Generating (pseudo)-random numbers

Scheme has a built-in function for generating random numbers:

(random n) returns a “random” element from the set \{0, 1, 2, \ldots, n − 1\}.

By “random,” we mean that it is not possible to predict the value that is returned, but after a large number of evaluations, each element should be returned approximately the same number of times. Thus, we might find

(random 10) \Rightarrow 3
(random 10) \Rightarrow 7
(random 10) \Rightarrow 4
(random 10) \Rightarrow 5
(random 10) \Rightarrow 7
(random 10) \Rightarrow 0
(random 10) \Rightarrow 2
(random 10) \Rightarrow 3
In the expression, (random \( n \)), \( n \) must be an integer. Thus, 

\[
\text{(random 10.0)} \Rightarrow \text{random: expects argument of type <exact integer in \([1, 2147483647]\)>; given 10.0}
\]

If we want to generate real-valued random numbers, we will need to extend scheme ourselves:

\[
\begin{align*}
\text{(define } & *\text{max-integer}* \text{ (- (expt 2 31) 1))} \\
\text{(define (uniform-random } & r) \\
\quad (* \text{(exact->inexact (/ (random *\text{max-integer}*)) *\text{max-integer}*))) \\
\quad r))
\end{align*}
\]

\[
\text{(uniform-random 10.0)} \Rightarrow 6.928134652286831
\]
Simulating games with dice

Using (random 6) we can simulate the roll of a single six-sided die. Note that the value of the above is in the set \{0, 1, 2, 3, 4, 5\}. Thus, we might prefer to define

\[
\text{(define (die-throw)}
  \text{ (+ (random 6) 1))}
\]

To roll two dice, try

\[
\text{(define (two-dice-throw)}
  \text{ (+ (die-throw) (die-throw))})
\]

Note that both of the above represent functions without any arguments.
Iteration in scheme using do

Scheme has a special form called do that performs iteration. The general structure of a do statement is

\[
\text{do (init}_1 \text{ init}_2 \ldots) \\
\text{(test value)} \\
expression_1 \\
\ldots \\
expression_n
\]

Here \( \text{init}_1 \) specified the initial and subsequent values of the first iteration index. The initialization expression \((\text{i 0} \ (+ \text{ i } 1))\) sets the iteration index \(\text{i}\) to zero, and increments it by 1 after each iteration. During each iteration, \(expression_1\) through \(expression_n\) are evaluated in sequence. \(test\) is a boolean expression, like \((= \text{ i } 100)\) that will terminate the loop when it evaluates to \#t. When this occurs, do evaluates the final expression \(value\) and returns the result.
Thus what do the following expressions do?

```
(do ((i 1 (+ i 1)) (val 1))
    ((> i 52) val)
    (set! val (* i val)))
```
Another example related to dice

Thus what do the following expressions do?

(define vector-inc!
    (lambda (vec index)
        (vector-set! vec index (+ (vector-ref vec index) 1))))

(define make-table
    (lambda (trials)
        (do ((vec (make-vector 13))
             (t 0 (+ t 1)))
            ((= t trials) vec)
            (vector-inc! vec (two-dice-roll)))))

> (make-table 100000)
13(0 0 2830 5553 8364 11248 13895 16562 13759 11164 8235 5645 2745)
How are random numbers generated?

The simplest method is called the *linear congruential method*. Following Donald Knuth’s *The Art of Computer Programming*, vol. 2, Addison-Wesley, (1998), let

\[
m = \text{the modulus}, \quad 0 < m;
\]

\[
a = \text{the multiplier}, \quad 0 \leq a < m;
\]

\[
c = \text{the increment}, \quad 0 \leq c < m;
\]

\[
X_0 = \text{the starting value}, \quad 0 \leq X_0 < m.
\]

Then the sequence of random values is obtained by

\[
X_{n+1} = (aX_n + c) \mod m, \quad n \geq 0,
\]

where \(x \mod m\) denotes the remainder obtained after dividing \(x\) by \(m\).
Linear congruential method

;;; Linear congruential method for generating pseudo-random numbers
(define *x* 5772156648)
(define (my-rand r)
  (let ((m (expt 2 32))
         (a 3141592621)
         (c 2718281829))
    (set! *x* (modulo (+ (* a *x*) c) m))
    (modulo *x* r))))

> (my-rand 10)
7
> (my-rand 10)
2
> (my-rand 10)
5
> (my-rand 10)
6