Goals of this tutorial

You should be able to...

- understand and write data definitions for lists
- understand and use the template for processing lists to write recursive functions consuming this type of data.
Group Problem: Mixed Data
Before we start reviewing lists, let’s review an example of a problem dealing with mixed data. Consider the following data definitions:

(define-struct pen (price colour))
;; A Pen is a (make-pen Num Sym)

(define-struct apple (price juicy?))
;; An Apple is a (make-apple Num Bool)

;; An Item is one of:
;; * Pen
;; * Apple
Group Problem: Mixed Data

Write a function `process` which consumes an `Item` and produces the colour of `item` if it is a pen and produces whether it is juicy if `item` is an Apple.

Provide a contract.
Review: List data definition

;; A (listof X) is one of:

;; ★ empty

;; ★ (cons X (listof X))

From the data definition, a list of values of any type 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: Basic list constructs

- **empty**: A value representing a list with 0 items.
- **cons**: Consumes an item and a list and produces a new, longer list.
- **first**: Consumes a nonempty list and produces the first item.
- **rest**: Consumes a nonempty list and produces the same list without the first item.
- **empty?**: Consumes a value and produces `true` if it is `empty` and `false` otherwise.
- **cons?**: Consumes a value and produces `true` if it is a `cons` value and `false` otherwise.
Review: Substitution rules

If \( a, b, c \) are values and \( c \) is non-empty:

\[(\text{first } (\text{cons } a \ b)) \Rightarrow a\]
\[(\text{rest } (\text{cons } a \ b)) \Rightarrow b\]
\[(\text{empty? } \text{empty}) \Rightarrow \text{true}\]
\[(\text{empty? } c) \Rightarrow \text{false}\]
\[(\text{cons? } (\text{cons } a \ b)) \Rightarrow \text{true}\]
\[(\text{cons? } a) \Rightarrow \text{false}\]
Review: List Templates and Data Definitions

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

;; listof-X-template: (listof X) → Any

(define (listof-X-template loX)
  (cond
    [(empty? loX) . . . ]
    [else _____________________________ ]))

A  (... (first loX) ... (rest loX) ... )
B  (... (first loX) ... (listof-X-template loX) ... )
C  (... (rest loX) ... (listof-X-template (first loX)) ... )
D  (... (first loX) ... (listof-X-template (rest loX)) ... )
E  (... (first loX) ... (listof-X-template (rest loX)) ... loX ... )
Group Problem - sum-num

As a warmup, based on the previous template, write a function `sum-num` that consumes a list of numbers and produces the sum of those numbers. Provide contract and examples.
Group Problem - longer-str

Write a function longer-str that consumes a list of strings and a target string and produces the number of strings in the list that have length greater than the target string. Provide contract and examples.

Hint: (string-length x) produces the length of the string x.
Group Problem - strings-equal?

Based on the previous template for list of X, write a function `strings-equal?` that consumes a list of strings and produces `true` if all of the strings are equal, and `false` otherwise. Include the contract and examples.

Hint: The template includes only one base case, but sometimes functions need multiple base cases.
Group Problem - list of Item

Recall:

(define-struct pen (price colour))

;; A Pen is a (make-pen Num Sym)

(define-struct apple (price juicy?))

;; An Apple is a (make-apple Num Bool)

;; An Item is one of:

;; * a Pen

;; * an Apple
Group Problem - list of Item

Write a function `total-price` that takes in a list of Items and produce the total price of those Items. Provide a contract.
Group Problem - list of Item

Next, write a function lon->loapp that takes in a list of positive numbers and produces a list of juicy apples with prices being the numbers in the consumed list, in the same order. Provide a contract.
Group Problem - list of Item

Lastly, let’s write a function `remove-pen` that takes in a list of pens and another pen and should return a new list, with all the pens in the consumed list that resembles the second argument removed from the original list. Provide a contract.