, event-driven, aspect oriented, and automata-based. Procedural and object-oriented are *imperative* – giving a sequence of computational steps that mutate the state of the program – while functional and logic programming (and some database query languages) are *declarative* – expressing the intent of the program without necessarily specifying the control flow. (Aspect-oriented is a bit odd.)

The Core Concepts of Functional Programming (FP)

In *functional programming* (FP), **functions are the fundamental unit of abstraction**.

> It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.
>
> – Alan Perlis

> Functional programming is a style of programming that emphasizes the evaluation of expressions rather than the execution of commands.
>
> – comp.lang.functional FAQ

Why Functional Programming?

- Composable abstractions
- Expressivity
- Easier to reason about (and thus check) a program
- Effective Concurrency and Parallelism
- Declarative structure; focus on data flow
- Avoid the complexity of mutating state
- Efficiently exploit recursive thinking

Languages:

- Key Functional Programming Languages:
  - Clojure
  - Ocaml
  - Haskell
  - Scala
  - SML (Standard ML)
  - Elm
- Languages with Significant FP Features
  - R
  - Ruby
  - Rust
  - Javascript
  - Common Lisp, Scheme, Racket
  - Julia
  - Mathematica
  - (Python)
- Languages with significant newly added FP features
  - C++
  - Java

First-Class Functions

While a precise definition is hard to pin down, one consistent requirement of FP is that **functions are first-class entities**.

An entity is "first class" if it can be:

- created on demand,
- stored in a variable or data structure,
- passed as arguments to functions,
- returned as values from functions.

For instance, integers are first-class entities:

- Create: `14`
- Store: `i = 7`
- Pass: `max(7,14)`
- Return: `max(7,14)`

Integers are "data."
Having first-class functions means that *functions are data too*.

```
chain_twice <- function(f, x) {
                  return( f(f(x)) )
               }
incr <- function(x) { return( x + 1 ) }
chain_twice(incr, 10)    #=> 12
incr_by <- function(increment) {
               return( function(x) { x + increment } )
           }
incr_by(10)(10)   #=> 20
i12 <- incr_by(12)
i17 <- incr_by(17)
i12(10) #=>22
i17(10) #=>27
chain_twice(incr_by(10), 10)   #=> 30

apply(M,  ACROSS_COLS,  function(x) { max(x[!is.na(x) && x != 999]) })

pairwise.distances(X, metric=function(x, y){ max(abs(x - y)) })

subtract.mean <- function(x)
{
    xbar <- mean(x)
    return( function(y) { return(y - xbar) } )
}
```

```clojure
(defn chain-twice
  "Compose a function with itself"
  [f]
  (comp f f))

(def incr2 (chain-twice inc))
(incr2 10)   ;=> 12

(defn incr-by [increment]
  (fn [x] (+ x increment)))

(def incr-by-10 (incr-by 10))
(def incr-by-20 (chain-twice (incr-by 10)))

(incr-by-10 10)     ;=> 20
(incr-by-20 10)     ;=> 30
```

```ruby
def call_twice(x, &f)
  f.call(f.call(x))
end

call_twice(10) {|x| x + 1}   #=> 12
```

```python
def call_twice(f, x):
    return f(f(x))
call_twice(lambda x: x + 1, 10)   #=> 12
```

These examples illustrate at least three use cases:

1. Succinct and optimizable representation
2. Parameterized strategies
3. Dynamically-defined operations

Notice that functions can be created on the fly *even without names*, what are called **anonymous functions**:

```r
integrate(function(x){x*x}, 0,  1)
  #=> 0.3333333 with absolute error < 3.7e-15
```

```clojure
(->> (sample (range 1 26) :size 77 :replacement true)
     (filter (fn [x] (pos? (mod x 5))))
     (filter #(pos? (mod %1 3)))
     frequencies)
```

Here, `filter` keeps only the elements of a sequence for which its first argument – a function – returns a truthy value. Functions such as `filter` are called **higher-order functions** is a function that takes one or more *functions as arguments* and/or *returns a function* as its result.

In R base, `Filter` is the analogous higher-order function, though there are nicer versions in several packages as we will see.

```
Filter(function(x) {x %% 3 != 0}, 1:10) #=>  1  2  4  5  7  8 10
```

Similarly in Python, the `filter` function is built-in, but comprehensions are an alternative.

Another example, in javascript:

```
// Abstracting Array Iteration
function forEach(array, itemAction) {
    for ( var i = 0; i < array.length; i++ ) {
        itemAction(array[i])
    }
}
forEach(["R", "SAS", "SPSS"], console.log);
forEach(["R", "SAS", "SPSS"], store);
forEach(["R", "SAS", "SPSS"], function(x) {myObject.add(x)});
```

Pure Functions (where possible)

A function is **pure** if it:

- always returns the same value when you pass it the same arguments
- has no *observable* side effects

Pure functions are deterministic and mathematically well-defined. They are easy to test, to reason about, to change, and to compose.

They represent reusable chunks of work that can be parceled out, allowing parallel/concurrent processing and laziness.

They are worry free, and there are huge benefits to using pure functions whenever possible.

Examples: pure and impure

```
pure <- function(x) {
    return( sin(x) )
}

global.state <- 10

not.pure <- function(x) {
    return( x + global.state )
}

also.not.pure <- function(x) {
    return( x + random_real() )
```

```r
}

another.not.pure <- function(x) {
    save_to_file(x, "storex.txt")
    return( x + random_real() )
}

u <- 10
and.again <- function(x) {
    ...
    print(...)              # Input/Output
    z <- rnorm(n)           # Changing internal state
    my.list$foo <- mean(x)  # Mutating objects' state
    u <<- u + 1             # Changing out-of-scope values
    ...
}
```

And equally subtle:

```python
def foo(a):
    a[0] = -1
```

When you see a call foo(x) for an array x, can you tell what happens?

Immutable State

This is a common pattern in imperative programming:

```
a = initial value
for index in IndexSet:
    a[index] = update based on index, a[index], ....
return a
```

Each step in the loop updates – or *mutates* – the state of the object.

This is so familiar that we don't really think about it, but it can have significant costs. In general, it is hard to reason about and keep track of objects' meaning when they can mutate with abandon.

Mutation introduces a greater dependence on time and order in operations. This makes it harder to parallelize or do lazy computation.

Immutable data structures do not change once created.

- We can pass the data to anywhere (even simultaneously), knowing that it will maintain its meaning.
- We can maintain the history of objects as they transform.
- With pure functions and immutable data, many calculations can be left to when they are needed (laziness).

FP favors *immutable data*, and some FP languages have no (or severely limited) notions of assignment. Immutability changes how you think about arranging your computation.

```r
for ( i in 1:n ) {
    a[i] <- a[i] + 1
}
```

versus

```r
Map(function(x){x+1}, a)
```

```clojure
(map inc a)
```

With immutable data and pure functions, computational steps correspond to *transforming* data rather than changing existing objects.

A key to making immutable data efficient are **persistent data structures**. These are immutable data structures that use *structure sharing* to maintain changed versions with minimum overhead. Speed up: parallel computation, local transients. (See "Persistent Vectors" exercise.)

```clojure
(def counts {:a 42
             :b 12
             :c 248
             :d 0})

(assoc counts :e 10 :f 12)
  ;=> {:a 42, :b 12, :c 248, :d 0, :e 10, :f 12}
counts
  ;=> {:a 42, :b 12, :c 248, :d 0}
```

Favoring immutable data and pure functions, makes values and transformations rather than actions the key ingredient of programs. So FP prefers expressions over statements

```python
def cleaned(x):
    if is_valid(x):
        return x
    else:
        return cleanup(x)

if valid(x):
    return 4 + 7 + 20
else:
    return 4 + 10 + 20

u = 10
if valid(x):
    u = 7
return(4 + u + 20)

return 4 + (7 if valid(x) else 10) + 20
```

```clojure
(defn cleaned [x]
  (if (valid? x) x (cleanup x)))
```

Closures

Closures are functions with an environment attached. The environment is persistent, private, and hidden. This is a powerful approach for associating state with functions that you pass into other functions. (In fact, an entire OOP system could be built from closures.)

```r
counter <- function(start=0, inc=1)
{
    value <- start
    return(function() {
        current <- value
        value <<- value + inc
        return(current)
    })
}
cc <- counter(10, 2)
dd <- counter(10, 2)
cc() # 10
cc() # 12
cc() # 14...
value  # Error: object 'value' not found
dd() # 10
```

```clojure
(defn counter
  ([] (counter 0 1))
  ([start] (counter start 1))
  ([start incr]
   (let [counter-value (atom start)]
     (fn [] (swap! counter-value + incr)))))


(def counter1 (counter))
(def counter2 (counter 10 5))
(def counter3 (counter 2))

(counter1)  ;=> 1
(counter1)  ;=> 2
(counter1)  ;=> 3

(counter2)  ;=> 15
(counter2)  ;=> 20
(counter2)  ;=> 25

(counter3)  ;=> 3
```

```
(counter3)   ;=> 4
(counter3)   ;=> 5
```

Only the anonymous function returned by counter() can access that internal state: it is private and unique to each instance.

Laziness (sometimes)

Lazy evaluation (or laziness for short) means that expressions are not evaluated *until their results are needed*.

```
fib = 0 : 1 : zipWith (+) fib (tail fib)

take 10 fib   -- => [0,1,1,2,3,5,8,13,21,34]
```

```
(def fib (cons 0 (cons 1 (lazy-seq (map + fib (rest  fib))))))

(take 10 fib) ;=> (0 1 1 2 3 5 8 13 21 34)
```

Declarative Style

Declarative style emphasizes telling the computer *what* to accomplish more than telling it *how* to accomplish it.

```
tokenize <- function(line) {
    line %>%
        str_extract_all("([A-Za-z][-A-Za-z]+)") %>%
        unlist %>%
        sapply(tolower) %>%
        as.character
}
```

```
(defn tokenize [line]
  (->> line
       (re-seq "([A-Za-z][-A-Za-z]+)")
       (map lower-case)))
```

Frequently Used Higher-Order Functions

There are many commonly used higher-order functions that operate on sequences or streams of information. Reduce, map, filter, partial application, and function composition are basic ones

Reduce (aka Fold)

The **reduce** operation is a general state updater. It successively updates based on a *reducing function*: a function that takes the state and an input value and updates the state: (f state input) -> new-state.

`(reduce f initial-value sequence)`

is semantically equivalent to:

```
state = initial-value
For each element in sequence:
   state = f(state, element)
return state
```

It can often be done in parallel over chunks of input, for substantial efficiency gains. Sequences of reducing functions can be combined efficiently, using the idea of **transducers**. Type-stable operations on deeply nested data structures can be efficiently performed as well.

```
Reduce(`+`, 1:10,  0)       # => 55
```

```
(reduce + (range 1 11))    ; => 55
(reduce + 10 (range 1 11)) ; => 65
```

Map

The **map** operation transforms a sequence (or several sequences) by applying a function to successive elements

`(map f seq)` and `(map f seq1 seq2 ... seqn)`

are semantically equivalent, respectively, to:

```
seq' = empty sequence
for element in seq:
    append f(element) to seq'
return seq'
```

and

```
seq' = empty sequence
for parallel element1 in seq1, element2 in seq2, ..., elementn in seqn:
    append f(element1, element2, ..., elementn) to seq'
return seq'
```

The sequence **need not** be an array; it can be any collection that can be traversed, including

- Hash tables (key-value pairs in some order)
- Sets
- Trees and Graphs
- Database queries
- Streams and Files

In base R, there are

- `lapply`: collection, function -> list
- `sapply`: collection, function -> matrix/vector
- `apply`: array/matrix, margin, function -> matrix/vector
- `Map`: function, collections... -> list, vector, or array
- `mapply`: same as Map

```
df[] <- lapply(df, func)
```

The package `purrr` has nicer, type-stable versions of `map` and other higher-order functions. It's a nice package for standard use.

Python has a builtin `map` function, with `map(f, iterable)` and `map(f, iterable1, ..., iterablen)` analogous to the above. But list comprehensions are more often used in python.

List Comprehensions

List comprehensions specify a list with a mathematical specification giving the elements in terms of a reference set and constraints. (Dictionary/hash comprehensions and set comprehensions use the same basic notation but give different return types.)

```
[2*x + 1 for x in range(10)]   # => [1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
[2*x + 1 for x in range(10) if x % 2 == 0] # => [1, 5, 9, 13, 17]
```

```
(for [x (range 10)] (+ (* 2 x) 1))  ;=> (1 3 5 7 9 11 13 15 17 19)
(for [x (range 10) :when (zero? (mod x 2))] (+ (* 2 x) 1)) ;=> (1 5 9 13 17)
```

```
[2*x + 1 | x <- [0..9]]    -- => [1,3,5,7,9,11,13,15,17,19]
[2*x + 1 | x <- [0..9], mod x 2 == 0]    -- => [1,5,9,13,17]
```

In the clojure and haskell examples, these sequences are lazily evaluated.

From Reduce

Map can be implemented from reduce with an initial state the empty list [] and a reducing function that appends `f(input)` to the current state.

This is semantically equivalent to:

```
(reduce (fn [state input] (conj state (f input))) [] sequence)
```

Filter and Remove

The **filter** operation extracts a subsequence of the input for which a given *predicate* returns true. The **remove** operation extracts the complementary subsequence.

`(filter predicate seq)` is semenatically equivalent to:

```
seq' = empty-sequence
for element in seq:
    if predicate(element):
        append element to seq'
return seq'
```

In R: `Filter` is in base R, but there are nice versions in the `purrr` package as well.
In python: filter exists but list comprehension combines map and filter in a simple way:

```
[x*x + 1 for x in range(10) if x % 2 == 0] #=> [1, 5, 17, 37, 65]
```

From Reduce

Filter can also be defined from reduce using a reducing function that appends new input to the updated state only if the predicate is true on that input.

This is semantically equivalent to:

```
(reduce (fn [state input] (if (predicate input) (conj state input) state))
        [] sequence)
```

Composition and Chaining

What do you think this gives? (The `comp` function represents function composition, which passes the return value of one function as the argument of the next. The `first` function returns the first element of a sequence or stream.)

```
(comp even? inc first)
```

```
(def f (comp even? inc first))

(f [1 3 4])   ;=> true
(f [0 1 2 3]) ;=> false
```

Exercise: write the function `compose(f1, ..., fn)` in R, or your favorite language. This returns the function f1 composed with f2 composed with . . . fn; it should take arbitrary arguments and pass those first to fn.

```
first <- function(seq) { seq[1] }
inc <- function(x) { x + 1 }
is_even <- function(x) { x %% 2 == 0 }

compose(is_even, inc, first)
```

Chaining is similar but slightly more general in that we have control over *which argument* the value is inserted into.

```clojure
(defn- finalize-config
  "Normalizes and checks the current snapshot config map.
   Throws an IllegalArgumentException if the config is
   invalid for any reason."
  [config]
  (or (-> config
          cleanup-moves
          cleanup-prior
          cleanup-policies
          valid-config?)
      (throw (IllegalArgumentException.
               "Invalid snapshot specification"))))
```

```javascript
$('#my-div')
  .css('background', 'blue')
  .height(100)
  .fadeIn(200);
```

Partial Application and Currying

Partial application takes a function of n arguments and fixes the first k of those arguments, returning a function of n - k arguments.

```clojure
(def f  (partial + 5))

(f 6)  ;=> 11
(f -5) ;=> 0
```

Currying is the decomposition of a multi-argument function into a sequence of functions that each take one argument and return another function (if more arguments needed) or the result. In the Haskell language, *all* functions can be automatically curried.

```haskell
div 11 2  -- => 5
let f = div 11
f 2       -- => 5
```

A Brief Clojure Primer

Why clojure?

- A powerful and elegantly designed language with a simple core and extensive libraries.
- Immutability, concurrency, rich core data types

- Same language runs on the JVM and the browser
- A significant *data-science* footprint
- Cutting-edge ideas: spec, transducers, metadata, async, . . .
- All the power of lisp: macros, REPL-driven development, single and multiple dispatch, destructuring, . . .
- Functional design, testing, rapid development
- Great community

Simple Syntax

A clojure form (expression) is either:

1. A *literal* piece of data

   - Numbers 42, -1.23, 8/7 (rational), BigInteger, . . .
   - Boolean Values `true` and `false`
   - Null value `nil`
   - Strings `"foo bar"` and characters `\c`
   - Symbols `'foo` or Keywords `:bar`
   - Regular expressions `#"[A-Z][a-z]*"`
   - Vectors `[1 2 3]`
   - Sets `#{"dog" "cat" "wolverine"}`
   - Maps `{:name "Col.  Mustard" :weapon "candlestick" :room "Drawing Room"}`
   - Lists `'(1 2 3)`
   - Evaluated symbols: `x`, `even?`, `upper-bound`

2. Function call `(+ 1 2 3)` `(f "input" :option 1)`

   Whitespace (including ',') and comments are ignored.
   Everything is an expression, which has a value, e.g.: `(if true 100 -100)` has the value 100.
   Functions are first class objects, and some literal objects act like functions too.

Simple Operations

Functions

```
(defn f
  "Documentation here"
  [arguments here]
  (body arguments here))

(f "called" "this way")

(defn f2
  "This function has more than one arity"
  ([] "no arguments value")
  ([one-argument] (+ one-argument 10))
  ([two arguments] [two arguments])
  ([two plus & more] {:first two :second plus :rest more}))

(f2)                         ;=> "no arguments value"
(f2 32)                      ;=> 42
(f2 "these all" "work")      ;=> ["these all", "work"]
(f2 :a :b :c :d :e :f)       ;=> {:first :a, :second :b,
                             ;    :rest [:c :d :e :f]}

;; Anonymous function and a shortcut

(fn [x y] (+ x y))   ; these are equivalent
#(+ %1 %2)

(#(+ %1 %2) 10 6)    ;=> 16

;; Closures

(let [env 10]
  (defn g [x]
    (+ env x)))

(g 20)  ;=> 30
```

Special forms

Binding values

```
(let [x 10,        ;x, y, and z are immutable
      y 20
      z 30]
  (+ x y z 4))     ;=> 64
; x, y, z are not bound out here

(let [simple-mutable (atom 0)] ; an atom is thread safe
  (swap! simple-mutable inc))  ;=> 1

(def answer 42)  ; globally bound vars
(def sums {[1 2 3] 6, [10 22] 32, [] 0})
(def v [1 1 3 5 8])
(def a-map {:a 1 :b 2 :c 3})
(def a-set #{:foo :bar :zap}) ; elements are unique

;; note any values can be keys for a map
```

Conditionals

```
(if true 1 0)   ;=> 1
(if false 1 0) ;=> 0
(if nil 1 0)   ;=> 0
;; only false and nil are falsy, all other values truthy
(if "" 1 0)     ;=> 1
(if 0 1 0)      ;=> 1
(if [] 1 0)     ;=> 1

;; Functions are values too
((if true + -) 4 2)   ;=> 6
((if false + -) 4 2) ;=> 2

(when true
  (println "Hello, world"))
```

Comparisons

```
(= 10 20)   ;=> false
(= [:a :b :c] [:a :b :c])   ;=> true
(< 10 20)   ;=> true
(<= 11 11) ;=> true
(or (= nil nil) (= 3 4))    ;=> true
(and (= nil nil) (= 3 4))  ;=> false
(and (= nil nil) (= 4 4))  ;=> true
(not false)                 ;=> true
```

Loops

```
(loop [step 0]
  (println (str "This is step " step "."))
  (if (> step 9)
    (println "Done.")
    (recur (inc step))))
```

What does this print?

Accessing data

```
(get sums [1 2 3])         ;=> 6
(sums [1 2 3])             ;=> 6  the map acts like a function
(get sums [2 2] :missing) ;=> :missing  (default value)

(get a-map :b)  ;=> 2
(a-map :b)      ;=> 2
(:b a-map)      ;=> 2  keywords also act like a function

(get v 2)      ;=> 3
```

```
(nth v 2)       ;=> 3
(nth v 10)      ;=> EXCEPTION
(nth v 10 42)   ;=> 42
(get v 10)      ;=> nil
```

Adding to collections

```
(conj v 13) ;=> [1 1 3 5 8 13]
v           ;=> [1 1 3 5 8]  (v is immutable)
(conj a-set :ahhh) ;=> #{:foo :bar :zap :ahhh}
a-set              ;=> #{:foo :bar :zap} structure shared

(assoc a-map :d 4) ;=> {:a 1, :b 2, :c 3, :d 4}
a-map              ;=> {:a 1, :b 2, :c 3}  structure shared
```

Destructuring

We can bind names to values within structures with **destructuring**:

```
(let [[x y] [1 2 3]]
  (vector x y))      ;=> [1 2]

(let [[x y & more] [1 2 3 4 5]]
  (vector x y more))  ;=> [1 2 '(3 4 5)]

(let [{:keys [a b c]} {:a 1 :b 2 :c 3}]
  [a b c])    ;=> [1 2 3]

(defn f [[x y] {:keys [a b c]}]
  [x y a b c])

(f [1 2] {:a 10 :b 20 :c 30}) ;=> [1 2 10 20 30]

(defn g [x y & more-args]
  (+ 4 x y (first more-args)))

(g 10 20 30 40 50 60)         ;=> 64
(apply g 10 [20 30 40 50 60]) ;=> 64

;; much more is possible
```

Namespaces and libraries

All code is defined within a **namespace** that controls access to each symbol.

This lets library code be loaded without stepping on other code.

For example: all the code I'm executing now is in the `user` namespace. There are mechanisms for importing symbols from other namespaces, which is how you work with libraries.

There are also ways to access features of the host platform (e.g., the JVM or the javascript environment).

This is beyond our goals for today

Examples

Means and Variances

We can compute simple means using the recurrence

$$\bar{x}_{n+1} = \bar{x}_n + K_n(x_{n+1} - \bar{x}_n),$$

where $K_n = 1/(n+1)$. The value K_n is a "gain" parameter that applies to the "residual" at the next step.

```clojure
(defn simple-update [[xbar n] x]
  (let [n' (+ n 1)
        K (/ n')]
    [(+ xbar (* K (- x xbar))), n']))

(def data (range 5))

(->> data
     (reduce simple-update [0.0 0])
     first)
```

This same gain idea works with weighted averages using the same recurrence with where $K_n = w_{n+1}/\sum_{i=1}^{n+1} w_i$.

```clojure
(defn weighted-update [[xbar wsum] [x w]]
  (let [wsum' (+ wsum w)
        K (/ w wsum')]
    [(+ xbar (* K (- x xbar))), wsum']))

(def data (range 5))
(def weights [1.0 2.0 3.0 4.0 5.0])

(->> (map vector data weights)
     (reduce weighted-update [0.0 0])
     first)
```

Alternatively, we could just keep track of the components

```clojure
(defn weighted-update-nodiv [[x-dot-w wsum] [x w]]
  [(+ x-dot-w (* x w)), (+ wsum w)])

(->> (map vector data weights)
     (reduce weighted-update-nodiv [0.0 0])
     (apply /))
```

but the gain form will come in handy soon.

We can do similar things with variance

```
(defn welford-variance
  "Updating function for Welford variance computation
  0-arity version gives initializer, 2-arity updater."
  ([] {:S 0.0 :xbar 0.0 :wsum 0.0})
  ([{:keys [S xbar wsum]} [x w]]
   (let [wsum' (+ wsum w)
         K (/ w wsum')
         xbar' (+ xbar (* K (- x xbar)))]
     {:S    (+ S (* w (- x xbar) (- x xbar')))
      :xbar xbar'
      :wsum wsum'})))

(let [extract-parts (juxt :S (comp dec :wsum))]
  (->> (map vector data (repeat 1))
       (reduce welford-variance (welford-variance))
       extract-parts
       (apply /)))
```

Least Squares

Consider a basic homoskedastic regression model

$$y = X\beta + \sigma^2\epsilon,$$

where X is $n \times p$, β is $p \times 1$, and ϵ is mean 0, unit variance noise.

We can compute $\hat{\beta} = (X^T X)^{-1} X^T y$ directly with various methods, but it is useful to think of this *sequentially*. With the help of the Woodbury formula for one step updates of an inverse, we have:

```
(defn as-scalar [mat]
  (select mat 0 0))

(defn %*% [a b]
  (let [M (mmul a b)]
    (if (every? #(= % 1) (shape M)) (as-scalar M) M)))

(defn least-squares-update [sigma-squared p scale]
  (fn
    ([] {:beta-hat (new-matrix p 1)
         :V (mul (identity-matrix p) sigma-squared scale)})
    ([{:keys [beta-hat V]} [x y]]
     (let [VxT (mmul V (transpose x))
           D (+ sigma-squared (%*% x VxT))
           K (div VxT D)
           residual (- y (%*% x beta-hat))]
       {:beta-hat (add beta-hat (mul K residual))
        :V (sub V (mul (mmul K (transpose K)) D))}))))
```

```

(def lsdata
  "Sequential regression data, each element of which
  contains a row of X and the corresponding y."
  [[[[1.0 0.0 0.0 0.0]]    -2.28442]
   [[[1.0 1.0 1.0 1.0]]    -4.83168]
   [[[1.0 -1.0 1.0 -1.0]] -10.46010]
   [[[1.0 -2.0 4.0 -8.0]]   1.40488]
   [[[1.0 2.0 4.0 8.0]]   -40.80790]])

(def updater (least-squares-update 1.0 4 1000.0))

(->> lsdata
     (reduce updater (updater))
     :beta-hat)
```

This is regression analysis in a functional style, with a few simple lines

The Kalman Filter

Our sequential least squares actually solves another problem: finding the minimum variance predictor for a linear dynamical system driven by a stochastic process.

Consider state vectors x_t and observation vectors y_t that evolve as follows:

$$\begin{aligned} x_{t+1} &= Ax_t + \delta_t \\ y_{t+1} &= Cx_t + \epsilon_t, \end{aligned}$$

where δ_t and ϵ_t are each, say, iid Normal, mean 0 noise with fixed covariance. (These are called the *state noise* and *measurement noise*, respectively.)

If $\hat{x}_{t+1}$ (and $\hat{y}_{t+1}$) is the best predictor of x_{t+1} (and y_{t+1}) given observations up to time t, then

$$\hat{x}_{t+1} = A\hat{x}_t + K_t(y_t - \hat{y}_t),$$

which is of the same form we just computed.

This gives us a functional implementation of the **Kalman Filter**.

Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) is a simulation method where we create a Markov chain whose *limiting distribution* is a distribution from which we want to sample.

```
(defn mcmc-step
  "Make one step in MH chain, choosing random move."
  [state moves]
  (let [move (random-choice moves)]
    (metropolis-hastings-step (move state))))

(defn mcmc-sample
  "Generate an MCMC sample from an initial state.
   Returns a lazy sequence.

   Keyword arguments:

     :move    -- a collection of moves, which are
                 functions from state to a candidate
     :select  -- a selector function which indicates
                 a boolean for each index and state
                 if that state should be kept in the
                 output
     :extract -- a function to extract features
                 (e.g., parameters) from the state
  "
  [initial-state & {:keys [moves select extract]
                    :or {select (constantly true),
                         extract identity}}]
  (letfn [(stepper [index state]
            (lazy-seq
             (if (select index state)
               (cons (extract state)
                     (stepper (inc index) (mcmc-step state moves)))
               (stepper (inc index) (mcmc-step state moves)))))]
    (stepper 0 initial-state)))

(def chain
  (mcmc-sample initial-state
               :select (mcmc-select :burn-in 1000 :skip 5)
               :extract :theta1
               :moves [(univariate-move :theta1 random-walk 0.1)
                       (univariate-move :theta2 random-walk 0.4)
                       (univariate-move :theta3 random-walk 0.2)]))

(take 100 chain)
```

The design of this chain is modular, and easily adaptable to a wide variety of models.

Resources

- Hadley Wickham's *Advanced R*, see FP and Functionals.
- R bloggers on FP in R
- The `purr` and `magrittr` packages are good building blocks for functional-style R.
- The `dplyr` package embodies many functional concepts.
- In python, the `functools` module provides additional higher-order functions; the `itertools` module provides iterator methods, which supports a functional style.
- Clojure For the Brave and True
- Clojure main site

- Try Clojure online
- Haskell Book
- Gentle Intro to FP (in Scala but still)
- Steve Yegge's Execution in the Kingdom of Nouns, an OOP rant
- List out of lambda, how to produce high-level constructs from closures, etc.
- Talk by Rich Hickey Are we there yet? (esp. first 30 minutes)