# Chapter 13

# Booleans

## 13.1 A new data type

We've seen several data types so far: images, strings, numbers, and sub-types of number: integers, fractions, inexact numbers, complex numbers. Each data type is suitable for answering a different kind of question:

- images answer the question "what does ... look like?";
- strings answer questions like "what is your name?" or "what is the text of the Gettysburg Address?";
- integers answer questions like "how many ...?";
- fractions and inexact numbers answer questions like "how much ...?"

Now consider "true/false" or "yes/no" questions: "is Joe older than Chris?", "is your name Philip?", and so on. For each of these questions, there are only two possible answers: yes and no. None of the data types above seems quite right for the purpose. So Racket, like most programming languages, has a data type named "boolean", which has exactly two values, written `true` and `false`. You can type either of these into the Interactions pane, hit ENTER, and you'll get back the same value, just as if you had typed a number or a quoted string. Note that `true` is different from `"true"`: the former is a boolean, and the latter is a string.

SIDEBAR:

The word "boolean" is named after the 19th-century mathematician George Boole, who suggested that logical questions of truth and falsity could be addressed by the techniques of algebra, using "numbers" that were restricted to the values 0 and 1 (representing false and true, respectively).

## 13.2 Comparing strings

Racket has a number of built-in functions that produce Booleans. The first one we'll look at is

```
string=?  :  string string -> boolean
```

For example,

```
(check-expect (string=?  "hello" "goodbye") false)
(check-expect (string=?  "hello" "hello") true)
(check-expect (string=?  "hello" "Hello") false)
(check-expect (string=?  "hello" "hel lo") false)
(check-expect (string=?  "hello" (string-append "hel" "lo")) true)
```

Note that the two strings have to be *exactly* the same, right down to capitalization, spacing, and punctuation. Also note that, by convention, most functions that return Booleans (like `string=?`) have names ending in a question mark. (Racket doesn't *force* you to do this, but it's a good habit to follow, in order to get along with other Racket programmers.)

So now you know how to tell whether two strings are exactly the same. You can also test how two strings relate in alphabetical order:

```
; string<?  :  string string -> Boolean
; string<=?  :  string string -> Boolean
; string>?  :  string string -> Boolean
; string>=?  :  string string -> Boolean
```

**Practice Exercise 13.2.1** *Make up some examples involving `string<?`, `string<=?`, `string>?`, and `string>=?`, and see whether they produce the answer you expect. Try comparing a capitalized word like `"DOG"` with an uncapitalized one like `"cat"`. Try comparing either of those with a string made up of punctuation marks, like `"!#., &*"`, or a string made up of digits, like `"372.4"`.*

SIDEBAR:

Sometimes it's convenient to treat upper-case, lower-case, and mixed-case words all the same. Racket also provides "case-independent" versions of each of these functions:

```
; string-ci=?  :  string string -> Boolean
; string-ci<?  :  string string -> Boolean
; string-ci<=?  :  string string -> Boolean
; string-ci>?  :  string string -> Boolean
; string-ci>=?  :  string string -> Boolean
```

To see how these are used in practice, let's write some simple functions.

**Worked Exercise 13.2.2** ***Develop a function*** *`is-basketball?` that takes in a string representing the name of a game, and returns a Boolean indicating whether the game was `"basketball"`.*

**Solution:** The **contract** is clearly

```
; is-basketball?  :  string(game) -> boolean
```

For **examples**, we note that there are two possible answers: `true` and `false`. To test the program adequately, let's make sure we have an example that produces `true`, and one that produces `false`.

```
"Examples of is-basketball?:"
(check-expect (is-basketball?  "basketball") true)
(check-expect (is-basketball?  "cricket") false)
```

Next, we need to write a **skeleton**. The important decisions have already been made in the contract, so

```
(define (is-basketball?  game)
  ...)
```

For the **inventory**, we obviously need the parameter `game`. In addition, since the problem specifically mentions the string `"basketball"`, that string is likely to appear in the function:

```
(define (is-basketball?  game)
  ; game             a string
  ; "basketball"     another string
  ...)
```

Now, to fill in the **function body**, we notice that we have two strings (`game` and `"basketball"`) and we want a Boolean; conveniently enough, there's a built-in function `string=?` that takes in two strings and returns a Boolean. So let's use it:

```
(define (is-basketball?  game)
  ; game             a string
  ; "basketball"     another string
  (string=?  game "basketball")
  )
```

Now we can **test** the function on our two examples, and it should work. ∎

**Exercise 13.2.3** ***Develop a function*** `is-nintendo?` *that takes in a string and tells whether it was* `"nintendo"`*.*

**Exercise 13.2.4** ***Develop a function*** `empty-string?` *that takes in a string and tells whether it was* `""`*.*

**Exercise 13.2.5** ***Develop a function*** `in-first-half?` *that takes in a (lower-case) string and tells whether it's in the first half of the alphabet (*i.e. *it comes before* `"n"` *in alphabetical order)*

**Hint:** You'll need at least two examples: one in the first half of the alphabet, and one in the second half. It's also a good idea to have an example that's `"n"` itself; this is called a *borderline example*. What do you think is the "right answer" for this example?

## 13.3 Comparing numbers

Just as `string=?`, `string<?`, *etc.* allow us to compare strings, there are built-in functions that allow us to compare numbers. Here are the most common ones:

```
; = :  number number -> Boolean
; Tells whether the two numbers have the exact same value.

; < :  number number -> Boolean
; Tells whether the first number is less than the second.

; > :  number number -> Boolean
; Tells whether the first number is more than the second.

; <= :  number number -> Boolean
; Tells whether the first number is at most the second.

; >= :  number number -> Boolean
; Tells whether the first number is at least the second.
```

Note that these functions, despite returning Booleans, do *not* have names ending in a question-mark; their traditional mathematical names were so well-established that the designers of Racket decided to keep those names at the expense of the Racket convention.

To get some practice with these functions, let's start by trying some expressions:

```
(check-expect (= 3 4) false)
(check-expect (< 3 4) true)
(check-expect (> 3 4) false)
(check-expect (<= 3 4) true)
(check-expect (>= 3 4) false)
(define age 21)
(check-expect (> age 12) true)
(check-expect (< age 18) false)
(check-expect (= (+ 3 4) 5) false)
(check-expect (= (+ 3 4) 7) true)
```

Feel free to make up and try some more examples of your own.

Now let's try writing some simple functions that use the built-in number comparison operators.

**Worked Exercise 13.3.1** ***Develop a function*** `may-drive?` *that takes in the age of a person and returns whether that person is old enough to drive a car legally (which in most of the U.S. means "at least 16 years old").*

**Solution:** For the **contract**, we note that the function "takes in the age of a person", which sounds like it should be a number, "and returns whether . . . " The word "whether" in a problem statement almost always means a Boolean. So the contract should be

```
; may-drive?  :  number(age) -> Boolean
```

For the **examples**, we note first that there are two possible answers — `true` and `false` — and therefore there must be at least two examples. Furthermore, there's a *borderline* situation between *sub-ranges* of inputs (as there was with `in-first-half?` above), so we should also test the borderline case.

```
"Examples of may-drive?:"
(check-expect (may-drive?  15) false)
(check-expect (may-drive?  23) true)
(check-expect (may-drive?  16) true) ; borderline case
```

The **skeleton** is straightforward:

```
(define (may-drive?  age)
  ...)
```

The **inventory** lists the parameter `age` and the literal 16:

```
(define (may-drive?  age)
  ; age      a number
  ; 16       a fixed number we're likely to need
  ...)
```

Now we can fill in the **body**. We have two numbers, and we need a Boolean; conveniently enough, we know of several built-in functions (=, <, >, <=, >=) that take in two numbers and return a Boolean. Let's try >.

```
(define (may-drive?  age)
  ; age      a number
  ; 16       a fixed number we're likely to need
  (> age 16)
  )
```

That wasn't too bad. Now we **test** the function ... and we see that it gets one of the answers *wrong*! In particular, it gets the "clear-cut" cases right, but it gets the "borderline" example wrong. This is a common pattern; watch for it! It usually means we've got the direction of the comparison right, but either we should have added an = sign and didn't, or we shouldn't have but did. In this case, it means we should have used >= rather than >.

```
(define (may-drive?  age)
  ; age      a number
  ; 16       a fixed number we're likely to need
  ( >= age 16)
  )
```

Now we test this again; it should work correctly for all cases. ∎

**Practice Exercise 13.3.2** *Suppose we had mistakenly typed the < operator in the function body instead of > or >=.* ***What pattern*** *of right and wrong answers would we have gotten?* ***Try it*** *and see whether your prediction was right.*

*Likewise,* ***what pattern*** *of right and wrong answers would we have gotten if we had typed <= instead of >, >=, or <?* ***Try it*** *and see whether your prediction was right.*

*Now, suppose we had chosen the >= operator, but had its arguments in the opposite order:* `(>= 16 age)`. ***What pattern*** *of right and wrong answers would we have gotten?* ***Try it*** *and see whether your prediction was right.*

**Watch for these patterns** whenever you're debugging a program that involves subranges of numbers or strings.

**Exercise 13.3.3** ***Develop a function*** `may-drink?` *that takes in a person's age and returns whether the person is old enough to drink alcohol legally. (In most of the U.S., this means "at least 21 years old".)*

**Exercise 13.3.4** ***Develop a function*** `under-a-dollar?` *that takes in the price of an item in dollars (*e.g. `1.49` *or* `.98`*) and tells whether it's less than* `1.00` *.*

**Exercise 13.3.5** ***Develop a function*** `is-17?` *that takes in a number and tells whether it's exactly 17.*

## 13.4 Designing functions involving booleans

In the above examples, we've used the fact that the function returns a boolean to help us choose test cases: you need at least one test case for which the right answer is `true`, and at least one test case for which the right answer is `false`, or you haven't tested the function adequately. Furthermore, if the input consists of *sub-ranges* with *borderlines* between them, you also need to test right at the borderline. We'll incorporate this idea into the design recipe as follows:

1. Write a contract (and perhaps a purpose statement).
2. *Analyze input and output data types.*
3. Write examples of how to use the function, with correct answers. *If an input or output data type consists of two or more cases, be sure there's at least one example for each case. If an input type involves sub-ranges, be sure there's an example at each borderline.*
4. Write a function skeleton, specifying parameter names.
5. Write an inventory of available expressions, including parameter names and obviously relevant literals, along with their data types (and, if necessary, their values for a specific example).
6. Fill in the function body.
7. Test the function.

We've added one new step: *Analyze input and output data types.* When we were simply writing functions that took in or returned images or numbers, there wasn't much "analysis" to be done. But a function that returns a Boolean can be thought of as distinguishing two sub-categories of input: those inputs for which the right answer is `true`, and those for which it's `false`. And in many cases there are even more sub-categories, as we'll see in Section 13.7. Identifying these sub-categories (and any borderlines between them) early helps you choose good test cases.

**Exercise 13.4.1** ***Develop a function*** `much-older?` *that takes in two people's ages and tells whether the first is "much older" (which we'll define as "at least ten years older") than the second.*

**Exercise 13.4.2** ***Develop a function*** `within-distance?` *that takes in three numbers: x, y, and distance. The function should return whether or not the point* $(x, y)$ *is at most the specified distance from the point* $(0, 0)$*. The formula for the distance of a point to* $(0, 0)$ *is* $\sqrt{x^2 + y^2}$*.*

**Hint:** You may want to write an auxiliary function to compute the distance.

## 13.5 Comparing images

Just as we can compare strings or numbers to see whether they're the same, we can also compare two *images* to see whether they're the same:

```
; image=?  :  image image -> Boolean
```

But images don't have an "order", so there are no image functions analogous to `string<?`, `string>?`, *etc.*.

**Exercise 13.5.1** ***Develop a function*** *`is-green-triangle?` that takes in an image and tells whether it is exactly `(triangle 10 "solid" "green")`.*

## 13.6 Testing types

As we've seen, Racket has several built-in data types: numbers, strings, images, booleans, *etc.* It also has built-in functions to *tell whether* something is of a particular type:

```
; number?  :  anything -> boolean
; tells whether its argument is a number.
; image?  :  anything -> boolean
; tells whether its argument is an image.
; string?  :  anything -> boolean
; tells whether its argument is a string.
; boolean?  :  anything -> boolean
; tells whether its argument is a boolean.
; integer?  :  anything -> boolean
; tells whether its argument is an integer.
...
```

You've already seen the `image?`, `number?`, and `string?` functions: we used them in the `check-with` clause of an animation to specify what type the model is. In fact, `check-with` can work on *any* function that has contract `anything -> boolean`: if you wanted to write an animation with a Boolean model, you could say `(check-with boolean?)`. We'll see more applications of this in Chapter 21.

**Practice Exercise 13.6.1** ***Try*** *the following expressions in the interactions pane. For each one,* ***predict*** *what you think it will return, then see whether you were right. If not,* ***experiment*** *some more until you understand what the function does.* ***Make up*** *some similar examples of your own and try them similarly.*

```
(number?  3)
(number?  5/3)
(number?  "3")
(number?  "three")
(number?  true)
(integer?  3)
(integer?  5/3)
(integer?  "3")
(integer?  "three")
(string?  3)
(string?  "3")
(string?  "three")
(image?  3)
(image?  (circle 5 "solid" "green"))
(number?  (+ 3 4))
(number?  (> 3 4))
(boolean?  (> 3 4))
(boolean?  3)
(boolean?  false)
(boolean?  true)
```

SIDEBAR:

Mathematicians use the word *predicate* to mean a function that returns a Boolean, so sometimes you'll hear Racket programmers referring to "type predicates". A type predicate is simply any one of these functions that tell whether something is of a particular type. Another name for *type predicate* is *discriminator*.

**Common beginner mistake**
Students are often confused by the difference between `string?` and `string=?`, between `image?` and `image=?`, *etc.* All of these functions return Booleans, but they do different things.

The `string=?` function takes *two* arguments (which *must be strings*), and tells whether they're the *same* string. The `string?` function takes *one* argument (which may be of *any type*), and tells whether it is a string at all.

Likewise, `image=?` tells whether two images are the same, while `image?` tells whether something is an image at all.

And `boolean=?` (which you'll almost never need!) tells whether two booleans are the same, while `boolean?` tells whether something is a boolean at all.

Finally, `=` tells whether two numbers are the same, while `number?` tells whether something is a number at all.

## 13.7 Boolean operators

Advertisers like to divide the world of consumers into age categories, and one of their favorites is the "18-to-25 demographic": these people are typically living on their own for the first time, spending significant amounts of their own money for the first time, and forming their own spending habits. If an advertiser can get a 19-year-old in the habit of

buying a particular brand of shampoo or canned soup, it may pay off in decades of sales. For this reason, advertisers concentrate their work in places that 18-to-25-year-olds will see it.

**Worked Exercise 13.7.1** ***Develop a function*** *that takes in somebody's age, and decides whether the person is in the 18-to-25 demographic.*

**Solution:** The contract is easy:

```
; 18-to-25?  :  number (age) -> Boolean
```

Analyzing the data types tells us nothing new about the output type. The input is more interesting: it could be thought of as "18-to-25" and "everything else", but it seems more natural to break it down into *three* categories: under-18, 18-to-25, and over-25.

We'll need at least one example in each of these three categories, plus borderline examples for *both* borderlines — 18 and 25.

```
"Examples of 18-to-25?:"
(check-expect (18-to-25?  15) false)
(check-expect (18-to-25?  18) true)
(check-expect (18-to-25?  20) true)
(check-expect (18-to-25?  25) true)
(check-expect (18-to-25?  27) false)
```

We chose the examples based on the data analysis, and figured out the "right answers" by applying common sense to the problem: if the advertisers want "18-to-25-year-olds", they probably mean to include both 18-year-olds and 25-year-olds (even though somebody may be described as "25 years old" right up to the day before turning 26).

The skeleton is straightforward:

```
(define (18-to-25?  age)
  ...)
```

The inventory throws in a parameter and two literals:

```
(define (18-to-25?  age)
  ; age      a number
  ; 18       a fixed number we'll need
  ; 25       another fixed number we'll need
  ...)
```

In fact, we can predict some things we'll want to do with the numbers: we'll probably want to check whether `age` is at least 18, and whether `age` is at most 25:

```
(define (18-to-25?  age)
  ; age            a number
  ; 18             a fixed number we'll need
  ; 25             another fixed number we'll need
  ; (>= age 18)    a Boolean
  ; (<= age 25)    a Boolean
  )
```

But now we face a problem: we have two Booleans, both of which represent *part* of the right answer, but neither of which is the *whole* right answer. We need to *combine* the two, using a *Boolean operator*.

A Boolean operator is a function that takes in one or more Booleans and returns a Boolean. Racket has three common Boolean operators: `and`, `or`, and `not`. The `and` and `or` operators each take in two or more Booleans; the `not` operator takes in exactly one.

Let's try some examples of these in the Interactions pane:

```
(check-expect (or false false) false)
(check-expect (or false true) true)
(check-expect (or true false) true)
(check-expect (or true true) true)
(check-expect (or (= 3 5) (= (+ 3 4) 7)) true)
(check-expect (or (< 3 5) (= (+ 3 4) 5)) true)
(check-expect (or (> 3 5) (= (+ 3 4) 5)) false)
(check-expect (and false false) false)
(check-expect (and false true) false)
(check-expect (and true false) false)
(check-expect (and true true) true)
(check-expect (and (= 3 5) (= (+ 3 4) 7)) false)
(check-expect (and (< 3 5) (= (+ 3 4) 7)) true)
(check-expect (not (= 3 5)) true)
(check-expect (not (< 3 5)) false)
(check-expect (or false true false) true)
(check-expect (or false false false) false)
(check-expect (and false true false) false)
(check-expect (and true true true) true)
```

Now back to our problem. We have two Boolean expressions: one represents whether `age` is at least 18, and the other represents whether `age` is at most 25. To find out whether *both* of those things are true simultaneously, we need to combine the expressions using `and`:

```
(define (18-to-25?  age)
  ; age            a number
  ; 18             a fixed number we'll need
  ; 25             another fixed number we'll need
  ; (>= age 18)    a Boolean
  ; (<= age 25)    a Boolean
  (and (>= age 18)
       (<= age 25))
  )
```

Now we can test the function, and it should work correctly on all five examples. ∎

**Practice Exercise 13.7.2** ***What would have happened*** *if we had used* `or` *instead of* `and` *in defining the* `18-to-25?` *function?* ***Predict*** *the pattern of right and wrong answers, then* ***change*** *the function definition and check whether you were right.*

**Exercise 13.7.3** ***Develop a function*** `teenage?` *that takes in a person's age and returns whether the person is at least 13 but younger than 20.*

**Exercise 13.7.4** ***Develop a function*** `negative-or-over-100?` *that takes in a number and returns whether it is either negative (*i.e. *less than zero) or over 100.*

**Exercise 13.7.5** ***Develop a function*** `may-drive-but-not-drink?` *that takes in a person's age and tells whether the person is old enough to have a driver's license (in most of the U.S.) but* not *old enough to drink alcohol (in most of the U.S.).*

**Hint:** Re-use previously-written functions!

**Exercise 13.7.6** *The game of "craps" involves rolling a pair of dice, and (in a simplified version of the game) if the result is 7 or 11, you win.* ***Develop a function*** *named* `win-craps?` *that takes in a number and tells whether it's either a 7 or an 11.*

**Exercise 13.7.7** ***Develop a function*** *named* `play-craps` *that takes a dummy argument, rolls two dice, adds them up, and returns* `true` *or* `false` *depending on whether you won the roll.*

**Hint:** Re-use previously defined functions!

SIDEBAR:

How would you test a function like `play-craps`? It ignores its argument, so all you can see directly is that sometimes it returns `true` and sometimes `false`, regardless of the argument. *How much of the time* should it return `true`? *How many runs* would you need to make in order to tell whether it was behaving the way it should?

**Exercise 13.7.8** ***Develop a function*** `not-13?` *that takes a number and tells whether it's* not *exactly 13.*

**Exercise 13.7.9** ***Develop a function*** `not-single-letter?` *that takes a string and tells whether its length is anything other than 1.*

**Exercise 13.7.10** ***Develop a function*** `over-65-or-teenage?` *that takes in a person's age and tells whether the person is either over 65 or in his/her teens.*

**Exercise 13.7.11** ***Develop a function*** `lose-craps?` *that takes in a number and tells whether it is* not *either 7 or 11. That is, the result should be* `false` *for 7 and 11, and* `true` *for everything else.*

**Exercise 13.7.12** ***Develop a function*** `is-not-red-square?` *that takes in an image and tells whether it is anything other than a solid red square.*

**Hint:** Use `image-width` to find out how wide the image is.

**Exercise 13.7.13** ***Develop a function*** `any-two-same-pics?` *that takes in three images and tells whether any two (or more) of them are exactly the same.*

**Hint:** There are at least three different ways the answer could be `true`; test them all, as well as at least one case in which the answer should be `false`.

## 13.8 Short-circuit evaluation

Technically, `and` and `or` aren't really functions in Racket but rather something called *special forms*; `define` is also a *special form*. The main difference, for now, is that every argument to a function has to have a value, or the function call doesn't make sense. Obviously, `define` can't work that way, because its whole purpose is to define a variable or function that doesn't already have a meaning. The `and` and `or` operators *could* have been regular functions, but the designers of Racket chose to make them special forms in order to get something called *short-circuit evaluation*.

The idea is, if the first argument of an `or` is `true`, you don't really care what the rest of the arguments are; you already know the answer. Suppose you had a variable named $x$ defined, and think about an expression like

```
(or (= x 0) (> (/ 7 x) 2))
```

If `or` were an ordinary function, DrRacket would compute the Boolean values of `(= x 0)` and `(> (/ 7 x) 2)`, and then apply `or` to the results. If $x = 0$, the sub-expression `(/ 7 x)` would crash (because you can't divide by zero) before `or` ever got a chance to do its job. Instead, DrRacket computes the Boolean value of `(= x 0)`, and if it's `true`, `or` returns `true` immediately without even looking at its second argument (which doesn't have a value). Only if $x \neq 0$ does DrRacket try to compute `(> (/ 7 x) 2)`, and if $x \neq 0$, this expression is guaranteed to have a value so everything's OK.

Similarly, if the first argument of an `and` is `false`, you already know the answer and don't need to even look at the rest of the arguments. Try typing each of the two expressions

```
(and (> 2 3) y)
(and y (> 2 3))
```

in the Interactions pane of DrRacket. The first should return `false`; the second should complain that it's never heard of the variable $y$. (If you try these two expressions in the Definitions pane and hit "Step", "Check Syntax", or "Run", *both* will produce error messages, because the Definitions pane checks that all variable names are defined before they are used.)

For most purposes, you can ignore short-circuit evaluation. But on rare occasions, it makes a difference to your programming: you can make a program run faster, or even run without crashing, by putting the arguments of `or` or `and` in a different order.

## 13.9 Review of important words and concepts

Racket has a *Boolean* data type with two values — `true` and `false` — which is used for yes/no questions. A number of built-in functions allow you to compare strings for equality or order, compare numbers for equality or order, compare images for equality, and combine two or more Booleans into one.

We've added a new step to the design recipe for functions: *analyze the input and output data types*. For now, this means identifying interesting "sub-categories" of input and output. Once you've done this, it helps you in choosing good test cases: make sure to have at least one test case for each sub-category. In addition, if the sub-categories are *ranges* with *borderlines* in between them, make sure to test the function at the borderlines.

If the function doesn't pass all its tests, pay attention to *patterns of right and wrong answers*: was it *always* wrong, or only sometimes? Did it work on the "clear-cut" cases but not the borderlines? The borderlines but not the "clear-cut" cases? These patterns give you valuable clues in figuring out what's wrong with the program.

A *predicate* is any function that returns a Boolean. In Racket, most such functions (by convention) have names ending in a question mark. There are built-in *type predicates* — functions that take in *any type* of argument, and tell whether or not it is, say, a number. They tend to have obvious names: `number?`, `image?`, `string?`, `boolean?`, `integer?`, *etc.*

You can handle much more complicated and sophisticated categories of input by combining Boolean-valued expressions using the Boolean operators `and`, `or`, and `not`.

## 13.10 Reference: Functions involving Booleans

Here are the new built-in functions (and special forms) we've discussed in this chapter:

- `string=?`
- `string<?`
- `string<=?`
- `string>?`
- `string>=?`
- `string-ci=?`
- `string-ci<?`
- `string-ci<=?`
- `string-ci>?`
- `string-ci>=?`
- `=`
- `<`
- `>`
- `<=`
- `>=`
- `image=?`
- `image?`
- `number?`
- `string?`
- `boolean?`
- `boolean=?`
- `and`
- `or`
- `not`