```
;; In Scheme, parentheses are NOT arbitrary punctuation.
;; You must use them in exactly the right number and placement

1          ;; The number 1; similarly 2.5, -68, etc.
x          ;; A name; if not bound to a value, this gives an error
#t         ;; The true boolean; similarly #f
"A string" ;; As expected
'x         ;; A symbol
+ - * / < > <= >= = and or not eq? zero? null? number? ;; built-in functions

(fun arg1 ... argn)   ;; Use parentheses to call fun on arg1 through argn;
                      ;;   no punctuation between args
;; e.g.
(+ 3 (* 25 6))

(define (proc arg1 ... argn)  ;; Define a function; no punctuation
  body)                       ;;   between args; returns value of body

(if test    ;; Conditional; returns either val of then or of else
    then    ;; Note that there is also a more flexible conditional
    else)   ;;   called COND with somewhat more arcane syntax
;; e.g.
(define (count-down n)    ;; Most interesting functions use recursion
  (if (= n 1)
      (list 1)
      (cons n (count-down (- n 1)))))  ;; Speaking of cons...see below

(cons x y)          ;; Returns a pair -- prints as (x . y ) -- python 2-tuple
(first (cons x y)) ;; Returns the value of x -- sometimes spelled car
(rest (cons x y))  ;; Returns the value of y -- sometimes spelled cdr
nil                ;; The empty list; prints as ()

(cons x nil)        ;; Returns list of one element (the value of x)

(cons x             ;; Returns list of two elements --
  (cons y nil))     ;;   first is the value of x; rest is a list of
                    ;;   one element (the value of y)
(list x y)          ;; Equivalent to (cons x (cons y nil)), w/arbitrary # args

;; Useful list procedures:
append      ;; Non-destructive
reverse
map         ;; e.g. (map double (list 1 2 3)) --> (2 4 6)
filter      ;; e.g. (filter odd? (list 1 2 3)) --> (1 3)
null?       ;; true iff the list is empty, i.e., nil

(lambda (arg1 ... argn)   ;; an anonymous procedure
  body)
```

```
;; useful debugging functions -- often used within a begin (see below)
(display "string")
(display x)
(newline)


(begin      ;; scheme's version of blocks
  expr1     ;; sequence:  evaluate the first expression, throw out its value,
  expr2     ;; evaluate the next...
    .
    .
    .       ;; eventually returning the final value
)
```

Racket has built-in documentation

Python for Lisp Programmers can also be read in reverse.

http://www.cs.cornell.edu/courses/cs212/1999sp/handouts/scheme-quickref.htm is a very quick overview of R5RS BUT avoid anything that has a ! in its name