If you have not already, make sure you

- Read the Wikipedia entry on *Higher-order functions*.
Working with more than one item at once

So far we have written only functions that consume one or a few values, and may combine them in various ways.

More often we have a collection of data to process.

Racket is a dialect of LISP, which was originally designed for LIST Processing.

Our principal way of grouping values is the list.
What is a list?

The word *list* comes from Old English “líste”, meaning a strip (such a strip of cloth or paper).

> “His targe wíþ gold list He carf atvo.”
> (Guy of Warwick, ca. 1330)

→ A strip of paper with items written on it.
→ An ordered collection of items.

We can make a list really easily:

```scheme
(define wishes
  (list "comics" "turtle figures"
        "Donkey Kong" "Play-Doh Burger King"))
```

```scheme
(define primes (list 2 3 5 7 11 13 17 19))
```
A value may be a list

Lists behave just like any other value.

We can define constants which are lists:

```scheme
(define wishes
  (list "comics" "turtle figures"
       "Donkey Kong" "Play-Doh Burger King"))

(define primes (list 2 3 5 7 11 13 17 19))
```

We can have functions consume lists:

```scheme
(length wishes) => 4
(first wishes) => "comics"
(rest wishes) => (list "turtle figures" "Donkey Kong" "Play-Doh Burger King")
```

We can have functions return lists:

```scheme
(range 4 16 2) => (list 4 6 8 10 12 14)
(append (list 6 7 42) (list 3 5 15)) => (list 6 7 42 3 5 15)
```
Lists and the design recipe

In the design recipe, we specify the type of values in a list as follows:

- **Use** `(listof Type)` for a single type.
  
  `(listof Nat)` describes a list containing zero or more `Nat`. E.g. `(list 6 7 42)`
  
  `(listof Str)` describes a list containing zero or more `Str`. E.g. `(list "hello" "world")`

- **If a list may contain more than one type**, use `(listof (anyof Type1 Type2 ...))`.
  
  `(listof (anyof Num Str))` describes a list containing zero or more values, each of which is either a `Num` or a `Str`. E.g. `(list 3.14 "pie" "forty-two" -17)`

- **If a list is of known length and types**, use `(list Type1 Type2 ...)`.
  
  `(list Nat Str)` describes a list containing two values. The first value is a `Nat`, and the second value is a `Str`. E.g. `(list 6 "foo")`
  
  `(list "foo" 6)` is not a `(list Nat Str)`. It is a `(list Str Nat)`.
Transforming items in a list using map

We can store data in a list, but what can we do with them?

Use map to transform each item in a list, using a function.

(map F (list x0 x1 x2 ... xn)) ⇒ (list (F x0) (F x1) (F x2) ... (F xn))

(map sqr (list 2 3 5)) ⇒ (list (sqr 2) (sqr 3) (sqr 5)) ⇒ (list 4 9 25)

(define (double x) (+ x x))

(define (double-each L)
  (map double L))

(double-each (list 0 1 2 3 4)) ⇒ (list 0 2 4 6 8)
To use `map` on a list of values of some type:
write a function that consumes one single value of that type and transforms it as required.

I wish to transform each item in a list by \( f(x) = 10\sqrt{x} \):

```scheme
;; (10rootx n) return 10*sqrt(x)
;; 10rootx: Num -> Num
;; Requires: n >= 0
;; Examples:
(check-expect (10rootx 49) 70)
(define (10rootx x) (* 10 (sqrt x)))
```

```scheme
;; (10rootx-each L) return a list containing 10*sqrt(x) for each x in L.
;; 10rootx-each: (listof Num) -> (listof Num)
;; Requires: each value is >= 0
;; Examples:
(check-expect (10rootx-each (list 49 81 100)) (list 70 90 100))
(define (10rootx-each L) (map 10rootx L))
```
Working with map

To use map on a list of values of some type:
write a function that consumes one single value of that type and transforms it as required.

Exercise
Digital signals are often recorded as values between 0 and 255, but we often prefer to work with numbers between 0 and 1.
Write a function (squash-range L) that consumes a (listof Nat), and returns a (listof Num) so numbers on the interval [0, 255] are scaled to the interval [0, 1].
(squash-range (list 0 204 255)) => (list 0 0.8 1)

Exercise
Write a function that consumes a (listof Str), where each Str is a person’s name, and returns a list containing a greeting for each person.
(greet-each (list "Ali" "Carlos" "Sai")) => (list "Hi Ali!" "Hi Carlos!" "Hi Sai!")
Using `range` to build lists

`range` start end step returns the list that starts at start, and steps by step until just before it reaches end. This allows us to build new lists.

```
(range 4 10 1) => (list 4 5 6 7 8 9)
(range 4 10 2) => (list 4 6 8)
(range 20 8 -3) => (list 20 17 14 11)
(range 20 8 3) => '(); the empty list
```

To work with `range` and `map`:

1. get proper values from `range`; test it.
2. use `map` to transform these values as needed.

**Exercise**

Complete the function `list-cubes`.

```
;; (list-cubes n) return the list of cubes from 1*1*1 to n*n*n.
;; list-cubes: Nat -> (listof Nat)
;; Examples:
(check-expect (list-cubes 4) (list 1 8 27 64))
```
Summarizing a list using \textit{foldr}

\texttt{range} lets us create a list, and \texttt{map} lets us transform each item. What if I want to my result to be a combination of the items in the list, instead of the entire list?

What is the total of all the values in \((\texttt{list} \ 6 \ 5 \ 8 \ 5 \ 14 \ 4)\)?

\[
(+ \ 6 \ (+ \ 5 \ (+ \ 8 \ (+ \ 5 \ (+ \ 14 \ 4)))))) \Rightarrow 42
\]

To do this automatically, there is another function, \texttt{foldr}, meaning “fold right”.

\[
\texttt{(foldr \ F \ base \ (list \ x0 \ x1 \ ... \ xn))} \Rightarrow \texttt{(F \ x0 \ (F \ x1 \ (F \ ... \ (F \ xn \ base)))))}
\]

What does this mean?

We combine items, starting from the right, each time creating a new item to combine with.

\[
\texttt{(foldr \ + \ 0 \ (list \ 6 \ 5 \ 8 \ 5 \ 14 \ 4))}
\Rightarrow \texttt{(+ \ 6 \ (+ \ 5 \ (+ \ 8 \ (+ \ 5 \ (+ \ 14 \ (+ \ 4 \ 0))))))}
\Rightarrow \texttt{42}
\]
Strategy for working with foldr

\[
(f \text{oldr } F \text{ base } (\text{list } x0 \ x1 \ldots \ xn)) \Rightarrow (F \ x0 \ (F \ x1 \ (F \ldots \ (F \ xn \ \text{base}))))
\]

1. Figure out what the answer is when the list is empty. Use this as the base.
2. Write a function that consumes two values, new and old, where new is a value from the list, and old is an answer.

For example: consider finding the sum of items in a (listof Num).

1. The sum of nothing is zero, so the base is 0.
2. To calculate the sum of a value and another sum, just add the two values.

\[
\begin{align*}
\text{(define } & (\text{add } a \ b) \ (+ \ a \ b)) \\
\text{(define } & (\text{sum } L) \ (\text{foldr } \text{add } 0 \ L)) \\
(\text{sum } & () \Rightarrow 0) \\
(\text{sum } L & \Rightarrow (\text{add } 5 \ (\text{add } 8 \ (\text{add } 4 \ 0))) \Rightarrow 17 \\
\text{(We could use the built-in function } & +.\text{)}
\end{align*}
\]
Working with foldr

\[
(foldr \ F \ base \ (list \ x_0 \ x_1 \ ... \ x_n)) \Rightarrow (F \ x_0 \ (F \ x_1 \ (F \ ... \ (F \ x_n \ base))))
\]

1. Figure out what the answer is when the list is empty. Use this as the base.
2. Write a function that consumes two values, new and old, where new is a value from the list, and old is an answer.

Exercise

Write a function prod that returns the product of a \((listof \ Num)\).

(prod \ (list \ 2 \ 2 \ 3 \ 5)) \Rightarrow 60

Exercise

Write a function that returns the number of odd numbers in a \((listof \ Nat)\).

Hint: read the documentation on remainder.

Can you do this using \(map\) and \(foldr\)? Just using \(foldr\)?
Experiment with \texttt{fold-sub}. Describe how it behaves, and why.
\begin{verbatim}
(define (fold-sub L) (foldr - 0 L))
(fold-sub (list 6 5 2)) => ?
\end{verbatim}

Read the documentation on \texttt{string-length}.
Write a function that returns the total length of all the values in a \texttt{(listof Str)}.
\begin{verbatim}
(total-length (list "hello" "how" "r" "u"?")) => 11
\end{verbatim}
Exercises

Write a function that returns the average (mean) of a non-empty \(\text{listof Num}\).

\[
\text{average (list 2 4 9)) => 5} \\
\text{(average (list 4 5 6 6) => 5.25)}
\]

Recall that \(\text{length } L\) returns the number of values in \(L\).

The factorial function, \(n!\), returns the product of the numbers from 1 to \(n\). For example, \(4! = 1 \times 2 \times 3 \times 4 = 24\).

Write a function \(\text{factorial } n\) that returns \(n!\).

\[
\text{(factorial 5) => 120} \\
\text{(factorial 1) => 1}
\]
Write a function (sum-square-difference n) that consumes a Nat and returns the difference between the square of the sum of numbers from 0 to n, and the sum of the squares of those numbers.

\[(\text{sum-square-difference } 3) \Rightarrow (- (\text{sqr} (+ 0 1 2 3)) \ (\text{+} 0 1 4 9)) \Rightarrow 22\]
Multi-argument \texttt{map}

So far we have used \texttt{map} only with functions that consume a single value: like \((\text{map} \, F \, L)\), where \(F\) is a single-parameter function and \(L\) is a list. But \texttt{map} can do so much more! \texttt{map} works with any number of lists, all of the same length: \((\text{map} \, F \, L_1 \, L_2 \ldots)\)

For example, if we have two lists of equal length we can make a new list where the first value is the sum of the first values, the second is the sum of the second values, and so on.

\begin{verbatim}
;; (elementwise-sum \(L \, M)\) add each value in \(L\) to the corresponding value in \(M\).
;; elementwise-sum: (listof Int) (listof Int) --> (listof Int)
;; Requires: \(L\) and \(M\) are of equal length.
;; Examples:
(check-expect (elementwise-sum (\text{list} 2 3 3) (\text{list} 7 4 1)) (\text{list} 9 7 4))
\end{verbatim}

\begin{verbatim}
(define (elementwise-sum \(L \, M)\) (map + \(L \, M)))
\end{verbatim}

**Exercise**

Write a function \((\text{absdiff} \, a \, b)\) that consumes two \((\text{listof Int})\) and returns a \((\text{listof Nat})\) containing the absolute value of the difference between corresponding values.

\((\text{absdiff} \, (\text{list} 1 \, 3 \, 5 \, 7) \, (\text{list} 7 \, 3 \, 6 \, 1)) \Rightarrow (\text{list} 6 \, 0 \, 1 \, 6)\)
Multi-argument map

Recall that the distance of a point \((x, y)\) from \((0, 0)\), by the Pythagorean theorem, is

\[ \sqrt{x^2 + y^2} \]

You may use the \texttt{sqrt} function to compute the square root. \((\texttt{sqrt} 4) \Rightarrow 2\).

\begin{exercise}
Write a function \((\texttt{distances} \texttt{xs} \texttt{ys})\) that consumes two lists: the first contains \(x\) values, and the second contains \(y\) values. The output is a list containing the distance of each point \((x, y)\) from \((0, 0)\).

\((\texttt{distances} \texttt{(list 3 0 2)} \texttt{(list 4 7 2)}) \Rightarrow \texttt{(list 5 7 #i2.828427)}\)

\((\texttt{Since (3, 4) is at distance 5; (0, 7) is at distance 7; and (2, 2) is at distance } \sqrt{8} \approx 2.828427.)\)
\end{exercise}
Here is one solution:

;; (distance x y) return the distance from (x, y) to the origin.
;; distance: Num Num -> Num
;; Example:
(check-expect (distance 3 4) 5)

(define (distance x y)
  (sqrt (+ (sqr x) (sqr y))))

;; (distances xs ys) return the list of distances for each of xs, ys.
;; distances: (listof Num) (listof Num) -> (listof Num)
;; Example:
(check-within (distances (list 3 0 2) (list 4 7 2))
  (list 5 7 2.8284) 0.0001)

(define (distances xs ys)
  (map distance xs ys))
Multi-argument map

Suppose we have two (listof Str): one of first names, and one of matching last names:

```
(define gnames (list "Joseph" "Burt" "Douglas" "James" "David"))
(define snames (list "Hagey" "Matthews" "Wright" "Downey" "Johnston"))
```

Exercise

Complete join-names.

```
;; (join-names G S) Make a list of full names from G and S.
;; join-names: (listof Str) (listof Str) -> (listof Str)
;; Example:
(check-expect (join-names gnames snames)
  (list "Joseph Hagey" "Burt Matthews" "Douglas Wright"
       "James Downey" "David Johnston"))
```
Complete enumerate-words.

;; (enumerate-words L) format the values in L with their index, like:
;; 1. first item
;; 2. second item
;; 3. third item
;; enumerate-words: (listof Str) -> (listof Str)
;; Examples:
(check-expect (enumerate-words (list "Mercury" "Venus" "Earth" "Mars"
                                    "Jupiter" "Saturn" "Uranus" "Neptune"))
               (list "1. Mercury" "2. Venus" "3. Earth" "4. Mars"

Hint
Use range to make a list containing the needed numbers.
Module Summary

- Start storing information in lists, and describe lists in contracts.
- Transform list values using `map`, and `foldr`.
- Construct new lists using `range`, especially in combination with `map`.
- Use `foldr` to combine a list to a single value. This can be especially powerful when combined with `map`.
- Understand the use of `anyof` and be able to use it in your design recipes.

Before we begin the next module, please

- Read *How to Design Programs*, Section 4.