CS 696 Functional Programming and Design
Fall Semester, 2015
Doc 13 Assignment 2 Comments
Oct 20, 2015

```
(defn contains-sub-1 [text sub]
  (if (= sub (re-find (re-pattern sub) text))
    true
    false))
```

```
(defn contains-sub-1 [text sub]
  (= sub (re-find (re-pattern sub) text)))
```

```
(defn contains-pattern [text pattern]
  (= pattern (re-find (re-pattern pattern) text)))
```

```
(defn contains-pattern [text pattern]
  (re-find (re-pattern pattern) text))
```

```
(defn divisors [dividend]
 (filter (fn [divisor-element]
      (if (= 0 (mod dividend divisor-element))
       true
       false
       )
      )
     (range 1 (inc dividend))
     )
 )
```

```
(defn factor?
 [n factor]
 (= 0 (mod n factor)))

(defn divisors [dividend]
 (let [possible-factors (range 1 (inc dividend))]
  (filter factor? possible-factors)))
```

```
(defn divisors [dividend]
 (filter (fn [divisor-element]
      (= 0 (mod dividend divisor-element)))
     (range 1 (inc dividend))
     )
 )
```

```
;; 4. Write a function, pattern-count with two arguments. The first arguments is a string, lets c
;; it text, and the second argument is also a string, call it pattern. The function pattern-count
;; return the number of times the pattern occurs in the text. For example
;; (pattern-count “abababa” “aba”) returns 3
;; (pattern-count “aaaaa” “aa”) returns 4
;; (pattern-count “Abcde” “abc”) returns 0

(defn not-main-function
  [text pattern numberoftimes]
  (if (<= (count pattern) (count text))
    (let [sub-text (subs text 0 (count pattern))]
      (if (= sub-text pattern)
        (not-main-function (subs text 1) pattern (inc numberoftimes))
        (not-main-function (subs text 1) pattern numberoftimes)))
    numberoftimes))

(defn main_function
  [text pattern]
  (not-main-function text pattern 0))
```

not-main-function
numberoftimes
main_function
pattern-count??

Names

```
(defn FileData


(defn splitString


(defn String->Number [str]


(defn calcSpread
```

Names

Future assignments

If name does not follow Clojure structure lose one point

```
(defn divisors [number]
  "Function to calculate the divisors of a number"
  (for [i (range 1 (+ number 1))
        :when (= (rem number i) 0 )] i
                                      )
  )
```

```
;Function returns string only if the occurence is greater than n
(defn myStringF
  [p n]
  (if (>= (second p) n)
    [(apply str (first p)) (last p)])

  )

;Function returns only word from the word occurence pair
(defn return-string
  [p]
  (if nil? p)

  (first p))
```

```
(defn  find-abundance [x]

  (for [y (range 1 x) :when (> (abundance y) 0)] y))
```

```
(defn find-pattern [count1 text pattern]

 (if (>= (count text)(count pattern))
  (do
   (if (= (subs text 0 (count pattern)) pattern)
    (find-pattern (inc count1) (subs text 1) pattern) (find-pattern count1 (subs text 1) pattern))

   )count1))
```

```
(defn process-lineitem [line]

  (def items (.split line "\t"))

  (if  (not= (clojure.string/blank? line) true)
    (if (> (count items) 2)
      (if (not= (nth items 0) "Dy")
        (into {} { :day (convert-str-int (nth items 0)) :spread (-(convert-str-int (nth items 1)) (convert-str-int (nth items 2)))})
        )
      )
    )
  )

(defn convert-str-int [input]
  (Integer/parseInt (clojure.string/replace input #"\*" "") )
  )
```

Formatting

Bad formatting will lose points

```
(defn  find-abundance [x]

  (for [y (range 1 x) :when (> (abundance y) 0)] y))
```

```
(defn pattern-count
 [text pattern]
 (let [pattern-length (count pattern)
    pattern-sequence (seq pattern)]
  (loop [pattern-counter 0
      rem-text text]
   (if (< (count rem-text) pattern-length)
    pattern-counter
    (let [text-match? (= (take pattern-length rem-text) pattern-sequence)]
     (recur
      (if text-match?
       (inc pattern-counter)
       pattern-counter)
      (rest rem-text)))))))
```

```
(defn most-frequent-word [a n]
  (map :key
       (last
         (last
           (group-by :count
                     (into[]
                          (distinct
                            (mapv #(hash-map (keyword "key") (subs a % (+ n %))
                                              (keyword "count") (pattern-count a (subs a % (+ n %))))
                                  (range 0 (+(-(.length a)n)1))))))))))
```

Atom

```
(defn pattern-count [text pattern]
 (let [len-text (count text), len-pattern (count pattern), matches (atom 0)]
  (loop [index 0]
   (if (<= (+ index len-pattern) len-text)
    (do
     (let [sub-string (subs text index (+ index len-pattern))]
      (if (= sub-string pattern)
       (swap! matches inc)
       )
      )
     (recur (inc index))
     )
    )
   )
  (deref matches)
  )
 )
```

```
(let [numbers (vec (rest (range 300)))]
  (filterv integer? (map (fn [n] (if (> (abundance n) 0) n)) numbers))
  )
```

```
(defn abundant-numbers
  [n]
  (let [numbers (vec (rest (range n)))]
    (filterv integer? (map (fn [n] (if (> (abundance n) 0) n)) numbers))))
```

```
(defn sub-blocks
  "Returns a collection of sequential sub-string blocks in s of size n"
  [s n]
  (map #(subs s % (+ % n)) (range (- (count s) (dec n)))))

(defn equal?
  "Returns true if s1 and s2 are equal strings"
  [s1 s2]
  (= (compare s1 s2) 0))
```

```
(defn equal? [s1 s2] (= s1 s2))
```

```
(defn pattern-count
  "Returns a count of the occurrences of ptrn in s"
  [s ptrn]
  (count (filter #(equal? % ptrn) (sub-blocks s (count ptrn)))))
```

```
 Returns the parsed day data in s as a hash-map
 [s]
 (if (contains-day-data? s)
   (let [day-data (word-blocks s)]
     (zipmap [:day-number :max-temp :min-temp] (mapv read-string (take 3 day-data))))))

(defn day-temp-spread
 "Returns the temperature spread for day"
 [day]
 (- (day :max-temp) (day :min-temp)))

(defn max-temp-spread-day
 "Returns the day with the max temperature spread"
 [day1 day2]
 (if (> (day-temp-spread day1) (day-temp-spread day2))
   day1
   day2))

(defn maximum-spread
 "Returns the day number of the day that has the largest temperature spread.
 Input is path to data file."
 [path]
 (let [lines (clojure.string/split-lines (slurp path))]
   ((reduce max-temp-spread-day (remove nil? (map #(parse-day-data %) lines))) :day-
(defn fetch-data[path]
  (rest (rest (map  #(clojure.string/split % #"\t") (clojure.string/split-lines  (slurp path))))))
```

```
(ns gradeasssignment2.core
  (:require [clojure.string :as string]))

(defn fetch-data [path]
     (->> path
          slurp
          string/split-lines
          (map #(string/split % #”\t”)
         rest
         rest))
```

```
(ns gradeasssignment2.core
  (:require [clojure.string :as string]))

(defn fetch-data [path]
     (->> (slurp path)
          string/split-lines
          (map #(string/split % #”\t”)
          rest
          rest))
```

```
;get-temp-range takes one argument that is path of dat file
; it skips the first two rows and calculates temp range for all the days and return in vector
(defn get-temp-range[path]
  (for [[x y z & rest] (rest (rest(map #(str/split % #"\t") (str/split-lines (slurp path)))))]
    (vector (str x) (-(Integer/parseInt (re-find (re-pattern "\\d+") y)) (Integer/parseInt (re-find (re-pattern "\\d+") z))))))

;maximum-spread takes path of dat file and finds days with maximum temperature range.
(defn maximum-spread[path]
  (for [[x y]  (second (last (sort-by first (group-by second (get-temp-range path)))))]
    x))
```

```
(defn get-temp-range[path]
  (for [[x y z & rest] (rest (rest(map #(str/split % #"\t") (str/split-lines (slurp path)))))]
    (vector (str x) (-(Integer/parseInt (re-find (re-pattern "\\d+") y)) (Integer/parseInt (re-find (re-pattern "\\d+") z))))))
```

```
(defn fetch-data [path]
     (->> (slurp path)
           string/split-lines
           (map #(string/split % #”\t”)
          rest
          rest))

(defn get-temp-range[path]
  (for [[x y z & rest] (fetch-data path)]
    (vector (str x) (-(Integer/parseInt (re-find (re-pattern "\\d+") y)) (Integer/parseInt (re-find (re-pattern "\\d+") z))))))
```

```
(defn fetch-data [path]
  (->> (slurp path)
       string/split-lines
       (map #(string/split % #""\t""))
        rest
        rest))

  (defn get-temp-range [path]
    (for [[x y z & rest] (fetch-data path)]
      (vector
        (str x)
        (-
          (Integer/parseInt (re-find (re-pattern "\\d+") y))
          (Integer/parseInt (re-find (re-pattern "\\d+") z))))))
```

rest not used

Repeating same code

```
(defn fetch-data [path]
  (->> (slurp path)
       string/split-lines
       (map #(string/split % #""\t""")
        rest
        rest))

  (defn get-temp-range [path]
    (for [[x y z] (fetch-data path)]
      (vector
        (str x)
        (-
           (Integer/parseInt (re-find (re-pattern "\\d+") y))
           (Integer/parseInt (re-find (re-pattern "\\d+") z))))))
(defn fetch-data [path]
  (->> (slurp path)
       string/split-lines
       (map #(string/split % #""\t"")
        rest
        rest))


(defn get-temp-range [path]
  (for [[x y z] (fetch-data path)]
    (vector
      (str x)
      (- (string->int y)) (string->int z))))))
```

```
(vector a b) <-> [a b]
```

x is a string

```
(defn string->int
  [s]
  (Integer/parseInt (re-find (re-pattern "\\d+") s)))
```

```
(defn find-int
  [s]
  (re-find (re-pattern "\\d+") s))

(defn string->int
  [s]
  (-> (find-int s)
      Integer/parsInt))
```

```
(defn fetch-data [path]
  (->> (slurp path)
       string/split-lines
       (map #(string/split % #""\t""))
        rest
        rest))
```

```
(defn string->int
  [s]
  (Integer/parseInt (re-find (re-pattern "\\d+") s)))
```

```
(defn get-temp-range [path]
  (for [[x y z] (fetch-data path)]
    [x  (- (string->int y)) (string->int z))] ))
```

Did using [ ] help?

Should get-temp-range argument be path or contents of the file

get-temp-range   -> temperature-range

```
(defn divisor [x]
  (distinct (reduce #(if (zero? (mod x %2)) (conj %1 (/ x %2) %2) %1) () (range 1  (-> x (Math/
sqrt) (Math/round) (inc)))))
  )


 (defn divisors [x]
   (sort
     (distinct
       (reduce
         #(if (zero? (mod x %2)) (conj %1 (/ x %2) %2) %1)
         ()
         (range 1  (-> x (Math/sqrt) (Math/round) (inc)))
         )
       )
     )
   )
```

```
(defn most-frequent-word1
  [mainString n]
  (into [] (filter #(get-val % (frequencies (partition n 1 mainString))) (frequencies (partition n 1
mainString)))))
```

```
(defn most-frequent-word1
  [mainString n]
  (into []
        (filter
          #(get-val % (frequencies (partition n 1 mainString)))
          (frequencies (partition n 1 mainString)))))
```

```
(defn divisors
  [n]
  (filter
    (comp zero? (partial mod n))     ;a number is n's divisor iff n mod it gets 0
    (range 1 (inc n))))
```

```
((defn divisors-helper
   [x y]
   (if (= 0 (mod x y))
     y
     0)) 9 1)
```

```
(defn divisors-helper
  [x y]
  (if (= 0 (mod x y))
    y
    0))

(divisors-helper 9 1)
```

```
(defn find-clumps
    [string k L t]
    (let [possible-clumps (partition L 1 string)]
        (map #(apply str (first %)) (filter (fn
                              [[in _]]
                              (>
                               (count (filter #(>= (pattern-count % in) t) possible-clumps))
                               0))
                             (filter #(>= (second %) t) (freq-map string k))))

        ))
```

```
(defn max-frequent-sub [st k1]
  ( apply max(vals (all-sub st k1))))
```

```
;; why have argument n if you don't use it?
(defn abundant-numbers[n]
  (remove nil? (map abundant-helper (range 1 300))))
```

```
(defn all-sub [st k]
  (frequencies(map (fn[n] (clojure.string/join "" n))(partition k 1 st))))
```

```
(defn all-sub [st k]
  (map (fn[n] (clojure.string/join "" n))(partition k 1 st)))
```

```
(defn all-sub [st k]
  (frequencies(map (fn[n] (clojure.string/join "" n))(partition k 1 st))))
```

```
(defn max-frequent-sub [st k1]
  (apply max(vals (all-sub-a st k1))))
```

```
(defn most-frequent-word [string n]
  (let [map-result (all-sub-a string n)]
    (for [[k v] map-result
          :when (= v (max-frequent-sub string n))] k)))
```

```
(defn all-sub [st k]
  (map (fn[n] (clojure.string/join "" n))(partition k 1 st)))
```

redefined

rem-astrix not used

```
(defn maximum-spread
  [file-path]
  (let[parser(parse-line-to-vec file-path)]
    (let[rem-astrix (mapv replace-helper (into[](map get-three (nthnext parser 2))))]
      (let[range-vec (map range-helper (mapv replace-helper (into[](map get-three (nthnext
parser 2)))))]
        (let [temp-max (apply max(map second range-vec))]
          (first(nth range-vec (.indexOf (map second range-vec) temp-max))))))))
```

```
(defn max-spread-index
  "max-spread-index: find out the index of the map that
   has the largest spread.
   @param: path-string, destination directory."
  [path-string]
  (let [spread-kv (vec (map #(second %) (data-map directory)))]
    (let [spread-v (vec (map #(second %) spread-kv))]
      (.indexOf spread-v (apply max spread-v)))))
```

```
(defn max-spread-index
  [path-string]
  (let [spread-kv (vec (map #(second %) (data-map directory)))
        spread-v (vec (map #(second %) spread-kv))]
    (.indexOf spread-v (apply max spread-v))))
```

can define multiple values in one let

path-string not used

```
(defn abundance-under-300 []
  (filter (fn [n]
             (pos? (abundance n))
             )
           (range 1 (inc 300))
           )
  )


(defn abundant-range
  [n]
  #_(find abundant numbers less than n)
  (filter #(> (abundant %) 0) (range n)))
```

```
(defn abundant-range
  [n]
  #_(find abundant numbers less than n)
  (filter #(> (abundant %) 0) (range n)))
```

```
(defn abundant-range
  [n]
  #_(find abundant numbers less than n)
  (filter (comp pos? abundant) (range n)))
```

```
(maximum-spread "http://www.eli.sdsu.edu/courses/fall15/cs696/assignments/weather.dat")
```

```
(defn maximum-spread [path]
 (for [[x y]  (second (last (sort-by first( group-by second
                               (for [[x y z & rest] (rest (rest (with-open [rd (io/reader (io/file
path))]
                                                   (->> (line-seq rd) (map #(.split ^String
% "\t")) (mapv vec)))))]
                                (vector (str x) (-(Integer/parseInt (re-find (re-pattern "\\d+")
y)) (Integer/parseInt (re-find (re-pattern "\\d+") z))))))))))]
   x))
```

```
(defn maximum-spread [path]
 (require '[clojure.string :as str])
 (loop [n 0
    final []]
  (if (< n (count (clojure.string/split-lines (clojure.string/replace (slurp path)#"\t" " "))))
   ; The below statement is used to add each element of "data" into a blank vector, and then add it to a vector "final".
   (recur
    (inc n)
    (conj final (conj [] (nth (clojure.string/split-lines (clojure.string/replace (slurp path)#"\t" " ")) n))))
   (test2 (test1 final))
   )
  )
 )
```

```
(defn patt2 [list1 n]
  (partition n (for [x (range (- (count list1) (- n 1))) y (range n)]
               (nth list1 (+ x y))))
  )
```

Question 3

```
(defn patt1 [list1 n]
  (for [x (for [y (patt2 list1 n)]
          (into [] y))]
    (apply str x))
  )
```

Question 5

```
(defn patt [strng k l t]
  (for [x (partition l 1 strng)]
    (apply str (into [] x)))
  )

;; This generates chucks of k length for all l length string above.

(defn patt2 [strng k l t]
  (let [y (map #(partition k 1 %) (patt strng k l t))
        num (count (nth y 0))]
    (partition num (for [x y cnt (range (count x))]
                 (apply str (into [] (nth x cnt))))
               )
    )
  )
```

y not used

```
(defn vec-frequent-word [x y]
 (loop [incr 0 vect[]]
  (if (<= incr (- (count x)  y))
   (recur (inc incr)
        (conj vect (vec [(subs x incr (+ incr y)) (pattern-count x (subs x incr (+ incr y)))])))
   (for [[x y] (second (last (sort-by first (group-by second (distinct vect)))))]
    x))))
```

Which y not used?

How many X & Y’s?

;frequent-word takes three argument, first master string x, size of substring y and minimum frequency of the substring requires.

;This function loops over master string and check each possible substring of size Y, and its occurence in master string

;All the results are stored in vector which is checked to find strings with minimum frequency.

```
(defn frequent-word [x y z]
  (loop [incr 0
         vect[]]
    (if (<= incr (- (count x)  y))
      (recur (inc incr)
             (conj vect (vec [(subs x incr (+ incr y)) (pattern-count x (subs x incr (+ incr y)))])))
      (for [[x y] (filter #(>= (first %) z )  (group-by second (distinct vect)))]
        (for [[x y] y]
          x))
      )))
```