Goals of this tutorial
You should be able to...

• understand and write data definitions for lists.
• understand and use the templates for processing lists to write recursive functions consuming this type of data.
• step through list functions.

Review: List data definition
;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))

From the data definition, a list of values of type X is either empty, or it consists of a first value followed by a list of values (the rest of the list).

This is a recursive definition. It contains a base case, and a recursive (self-referential) case.

Recursive types should be processed with recursive functions.
Review: List Templates and Data Definitions
Recall the data definition and template for lists of Numbers from class:

;; A (listof Num) is one of:
;; ⋆ empty
;; ⋆ (cons Num (listof Num))

;; listof-num-template: (listof Num) → Any
(define (listof-num-template lonum)
  (cond
   [(empty? lonum) . . .
    [else . . .]]))

CQ: Which of the following goes in the blank for the listof-num-template?

A (… (first lonum) . . . (rest lonum) . . .)
B (… (first lonum) . . . (listof-num-template lonum) . . .)
C (… (rest lonum) . . . (listof-num-template (first lonum)) . . .)
D (… (first lonum) . . . (listof-num-template (rest lonum)) . . .)
E (… (first lonum) . . . (listof-num-template (rest lonum)) . . . lonum . . .)

Group Problem: sum-ints
Based on the previous template, write a function sum-ints that consumes a list of numbers and produces the sum of all the integers in the list. Include the full design recipe. The sum of the integers for the empty list is 0.

;; listof-num-template: (listof Num) → Any
(define (listof-num-template lonum)
  (cond
   [(empty? lonum) . . .
    [else . . . (first lonum) . . .
      (listof-num-template (rest lonum)) . . .]])
Base Case(s)

- What is the simplest value that the function can consume?
  - For a list, this is typically the empty list.

- What do I do when I reach the base case(s)?
  - Think about what your function produces, as indicated in your contract. If you are producing a list, your base case(s) will often produce the empty list.

- Questions will sometimes specify what to do in the base case(s).

  \[(\text{empty? lonum)} \ 0\]

Recursive Call(s)

- At least one of the arguments should change.

- Your recursive calls should be on smaller inputs, according to the data definition, and bring you closer to the base case(s).

- With lists, this typically means taking the rest of the list.

- If you have multiple recursive calls, the way that arguments change in each recursive call may differ from one another.

  \[(\text{sum-ints (rest lonum)})\]

Dealing with the First Element

- We know that the list is non-empty.

- You may want to check some property of \((\text{first lst})\).

- Your function may produce something using \((\text{first lst})\).

- Think about how to combine what you do with the first element with the result you get from the recursive function application on the rest of the list.

  \[(\text{integer? (first lonum)}) \ (+ \ (\text{first lonum}) \ . \ . \ .)]\]
Group Problem: list-replace
Write a function list-replace that consumes a target value, a replacement value, and a (listof Any). The function produces a new list, which is identical to the consumed list, except all occurrences of the target value (if any) are replaced with the replacement value. Include the full design recipe.

(list-replace 5 "five" (cons 5 (cons 'a (cons 5 empty))))
⇒ (cons "five" (cons 'a (cons "five" empty)))

(list-replace "five" 5 (cons 5 (cons 'a empty)))
⇒ (cons 5 (cons 'a empty))

Demonstration: Condensed Trace
(define (list-replace target repl loany)
  (cond
   [(empty? loany) empty]
   [(equal? (first loany) target)
    (cons repl (list-replace target repl (rest loany)))]
   [else (cons (first loany) (list-replace target repl (rest loany)))]))

Using our definition of list-replace, we will perform a condensed trace of:
(list-replace "bag" 'foo
  (cons "bag" (cons 86 (cons false (cons "bag" empty)))))
⇒ (cons 'foo (list-replace "bag" 'foo
  (cons 86 (cons false (cons "bag" empty))))
\begin{verbatim}
(list-replace "bag" 'foo
  (cons "bag" (cons 86 (cons false (cons "bag" empty))))))
⇒ (cons 'foo (list-replace "bag" 'foo
  (cons 86 (cons false (cons "bag" empty))))))
⇒ (cons 'foo (cons 86 (list-replace "bag" 'foo
  (cons false (cons "bag" empty))))))
⇒ (cons 'foo (cons 86 (cons false (list-replace "bag" 'foo
  (cons "bag" empty))))))
\end{verbatim}
Group Problem: has-upper-case?
Write a function `has-upper-case?` that consumes a string, and produces true if at least one of the characters in the consumed string is an upper-case character, and false otherwise. Note that the empty string does not contain any upper-case characters.

The built-in function `char-upper-case?` may be useful. You may not use any built-in string functions other than `string->list` and `list->string`.

You will find it helpful to make `has-upper-case?` a wrapper function that simply calls a helper function. Include the full design recipe for `has-upper-case?`.

Group Problem: length-upper-strings
Write a function `length-upper-strings` that consumes a list of strings, and produces the total lengths of the strings in the consumed list that contain at least one upper-case character.

You may find it helpful to use the `has-upper-case?` function that you just wrote. In particular, you may not use any built-in string functions other than `string-length`, `string->list`, and `list->string`. 

```
(list-replace "bag" 'foo
  (cons "bag" (cons 86 (cons false (cons "bag" empty))))))
⇒ (cons 'foo (list-replace "bag" 'foo
  (cons 86 (cons false (cons "bag" empty))))))
⇒ (cons 'foo (cons 86 (list-replace "bag" 'foo
  (cons false (cons "bag" empty))))))
⇒ (cons 'foo (cons 86 (cons false (list-replace "bag" 'foo
  (cons "bag" empty))))))
⇒ (cons 'foo (cons 86 (cons false (cons 'foo
  (list-replace "bag" 'foo empty))))))
⇒ (cons 'foo (cons 86 (cons false (cons 'foo empty))))))
```

```
Group Problem: has-upper-case?
Write a function `has-upper-case?` that consumes a string, and produces true if at least one of the characters in the consumed string is an upper-case character, and false otherwise. Note that the empty string does not contain any upper-case characters.

The built-in function `char-upper-case?` may be useful. You may not use any built-in string functions other than `string->list` and `list->string`.

You will find it helpful to make `has-upper-case?` a wrapper function that simply calls a helper function. Include the full design recipe for `has-upper-case?`.

Group Problem: length-upper-strings
Write a function `length-upper-strings` that consumes a list of strings, and produces the total lengths of the strings in the consumed list that contain at least one upper-case character.

You may find it helpful to use the `has-upper-case?` function that you just wrote. In particular, you may not use any built-in string functions other than `string-length`, `string->list`, and `list->string`.

```
(length-upper-strings (cons "CS135" (cons "fourth"
  (cons "Tutorial" empty))))
⇒ 13
(length-upper-strings (cons "testing" (cons "one-two" empty))) ⇒ 0
```