If you have not already, make sure you

- Read *How to Design Programs*, Section 4.
Booleans (Bool)

<, >, <=, >=, =, and **equal?** are new functions, each of which returns a Boolean value (Bool).

\[
\begin{align*}
(\lt 4 6) & \iff 4 < 6 \\
(\gt 4 6) & \iff 4 > 6 \\
(= 5 7) & \iff 5 = 7 \\
(\geq 5 5) & \iff 5 \geq 5 \\
(\leq 5 5) & \iff 5 \leq 5
\end{align*}
\]

Each returns **#true** or **#false**. These are the only values a **Bool** may take.

(You may see **#true** called **true** or **#t**, and see **#false** called **false** or **#f**. They are identical.) A function which returns a **Bool** is called a **predicate**. For many predicates in Racket, the name ends with **?**, as in **string=?**. Other predicates include **even?**, **odd?**, and **equal?**.)
A sin-squared window, used in signal processing, can be described by the following piecewise function:

\[
f(x) = \begin{cases} 
0 & \text{for } x < 0 \\
1 & \text{for } x \geq 1 \\
\sin^2(x\pi/2) & \text{for } 0 \leq x < 1
\end{cases}
\]

Racket gives us an easy way to design such things in a special form called \texttt{cond}.
Using `cond` to describe cases

\[ f(x) = \begin{cases} 
  0 & \text{for } x < 0 \\
  1 & \text{for } x \geq 1 \\
  \sin^2(x \pi/2) & \text{for } 0 \leq x < 1 
\end{cases} \]

\[ \text{(define (ssqw x)} \]
\[ \text{ (cond \]
\[ [(< x 0) 0] \]
\[ [>= x 1] 1] \]
\[ [(< x 1) (sqr (sin (* x pi 0.5)))])]) \]

`cond` is a special form, not a function. We deal with it in a special way. In particular, do not evaluate its arguments until necessary.

Each argument of `cond` is a pair of square brackets around a pair of expressions: [question answer].
Evaluating a cond statement

How do we evaluate a cond?

**Informally**, evaluate a cond by considering the question/answer pairs in order, top to bottom. When considering a question/answer pair, evaluate the question. If the question evaluates to #true, the whole cond returns the answer.

```
define (ssqw x)  
  (cond  
    [(< x 0) 0]  
    [(>= x 1) 1]  
    [(< x 1) (sqr (sin (* x pi 0.5)))]))
```

For example consider, (ssqw 4).

=> (cond [(< 4 0) 0]  
      [(>= 4 1) 1]  
      [(< 4 1) (sqr (sin (* 4 pi 0.5)))]))

The first question is #false, but the second question is #true!

=> 1
No satisfied questions

What happens if none of the questions evaluate to \texttt{#true}?

\begin{verbatim}
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [(< n 0) (- n)])
)
\end{verbatim}

An error occurs if we try to run \texttt{(absolute-value 0)}

This can be helpful — if we try to consider all the possibilities, but we miss one, testing may raise this error. Then we can fix it. But sometimes we want to only describe some conditions, and do something different if none of them are satisfied.
We *could* use a question which always evaluates to `#true`:

```lisp
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [#true (- n)]))
```

Remember: the question/answer pairs are considered *in order*, top to bottom, and it stops as soon as it finds a question which evaluates to `#true`.

This is useful sufficiently frequently that there is special keyword for it: `else`.

```lisp
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [else (- n)]))
```
Recall we are imagining interpreting our programs as a series of substitutions, called a trace. How do we formally trace cond?
The general form of a conditional is

\[
\text{(cond } [\text{question1 answer1}] [\text{question2 answer2}] \ldots [\text{questionk answerk}])
\]

To evaluate the conditional, evaluate question1, then perform the following substitutions:

- \( (\text{cond} \ [\text{false exp0}] [\text{exp1 exp2}] \ldots) \Rightarrow (\text{cond} \ [\text{exp1 exp2}] \ldots) \)
- \( (\text{cond} \ [\text{true exp0}] [\text{exp1 exp2}] \ldots) \Rightarrow \text{exp0} \)
- \( (\text{cond} \ [\text{else exp0}]) \Rightarrow \text{exp0} \)
Tracing cond example

- `(cond [#false exp0][exp1 exp2]...) => (cond [exp1 exp2]...)`
- `(cond [#true exp0][exp1 exp2]...) => exp0`
- `(cond [else exp0]) => exp0`

```lisp
(define (ssqw x) ...)
(ssqw 0)
=> (cond [(< 0 0) 0] [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
=> (cond [#false 0] [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
=> (cond [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
=> (cond [#false 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
=> (cond [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
=> (cond [#true (sqr (sin (* 0 pi 0.5)))])
=> (sqr (sin (* 0 pi 0.5)))
=> (sqr (sin 0))
=> (sqr 0)
=> 0
```
(define (qux a b)
  (cond
    [(= a b) 42]
    [(> a (+ 3 b)) (* a b)]
    [(> a b) (- b a)]
    [else -42]))

(qux 5 4)

Perform a complete trace of this program.
You should write tests:
  - so each question is evaluated to \texttt{true} at least once, to verify each answer is tested.
  - to test boundaries; it is easy to get “off-by-one” errors!

Suppose I wanted a function which returns 0 for integers \( x < 0 \), 1 for integers \( 0 \leq x \leq 10 \), and 2 for other integers. What should I test?
I should check boundaries \((-1, 0, 1)\) and \((10, 11)\), some other negative number, and some larger number.

```scheme
;; (categorize n) return 0 for negative, 1 for non-negative <= 10, 2 otherwise.
(define (categorize n)
  (cond [(< n 0) 0]
        [(<= n 10) 1]
        [else 2]))
```

;; Tests:
(check-expect (categorize -5) 0)
(check-expect (categorize -1) 0)
(check-expect (categorize 1) 1)
(check-expect (categorize 10) 1)
(check-expect (categorize 11) 2)
(check-expect (categorize 50) 2)
(check-expect (categorize 0) 1)
Conditionals can be very useful in combination with map.

;; (fix-limit val) replace val with 20 if it is greater than 20, and with 10 if it is lower than 10.
;; fix-limit: Num -> Num
;; Example:
(check-expect (fix-limit 5) 10)

(define (fix-limit val)
  (cond [(> val 20) 20]
        [(< val 10) 10]
        [else val]))

;; (fix-list L) Replace each value in L with 20 if it is greater than 20, and with 10 if it is lower than 10.
;; fix-list: Num Num (listof Num) -> (listof Num)
;; Example:
(check-expect (fix-list (list 8 12 18 22)) (list 10 12 18 20))

(define (fix-list L)
  (map fix-limit L))
Conditionals can be very useful in combination with map.

;;; (fix-limit val) replace val with 20 if it is greater
;;; than 20, and with 10 if it is lower than 10.
(define (fix-limit val)
    (cond [(> val 20) 20]
          [(< val 10) 10]
          [else val]))

;;; (fix-list L) Replace each value in L with 20 if it is greater
;;; than 20, and with 10 if it is lower than 10.
(define (fix-list L)
    (map fix-limit L))

Exercise
Using cond and map, write a function neg-odd that consumes a (listof Nat). The function returns a (listof Int) where all odd numbers are negative, and all even numbers positive.
(neg-odd (list 2 5 8 11 14 17)) => (list 2 -5 8 -11 14 -17)
We combine predicates using the special forms \texttt{and}, \texttt{or}, and the function \texttt{not}. These all consume and return \texttt{Bool} values.

- \texttt{and} returns \texttt{#false} if at least one of its arguments is \texttt{#false}, and \texttt{#true} otherwise.
- \texttt{or} returns \texttt{#true} if at least one of its arguments is \texttt{#true} and \texttt{#false} otherwise.
- \texttt{not} returns \texttt{#true} if its argument is \texttt{#false}, and \texttt{#false} if its argument is \texttt{#true}.

A few examples:

\begin{itemize}
  \item \texttt{(and (> 5 4) (> 7 2))} => \texttt{#true}
  \item \texttt{(or (>= 5 4) (> 7 2))} => \texttt{#true}
  \item \texttt{(and (>= 5 5) (<= 7 2) (> 5 1))} => \texttt{#false}
  \item \texttt{(or (> 4 5) (> 2 7) (< 9 4))} => \texttt{#false}
  \item \texttt{(not (= 5 4))} => \texttt{#true}
  \item \texttt{(and #true (< 3 7) (>= 9 1))} => \texttt{#true}
\end{itemize}

Both \texttt{or} and \texttt{and} require at least two arguments, but may have more.

Write a function that consumes a \texttt{Num}, and returns

- "big" if $80 < x \leq 100$,
- "small" if $0 < x \leq 80$,
- "invalid" otherwise.
**An important subtlety interpreting and and or: short-circuiting**

**and** and **or** are *not* functions. They are **special forms**. Do not evaluate their arguments until necessary.
Informally, evaluate the arguments one by one, and *stop as soon as possible*.

For example:

```scheme
(define (baz x)
  (and (not (= 0 x))
       (> 0 (cos (/ 1 x)))))
```

If I run `(baz 0)`, attempting to evaluate the expression `/ 1 x`, would cause a division by zero error. But when x is zero, the first argument of **and** is **#false**, so the second is not evaluated.
Substitution rules for and

Use the following rules for tracing and:

- \((\text{and} \ #\text{true} \ \text{exp} \ \ldots) \Rightarrow (\text{and} \ \text{exp} \ \ldots)\)
- \((\text{and} \ #\text{false} \ \text{exp} \ \ldots) \Rightarrow #\text{false}\)
- \((\text{and}) \Rightarrow #\text{true}\)

Note: this is not what the stepper does! If in this course you are asked to perform a trace, follow these rules.

Exercise

Perform a trace of

\((\text{and} \ (\ = \ 3 \ 3 \ > \ 7 \ 4 \ < \ 7 \ 4 \ > \ 0 \ (/ \ 3 \ 0)))\)

\[\Rightarrow (\text{and} \ #\text{true} \ (> \ 7 \ 4 \ < \ 7 \ 4 \ > \ 0 \ (/ \ 3 \ 0)))\]
\[\Rightarrow (\text{and} \ (> \ 7 \ 4 \ < \ 7 \ 4 \ > \ 0 \ (/ \ 3 \ 0)))\]
\[\Rightarrow (\text{and} \ #\text{true} \ (> \ 0 \ (/ \ 3 \ 0)))\]
\[\Rightarrow (\text{and} \ (> \ 0 \ (/ \ 3 \ 0)))\]
\[\Rightarrow (\text{and} \ #\text{false} \ (> \ 0 \ (/ \ 3 \ 0)))\]
\[\Rightarrow #\text{false}\]
Use the following rules for tracing or:

- \((\text{or} \ #\text{true} \ \text{exp} \ \ldots) \Rightarrow \ #\text{true}\)
- \((\text{or} \ #\text{false} \ \text{exp} \ \ldots) \Rightarrow (\text{or} \ \text{exp} \ \ldots)\)
- \((\text{or}) \Rightarrow \ #\text{false}\)

Note: this is not what the stepper does! If in this course you are asked to perform a trace, follow these rules.

**Exercise**

Perform a trace of

\((\text{or} \ (< \ 7 \ 4) \ (= \ 3 \ 3) \ (> \ 7 \ 4) \ (> \ 0 \ (/ \ 3 \ 0)))\)

\(\Rightarrow (\text{or} \ #\text{false} \ (= \ 3 \ 3) \ (> \ 7 \ 4) \ (> \ 0 \ (/ \ 3 \ 0)))\)

\(\Rightarrow (\text{or} \ (= \ 3 \ 3) \ (> \ 7 \ 4) \ (> \ 0 \ (/ \ 3 \ 0)))\)

\(\Rightarrow (\text{or} \ #\text{true} \ (> \ 7 \ 4) \ (> \ 0 \ (/ \ 3 \ 0)))\)

\(\Rightarrow \ #\text{true}\)
A museum offers free admission for people who arrive after 5 pm. Otherwise, the cost of admission is based on a person’s age: age 10 and under are charged $5 and everyone else is charged $10.

Exercise

Complete the function \(\text{admission after5? age}\) that returns the admission cost.

\[
\text{admission: Bool Nat -> Num}
\]
Sometimes it is desirable to flatten the conditionals.

```plaintext
;; admission: Bool Nat
-> Nat
(define (admission after5? age)
 (cond
  [after5? 0]
  [else
    (cond
      [(<= age 10) 5]
      [else 10]
    )
  ]
)

(define (admission after5? age)
 (cond
  [after5? 0]
  [(and
    (not after5?)
    [(<= age 10) 5]
    [else 10]
  )]
)

(define (admission after5? age)
 (cond
  [after5? 0]
  [(<= age 10) 5]
  [else 10]
))
```
Uses of `cond`

Conditionals can be used like any other expression:

```scheme
(define (add-1-if-even n)
  (+ n
    (cond
      [(even? n) 1]
      [else 0])))

(or (= x 0)
  (cond
    [(positive? x) (> x 100)]
    [else (< x -100)]))
```
The character datatype Char

There is another datatype, Char, which represents a single character. Some Char values include \a, \4, \space, !

Check equality using char=? or equal?

(char=? \e \e) => #true
(char=? \# \&) => #false
(equal? \X "X") => #false
(equal? \7 7) => #false

Char values can also be compared:

(char<? \e \q) => #true
(char<? \e \Q) => #false

...and ignoring uppercase/lowercase:

(char-ci=? \t \T) => #true
(char-ci<? \e \Q) => #true

Documentation for Char is with that for Str, on the course website.
Strings as lists of characters

In several ways, a Str resembles a list.

Both have a length, and there are ways to get values from both:

- \( \text{(length (list 6 42 7))} => 3 \)
- \( \text{(second (list 6 42 7))} => 42 \)
- \( \text{(string-length "hello world!")} \) => 12
- \( \text{(substring "hello world!" 1 2)} \) => "e"

We can convert a Str to a \((\text{listof Char})\) and back, using \text{string->list} and \text{list->string}.

- \( \text{(string->list "hi there")} \) => \((\text{list #\h #\i #\s\pace #\t #\h #\e #\r #\e})\)
- \( \text{(list->string (list #\h #\i #\s\pace #\t #\h #\e #\r #\e))} \) => "hi there"

Exercise

Write a function \text{drop-e} that converts a Str to a \((\text{listof Char})\), replaces each \#\e with a \#\*, and converts it back to a Str.

\( \text{(drop-e "hello world, how are you?<")} \) => "h*ll* world, how ar* you?"
Black-box and white-box testing

“In science, computing, and engineering, a black box is a device…which can be viewed in terms of its inputs and outputs, without any knowledge of its internal workings.” (Wikipedia)

Black-box testing refers to testing without reference to how the program works. Black-box tests should be written before you write your code. Your examples are black-box tests.

“A white box is a subsystem whose internals can be viewed but usually not altered.” (Wikipedia)

White-box testing should exercise every line of code. Design a test to check both sides of every question in every cond. These tests are designed after you write your code, by looking at how the code works.
You can use the built-in *type predicates* to tell what type a value is:

- `(number? 42) => #true`
- `(char? 42) => #false`
- `(string? 42) => #false`
- `(boolean? 42) => #false`

Using this we can write a function that can consume values of different types. For example:

```scheme
;;; (any->string x) return a string representing x.
;;; any->string: (anyof Str Num Bool) -> Str
;;; Examples:
(check-expect (any->string "foo") "foo")
(check-expect (any->string 42) "42")
(check-expect (any->string #true) "#true")

(define (any->string x)
  (cond [(string? x) x]
        [(boolean? x)
          (cond [x "#true"
                 [else "#false"]])
        [(number? x) (number->string x)]))
```
Take a close look at the contact for this function:

`; (any->string x) return a string representing x.`
`; any->string: (anyof Str Num Bool) -> Str`

Where you see (anyof ...), that represents a single parameter, that can have any of the type in the brackets.

The ability to have parameters with different types is called *dynamic typing*. Some languages instead have *static typing*, where each parameter can have only one type.

Static and dynamic typing have their advantages and disadvantages. But if we are using dynamic typing, it is important to use the contract to keep track of what the types are!
While Racket does not enforce contracts, we will always assume that contracts are followed.

Never call a function with arguments that violate the contract and requirements. If you desire to use one of your own helper functions in a way that violates its contract, that likely means you should modify its contract!

If necessary, you may use anyof in your contract. But first consider if there is a way to accomplish your goal using a simpler contract. It is usually bad style to write functions that can consume multiple types (and a bad habit to have if someday you learn a statically-typed language).
Module summary

- Become comfortable using \texttt{cond} expressions, \texttt{and}, \texttt{or}, and \texttt{not}.
- Get used to combining these statements with the rest of our tools.
- Test these expressions, and know what black-box and white-box testing are.
- Make sure you understand short-circuiting in \texttt{and} and \texttt{or}.
- Become skillful at tracing code which includes \texttt{cond}, \texttt{and}, and \texttt{or}.
- Be able to convert between a \texttt{Str} and a \texttt{(listof Char)}.

Before we begin the next module, please

- Read the Wikipedia entry on \textit{Filter (higher order function)}.
- Read \textit{How to Design Programs}, Section 9.