If you have not already, make sure you

- Read the Wikipedia entry on *Higher-order functions*. 
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*. 

---

**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 wip 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"))

(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:

```
(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:

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

We can have functions return lists:

```
(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 \texttt{map}

We can \textit{store} data in a list, but what can we \textit{do} with them?

Use \texttt{map} to transform each item in a list, using a function.

\begin{equation}
\text{(map } F \ (\text{list } x_0 \ x_1 \ x_2 \ldots \ x_n)) \Rightarrow (\text{list } (F \ x_0) \ (F \ x_1) \ (F \ x_2) \ldots \ (F \ x_n))
\end{equation}

\begin{equation}
(\text{map } \text{sqr} \ (\text{list } 2 \ 3 \ 5)) \Rightarrow (\text{list } (\text{sqr} \ 2) \ (\text{sqr} \ 3) \ (\text{sqr} \ 5)) \Rightarrow (\text{list } 4 \ 9 \ 25)
\end{equation}

\begin{equation}
(\text{define } \ (\text{double } x) \ (+ \ x \ x))
\end{equation}

\begin{equation}
(\text{define } \ (\text{double-each } L)
\begin{align*}
  & (\text{map } \text{double} \ L)) \\
\end{align*}
\end{equation}

\begin{equation}
(\text{double-each } (\text{list } 0 \ 1 \ 2 \ 3 \ 4)) \Rightarrow (\text{list } 0 \ 2 \ 4 \ 6 \ 8)
\end{equation}
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 \( f(x) = 10\sqrt{x} \):

\[
\begin{align*}
; &; \text{(10rootx n) return } 10*\sqrt{x} \\
; &; 10\text{rootx: } \text{Num} \rightarrow \text{Num} \\
; &; \text{Requires: } n \geq 0 \\
; &; \text{Examples:}
\end{align*}
\]

(check-expect (10rootx 49) 70)  
(define (10rootx x) (* 10 (sqrt x)))

\[
\begin{align*}
; &; \text{(10rootx-each L) return a list containing } 10*\sqrt{x} \text{ for each } x \text{ in } L. \\
; &; 10\text{rootx-each: (listof Num)} \rightarrow \text{(listof Num)} \\
&; \text{Requires: each value is } \geq 0 \\
&; \text{Examples:}
\end{align*}
\]

(check-expect (10rootx-each (list 49 81 100)) (list 70 90 100))  
(define (10rootx-each L) (map 10rootx L))
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.

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)
```

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. `n
`;; list-cubes: Nat -> (listof Nat)
`;; Examples:
(check-expect (list-cubes 4) (list 1 8 27 64))

CS 115 Module 3: Big Data – Working with Lists
Summarizing a list using `foldr`

`range` lets us create a list, and `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 `list 6 5 8 5 14 4`?

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

To do this automatically, there is another function, `foldr`, meaning “fold right”.

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

What does this mean?

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

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

`foldr F base (list x0 x1 ... xn)) => (F x0 (F x1 (F ... (F xn 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.

```
(define (add a b) (+ a b))
(define (sum L) (foldr add 0 L))
(sum '()) => 0
(sum (list 5 8 4)) => (add 5 (add 8 (add 4 0))) => 17
```
(We could use the built-in function `+`.)
Figure out what the answer is when the list is empty. Use this as the base.

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).

(\(\text{prod (list 2 2 3 5)}\) => 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`?
Exercises

Exercise

Experiment with fold-sub. Describe how it behaves, and why.

(define (fold-sub L) (foldr - 0 L))
(fold-sub (list 6 5 2)) => ?

Exercise

Write a function flatten that consumes a (listof (listof Any)) and returns a list containing all the values in the lists.

(flatten (list (list 1 2) (list 3 4) (list 7))) => (list 1 2 3 4 7)
Hint: read the documentation on append.

Exercise

Read the documentation on string-length.
Write a function that returns the total length of all the values in a (listof Str).
Exercises

Exercise

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

\(\text{average (list 2 4 9))} \Rightarrow 5\)
\(\text{average (list 4 5 6 6)} \Rightarrow 5.25\)

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

Exercise

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 \(n!\).

\(\text{factorial 5} \Rightarrow 120\)
\(\text{factorial 1} \Rightarrow 1\)
Write a function \((\text{sum-square-difference } n)\) that consumes a \textbf{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.

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

\(\text{square of the sum} \quad \text{sum of the squares}\)
Multi-argument map

So far we have used 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 map can do so much more!

map works with any number of lists, all of the same length: \((\text{map} \ F \ L1 \ L2 \ldots)\)

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.

;;; (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 \((\text{list} \ 2 \ 3 \ 3)\) \((\text{list} \ 7 \ 4 \ 1)\)) \((\text{list} \ 9 \ 7 \ 4)\))

(define (elementwise-sum \(L\) \(M\)) (map + \(L\) \(M\)))

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.

(absdiff \((\text{list} \ 1 \ 3 \ 5 \ 7)\) \((\text{list} \ 7 \ 3 \ 6 \ 1)\)) => \((\text{list} \ 6 \ 0 \ 1 \ 6)\)
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 `sqrt` function to compute the square root. \((\text{sqrt} 4) \Rightarrow 2\).

**Exercise**

Write a function \((\text{distances } xs \ ys)\) 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 \((0, 0)\).

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

\((\text{Since } (3, 4) \text{ is at distance } 5; (0, 7) \text{ is at distance } 7; \text{ and } (2, 2) \text{ is at distance } \sqrt{8} \approx 2.828427.)\)
Multi-argument map

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.