```
; vim: ts=2 sw=2 et :
(defstruct  about
  (what       "wicked.lisp do cool words")
  (copyright "(c) 2022 Tim Menzies")
  (file       "where to get data          ")
  (cohen      "small effect size          ")
  (far        "how far to seek distant items ")
  (help       "show help                  ")
  (some       "max samples for distance calcs")
  (lnorm      "norm; –L 2 means 'euclidean'  ")
  (seed       "random number seed         ")
  (cli        '(file cohen far help some lnorm seed)))

(defvar *about* (make-about
  :cohen .35 :far .9 :help  nil :some 256 :lnorm 2 :seed 10019
  :file "../data/auto93.csv" ))

(defmethod print-object ((x about) s)
  (let ((ab (make-about)))
    (format s "~&~a~%~a~%~%OPTIONS:~%" (about-what ab) (about-copyright ab))
    (dolist (y (about-cli ab))
      (format s " --~(~8a~) ~4a ~a~%" y (slot-value ab y) (slot-value x y)))))

;;;;-------------------------------------------------------------------
(defmacro ! (x) `(slot-value *about* ',x))

(defmacro ? (s x &rest xs)
  (if xs `(? (slot-value ,s ',x) ,@xs) `(slot-value ,s ',x)))

(defun sum (lst &optional (f #'identity))
 (reduce '+ (mapcar f lst)))

(defmacro has (key dictionary)
  `(cdr (or (assoc ,key ,dictionary :test 'equal)
            (car (setf ,dictionary (cons (cons ,key 0) ,dictionary))))))

(defun words (s &optional (sep #\,) (x 0) (y (position sep s :start (1+ x))))
  (cons (subseq s x y) (and y (words s sep (1+ y)))))

(defun thing (s)
  (let ((s (string-trim '(#\Space #\Tab) s)))
    (if (equal s "?") #\?
      (let ((x (ignore-errors (read-from-string s)))) (if (numberp x) x s)))))

(defmacro with-csv ((cells file &optional out) &body body)
  (let ((s (gensym)))
    `(let (,cells)
       (with-open-file (,s ,file)
         (loop while (setf ,cells (mapcar 'thing (words (read-line ,s nil))))
           do ,@body) ,out))))

(defmacro do-cells ((at cell rows &optional out) &body body)
  (let ((row (gensym)))
    `(let (,cell) (dolist (,row ,rows ,out)
                    (setf ,cell (elt (slot-value ,row 'cells) ,at))
                    (when (not (equal ,cell #\?)) ,@body)))))

(defun cli ()
  (labels ((args ()     #+clisp ext:*args* #+sbcl sb-ext:*posix-argv*)
           (has  (x)    (member x (args) :test 'equalp))
           (new  (x b4) (if (has x) (cond ((eq b4 t)   nil)
                                          ((eq b4 nil) t)
                                          (t (thing (elt (has x) 1))))
                            b4)))
    (dolist (slot (? cli) *about*)
      (setf (slot-value *about* slot)
            (new (format nil "--~(~a~)" slot) (slot-value *about* slot))))))

;;;;-------------------------------------------------------------------
(defstruct (row (:constructor %row-make)) cells klass)

(defmethod make-row ((lst cons)) (setf (? row cells) lst))
(defmethod make-row ((row1 row)) (make-row (? row cells)))

(defmethod at ((self row)) (elt (at self cells) at))

;;;;-------------------------------------------------------------------
(defstruct (num (:constructor %make-num))
  (n 0) (at 0) (w 0) (txt "") (mid 0) (div 0) (m2 0) (lo 1E32) (hi -1E32))

(defun make-num (at txt rows &aux (it (%make-num :at at :txt txt)))
  (with-slots (w n most mid div m2) it
    (setf w (if (lessp txt) -1 1))
    (do-cells (at x rows it)
      (let ((d (- x mid)))
        (incf n)
        (incf mid (/ d n))
        (incf m2  (* d (- x mid)))
        (setf div  (if (< n 2) 0 (sqrt (/ m2 ( - n 1)))))))))

(defmethod dist ((self num) x y)
  (cond ((and (eq x #\?) (eq y #\?)) (return-from dist 1))
        ((eq x #\?) (setf y (norm self y) x (if (< y .5) 1 0)))
        ((eq y #\?) (setf x (norm self x) y (if (< x .5) 1 0)))
        (t          (setf x (norm self x) y (norm self y))))
  (abs (- x y)))

;;;;-------------------------------------------------------------------
(defstruct (sym (:constructor %make-sym))
  (n 0) (at 0) (txt "") all (most 0) mid (div 0))

(defun make-sym (at txt rows &aux (it (%make-sym :at at :txt txt)))
  (with-slots (n all most mid div) it
    (do-cells (at x rows it)
      (incf n)
      (let ((tmp (incf (has x all))))
        (if (> tmp most)
          (setf most tmp
                mid  x))))
    (dolist (two all)
      (print two)
      (let ((p (/ (cdr two) n))) (decf div (* p (log p 2)))))))

(defmethod dist ((self sym) x y)
  (if (and (eq x #\?) (eq y #\?)) 0 (if (equal x y) 0 1)))


;;;;-------------------------------------------------------------------
(labels ((pre=   (s x) (eql x (char s 0)))
         (end=   (s x) (eql x (char s (1- (length s))))))
  (defun ignorep (s)   (end= s #\X))
  (defun klassp  (s)   (end= s #\!))
  (defun lessp   (s)   (end= s #\-))
  (defun morep   (s)   (end= s #\+))
  (defun goalp   (s)   (or (end= s #\+) (end= s #\-) (end= s #\!)))
  (defun nump    (s)   (if (pre= s #\$) 'make-num 'make-sym)))

(defstruct (egs (:constructor %make-egs)) rows cols x y names)

(defun make-egs (rows)
  (let* ((it (%make-egs :names (pop rows) :rows  (mapcar #'make-row rows))))
    (loop for txt in names for at from 0 do
      (let ((col (funcall (if (nump) 'make-num 'make-sym) at txt rows)))
        (push col (? it cols))
        (if (not (ignorep txt))
          (if (goalp txt) (push col (? it y) (? it x))))))
    it))

(defmethod dist ((self egs) row1 row2)
  (labels ((d1   (col) (dist col (at row1 (? col at)) (at row2 (? col at))))
           (col1 (col) (expt (d1 col) (? lnorm))))
    (expt (/ (sum 'col1 (? self x)) (length (? self x)))
          (/ 1 (? lnorm)))))
;;;;-------------------------------------------------------------------

(cli)
(print *about*)


```

```
;(print (? help))
;

; (defstruct (cols (:constructor %make-cols)) all x y txts klass)
;
; (defun make-cols (txts &aux (at -1) x y klass all)
;   (dolist (s txts (%make-cols :txts txts :all (reverse all)
;                               :x (reverse x) :y (reverse y) :klass klass))
;     (let ((now (funcall (if (is s 'num) 'make-num 'make-sym) (incf at) s)))
;       (push now all)
;       (when (not (is s 'ignore))
;         (if (is s 'goal)  (push  now y) (push now x))
;         (if (is s 'klass) (setf klass now))))))
;
;
; (defmacro with-csv ((lst file &optional out) &body body)
;   (let ((str (gensym)))
;     `(let (,lst)
;        (with-open-file (,str ,file)
;          (loop while (setf ,lst (read-line ,str nil)) do ,@body)
;          ,out))))
;
; ;;;;
; (defun args ()  #+clisp ext:*args* #+sbcl sb-ext:*posix-argv*)
; (defun stop (n) #+sbcl (sb-ext:exit :code n) #+:clisp (ext:exit n))
;
; (defun any (seq)    (elt seq (randi (length seq))))
; (defun many (seq n) (let (a) (dotimes (i n a) (push (any seq) a))))
;
; (defun per (seq &optional (p .5) &aux (v (thing seq 'vector)))
;   (elt v (floor (* p (length v)))))
;
; (defun trim (s) (string-trim '(#\Space #\Tab) s))
;
; (defun thing (s &aux (s1 (trim s)))
;   (if (equal s1 "?") #\? (let ((x (ignore-errors (read-from-string s1))))
;                            (if (numberp x) x s1))))
;
; (defun words (s &optional (sep #\,) (x 0) (y (position sep s :start (1+ x))))
;   (cons (subseq s x y) (and y (words s sep (1+ y)))))
;
; (defun sd (seq &optional (key 'identity))
;   (if (<= (length seq) 5) 0
;     (/ (- (funcall key (per seq .9)) (funcall key (per seq .1))) 2.56)))
;
; (defun ent (alist &aux (n 0) (e 0))
;   (dolist (two alist) (incf n (cdr two)))
;   (dolist (two alist e) (let ((p (/ (cdr two) n))) (decf e (* p (log p 2))))))
;
; (defun round2 (number &optional (digits 2))
;   (let* ((div (expt 10 digits))
;          (tmp (/ (round (* number div)) div)))
;     (if (zerop digits) (floor tmp) (float tmp))))
;
; (defun round2s (seq &optional (digits 2))
;   (map 'list (lambda (x) (round2 x digits)) seq))
;
; (labels ((park-miller () (setf *seed* (mod (* 16807.0d0 *seed*) 2147483647.0d0
0))
;                          (/ *seed* 2147483647.0d0)))
;   (defun randf (&optional (n 1)) (*   n (- 1.0d0 (park-miller)))) ;XX check th
is
;   (defun randi (&optional (n 1)) (floor (* n (park-miller)))))
;
; ;;;-------------------------------------------------------------------------
--
; ;;;-------------------------------------------------------------------------
--
; ;;;-------------------------------------------------------------------------
--
;
```