; The Function 'apply'
; ======================
; When you want to use a function, but have a list of the arguments instead of just the arguments.

; The following is an error:
; (+ (list 1 0 4))

; But this isn't:
(apply + (list 1 0 4))

; It means:
(+ 1 0 4)

(check-expect (apply + (list 1 0 4))
  (+ 1 0 4))

(check-expect (apply + (list 1 0 4))
  5)

; That just shows 'apply' used to do (+ 1 0 4) in a more complicated way.
; It's silly to put some explicit values into a list just to immediately use 'apply'.

; In the first 'apply' example from the previous lecture, 'apply' was used with 'beside' and a list
; that was generated via 'range' and 'map': writing out the 'beside' expressions manually would be
; more effort.

; Intermediate Step
; -------------
; The general intermediate step (which unfortunately isn't shown in the Stepper), is this:
; (apply f (list a b c ...))
; (f a b c ...)

; Question.
; Suppose the following definition is run:
(define a-list (list "Hello" " my " "friend!"))
; Show the steps to produce the result of:
(apply string-append a-list)
; Answer.
(apply string-append (list "Hello" " my " "friend!")) ; Step.
(string-append "Hello" " my " "friend!")); Step.
"Hello my friend!" ; Result value.

; Contrasted with 'map'
; ---------------------

; Recall that the addition function takes any number of numbers and adds them up.
; + : number ... -> number

; It can take a single number, and then just produces that number.
(check-expect (+ 123) 123)

; That used addition as a unary function.
; + : number -> number
; So we can use '+' to contrast the behaviour of 'map' and 'apply':
(check-expect (map + (list 1 0 4))
  (list (+ 1) (+ 0) (+ 4)))
(check-expect (apply + (list 1 0 4))
  (+ 1 0 4))
Main Use of 'apply'

The function 'apply' is most useful with functions that take any amount of arguments
(or at least lots of arguments).
Its main purpose is to work on lists that are computed, rather than known ahead of time.

Some examples of functions most useful to use with 'apply':
- * * < <= > min max
- beside above overlay
- string-append

< : number number ... -> number
(< a b c ...) asks whether the numbers a b c ... are in increasing order.

(check-expect (< 104 123 321 1000) #true)
(check-expect (< 104 123 121 1000) #false)

The range function with a positive step produces a list of numbers in increasing order.
(check-expect (apply < (range 1 104 3)) #true)

Recall the unary function 'random'.
- random : number -> number
  (random n) produces a random number from (range 0 n 1), in other words a random number
  from 0, 1, 2, ..., n-1.

Consider:
(map random (range 101 104 1))
The following are the steps, but because of the strange behaviour of 'random'
(it's a function that's not completely determined by what you give it), this will
probably produce different lists when you run it.
(map random (list 101 102 103))
(list (random 101) (random 102) (random 103))

This will probably produce #false, but not always:
(apply < (map random (range 101 104 1)))

Question.
Suppose the following definition is run:
(define some-numbers (list -3 5 -7))
Show the steps to produce the result of:
(apply < some-numbers)
Answer.
(apply < (list -3 5 -7)) ; Step.
(< -3 5 -7) ; Step.
#false ; Result value.

Question. Show the steps to produce the result of:
(apply < (map sqr some-numbers))
Answer.
(apply < (map sqr (list -3 5 -7))) ; Step.
(apply < (list (sqr -3) (sqr 5) (sqr 7))) ; Step.
(apply < (list 9 25 49)) ; Step.

Exercise.
Show the steps to produce the result of:
(apply min some-numbers)

Exercise.
Show the steps to produce the result of:
(apply min (map - some-numbers))
; Mapping then applying is a common use of 'apply'.
;
; Here's a unary function we made that takes a height and produces a solid maroon ellipse that is
; 10 pixels wide and the given height.
;
; oval : number -> image
(define (oval height)
  (ellipse 10 height "solid" "maroon"))

; Mapping that with a range can make lots of ellipses.
(check-expect (map oval (range 1 6 1))
  (map oval (list 1 2 3 4 5)))
(check-expect (map oval (range 1 6 1))
  (list (oval 1) (oval 2) (oval 3) (oval 4) (oval 5)))

; This makes a list of 19 ellipses with heights 1, 4, 9, 16, ..., 19*19.
(map oval (map sqr (range 1 20 1))))

; Applying 'beside' makes that into one (somewhat interesting) image:
(apply beside (map oval (map sqr (range 1 20 1))))

; I have the urge to make a hat (or a roof?):
(rotate -90 (apply beside (map oval (map sqr (range 1 20 1))))))

; Exercise.
; Show the steps to to produce the result of the following expression:
(rotate -90 (apply beside (map oval (map sqr (range 4 7 1))))))

; Exercise.
; Design and define a function 'flat-oval' that takes the width for an ellipse of height 10 pixels,
; and use that to make the hat again without using 'rotate'.

; Exercise.
; Design and define a function that makes an ellipse at a given angle.
; Use that to make the following images: