Introduction to Data Structures

Students encounter Data Structures in the context of using coordinates to model 2D animation. They explore constructors and fields, creating a "digital bakery" using structures to model cakes.

Product Outcomes	• Students identify real-world behaviors that require data structures • Students make use of a complex data structure: Cake • Students define variables bound to Cakes • Students will generalize their understanding of function constructors and accessors • Students write code that extracts each field from those Cakes • Students will write functions that access fields of a CakeType
Materials	• PDF of all Handouts and Page • Package Delivery Starter File • Bakery Starter File • Slides are not yet available for this lesson • Printable Lesson Plan (a PDF of this web page)
Prerequisites	• Simple Data Types • Contracts • Simple Inequalities • Piecewise Functions and Conditionals • Compound Inequalities: Solutions & Non-Solutions
Supplemental Materials	• Additional Printable Pages for Scaffolding and Practice

Glossary

calling :: using a function by giving it inputs
constructor :: a function that creates instances of a data structure
contract :: a statement of the name, domain, and range of a function
data block :: code that lists the name, constructor(s), and field(s) of a data structure
data structure :: a 'container' data type, which has fields that can hold other data (e.g. - a 'coordinate'

is a data structure holding number fields x and y)

domain :: the type or set of inputs a function expects, i.e., the independent variable(s) that govern the output of the function

dot accessor :: a way to extract the values of fields an instance

field :: a part of a data structure that has a name and holds a single value of a specified data type

instance :: a specific example of a data structure, with specific values for each field (e.g. - (4,5) is an instance of an (x,y) coordinate

name :: how we refer to a function or value defined in a language (examples: +, *, star, circle)

purpose statement :: a concise, detailed description of what a function does with its inputs

range :: the type or set of outputs that a function produces, i.e., the dependent variable(s)

variable :: a name or symbol that stands for some value or expression, often a value or expression that changes

Review *15 minutes*

Launch

In the previous unit, you reviewed almost everything from Bootstrap:Algebra including data types, Contracts, and the Design Recipe. In this unit you will go above and beyond all that, and learn an entirely new construct that will be the basis for everything you'll do in Bootstrap:Reactive.

Ask a few introductory review questions to test students' understanding:

- What are the three parts of a Contract?
- What is the Pyret code to draw a solid, green triangle of size 22?
- Why is it important to write at least 2 examples before defining a function?

Investigate

To make sure the material from the previous unit is fresh in your mind, tackle the following activity:

Turn to Word Problem: double-radius in your workbook. Write a function called `double-radius`, which takes in a radius and a color. It produces an outlined circle of whatever color was passed in, with radius twice as big as the input.

If walking through this example as a class, use a projector so kids can see the function being written on the computer.

Remember how to use the design recipe to work through word problems?

Step 1: Contract and Purpose Statement

- What is the ***Name*** of this function? How do you know?
- How many inputs does it have in its ***Domain***?
- What kind of data is the ***Domain***?
- What is the ***Range*** of this function?
- What does this function do? Write a ***Purpose Statement*** describing what the function does in plain English.

```
# double-radius :: Number, String -> Image
# Makes an outlined circle that has twice the given radius.
```

Review the purpose of Contracts: once we know the Name, Domain, and Range of a function, it's easy to write examples using those data types.

Step 2: Examples

Using only the Contract and Purpose Statement, see if you can answer the following questions:

- Every example begins with the name of the function. Where could you find the name of the function?
- Every example has to include sample inputs. Where could you find out how many inputs this function needs, and what type(s) they are?
- Every example has to include an expression for what the function should do when given an input. Where could you look to find out what this function does?
- Write two examples on your paper, then circle and label what is changing between them. When labeling, think about what the changing things represent.

Don't forget to include the lines `examples:` and `end`! Your examples should look similar to:

```
examples:
  double-radius(50, "pink") is circle(50 * 2, "outline", "pink")
  double-radius(918, "orange") is circle(918 * 2, "outline", "orange")
end
```

Each one of these answers can be found in the Contract or Purpose Statement. Suggestion: Write these steps on the board, and draw arrows between them to highlight the process. The goal here is to get students into the habit of asking themselves these questions each time they write examples, and then using their own work from the previous step to find the answers.

Step 3: Definition

Once you know what is changing between our two examples, you can define the function easily. The things that were circled and labeled in the examples will be replaced with ***variables*** in the function definition.

Underneath your examples, copy everything that **doesn't** change, and replace the changing things with the variable names you used. (Don't forget to add the `fun` and `end` keywords, as well as the single colon (`:`) after the function header!)

```
# double-radius :: Number, String -> Image
# Makes an outlined circle that's twice the radius.
fun double-radius(radius, color):
  circle(radius * 2, "outline", color)
end
```

For more practice, turn to Word Problem: double-width in your workbook and complete the Design Recipe for the `double-width` function.

Check students understanding: Why do we use variables in place of specific values? Why is it important to have descriptive variable names, as opposed to n or x? Remind students about nested functions: A function whose range is a number can be used inside of a function requiring a number in its domain, as in circle(2 * 25, "outline", "red").

Introducing Structures *30 minutes*

Open the Package Delivery Starter File file on your computer and press "Run". What happens?

The drone tries to deliver a package directly to a house, but the box falls straight down, outside of the delivery zone. We want the package to fall *diagonally*, and land right in front of the house. Let's take a look at the code to see why it falls into the road instead. There are a few new concepts in this file, but first, let's focus on what you already know.

Look at the function defined here called `next-position`.

- What is this function's Domain? Its Range?
- What does `next-position` do with its inputs?

This function takes in two numbers, representing the x- and y-coordinate of the box, but it only produces a new y-coordinate (after subtracting 5). If only the y-coordinate is changing, the box will always fall straight down. To reach the house, it will have to fall diagonally.

How should the box's x-coordinate change if it moves diagonally to the right (toward the house)? How should its y-coordinate change?

Functions can return only one thing at a time, but we want to return a new x- **and** a y-coordinate in order to make the box fall diagonally. Thankfully, we have a way to combine multiple things within one container, called a ***Data Structure***. For this project, we've created a structure for you to use called `DeliveryState`, which contains two Numbers. These represent an x and a y-coordinate.

Look at line 5, where we've defined `DeliveryState`. We'll go through the new syntax for defining a data structure, because very soon you'll be defining brand new structures of your own!

```
# The DeliveryState is two numbers: an x-coordinate and a y-coordinate
data DeliveryState:
   | delivery(
      x :: Number,
      y :: Number)
end
```

- On the first line, we've written a comment that describes the stucture. We're calling it `DeliveryState`, and it contains Numbers for the x- and y-coordinate.

- You're already familiar with built-in data types like `Number`, `String`, `Image` and `Boolean`. On the next line, the `data` keyword allows us to create brand new data types of our own! Here, we are making a data type called `DeliveryState`. We choose this name, because it represents the current state — or position — of the package being delivered. Pyret lets us write any name after `data`, but it's good habit to choose a meaningful name and capitalize it.
- The next line begins with the `|` symbol, sometimes called a "bar" or "pipe", followed by the name of the ***constructor*** function for this structure: `delivery`. This is similar to what you've seen before: to create an Image, we call the function that creates it: `rectangle`, `triangle`, `square`, etc. To create a `DeliveryState`, we can use the `delivery` ***constructor*** function with its inputs (x and y).

This ***data block*** tells us that we're defining a new data type called `DeliveryState`, whose constructor function `delivery` takes in two Numbers: x and y. Once we've listed each input and its data type, we finish defining the structure with the `end` keyword, just like finishing an `example` block.

In the Interactions Area, practice making some `DeliveryStates` using the `delivery()` constructor function. Try making a `DeliveryState` that represents the box's position if it's on the road, another when it's in the air, above the house, and one when it's right in front of the house — a successful delivery!

Students will soon be writing creating new data structures. Cover this new syntax carefully, paying special attention to capitalization (the name of the structure is capitalized (`DeliveryState`), whereas its constructor function (delivery) is lowercase), double colons (::) before data types, and commas between inputs to the constructor function.

Now it's up to us to get this box delivered sucessfully, and make sure it lands at the house.

Turn to Word Problem: next-position in your workbook, read the word problem, and fill in the Contract and Purpose Statement for the function `next-position`.

```
# next-position :: Number, Number -> DeliveryState
# Given 2 numbers, make a DeliveryState by
# adding 5 to x and subtracting 5 from y
```

Point out that we're now using a new data type in a contract: next-position consumes two Numbers, and produces a DeliveryState. Once we've defined a new data structure using the above data block, we can use it just like other data types.

Now for our two examples. Using, or ***calling*** `next-position` with two numbers is easy, but what happens to those numbers? We can't return both at the same time…unless we use a data structure! To do so we'll need to use the constructor function to make a structure from the data we already have.

- According to the definition for `DeliveryState`, what function makes a DeliveryState? What is its contract?
- `# delivery :: Number, Number -> DeliveryState`
- What two things are part of a DeliveryState? Do we have values for those things as part of our first example?
- We don't want our DeliveryState to contain the same x and y values we gave the `next-position` function. How will the values change? (Remember to show your work!)
- Your first example should look something like:

```
examples:
  next-position(30, 250) is delivery(30 + 5, 250 - 5)
end
```

- Once your first example is complete, write one more example with different inputs for the x and y coordinates.

Remind students to show every step of their work in the example step of the design recipe: if the x-coordinate increases by 5 while the y-coordinate decreases by 5, they should show the addition and subtraction within the DeliveryState data structure, instead of just returning the new numbers.

Now that you have two examples, it's time to define the function. You know the drill: circle and label everything that changes between your two examples, copy everything that stays the same, and replace the changing things with the variables you chose.

When you finish, your function definition should look like:

```
fun next-position(x, y):
  delivery(x + 5, y - 5)
end
```

Now, instead of just changing and returning one number (a y-coordinate), we can return **both** the x and y-coordinates of the box within a ***Data Structure***.

Open the Package Delivery code again and replace the original `next-position` function with the one in your workbook to make the box land within the dlivery zone, in front of the house! Don't forget to change the given examples to match your new function definition.

Synthesize

Until now, a function could only return atomic values: single Numbers, Strings, Images, or Booleans. In Bootstrap:Reactive, our functions will still return one value, but that value can be a ***Data Structure***, (or just "structure" for short) containing any number of values. This way we can return both the x- and y-coordinate of a package using a `DeliveryState`. Later on, we'll create new structures to record detail about characters in a game, like their health, position, amount of armor, or inventory.

In Bootstrap:Algebra, students' games were made by keeping track of just a few numbers: the x-positions of the danger and target, and y-position of the player. In Bootstrap:Reactive, students' games will be much more complex, and will require many more values to move characters, test conditions, keep track of the score, etc. Data structures simplify code by organizing multiple values: You couldn't represent every part of a player (position, health, inventory, etc.) with one number or string, but you can refer to all these things collectively with a data structure. This way, we can have one value (a data structure) containing multiple other values that can be accessed individually.

Cakes *30 minutes*

Overview

Students walk through the process of defining a data structure based on a word problem.

Launch

Suppose you own a famous bakery. You bake things like cookies, pastries, and tarts, but you're especially known for your world-famous cakes. What type of thing is a cake? Is it a number? String? Image? Boolean? You couldn't describe all of the important things about a cake with any one of those data types. However, we could say that we care about a couple of details about each cake, each of which can be described with the types we already know.

For each of the following aspects of a cake, think about what data type you might use to represent it:

- The flavor of the cake. That could be "Chocolate", "Strawberry", "Red Velvet", or something else.
- The number of layers
- Whether or not the cake is an ice cream cake.

What data type could we use to represent the entire cake?

Now that we know everything that is part of a cake, we can use a data structure to represent the cake itself. Let's take a look at how this works.

Investigate

Open your workbook to Data Structure: CakeType.

On this page, we will define a data structure for cakes, which we call `CakeType` (since this is now a new data TYPE). At the top of this page we see a comment, stating what things are part of a `CakeType`. Below that is a line that says `data CakeType:`, which begins the definition of a new data structure, called CakeType. On the next line, we define the function that makes a CakeType (`cake`), and how *exactly* to make a CakeType — the names of each thing in a CakeType, and their data types. Each piece of information that makes up a cake (the flavor, etc) is called a ***field***. A field has both a descriptive name (like `flavor`) and a data type.

What name describes the first field in a `CakeType`? What data type can we use to represent it?

Refer students back to their language table, to see what Types are available.

There is a little bit of new syntax involved in defining structures. On the first line on Data Structure: CakeType, we write `flavor :: String`, which tells Pyret that the first element of *any* CakeType will be its flavor, represented by a String. This line shows how to define one field in a data structure.

What name describes the second field in a `CakeType`? What data type can we use to represent it?

On the next line, write `layers :: Number,`, which tells Pyret that the second element of any CakeType will be its number of layers, represented by a Number.

What data structure should we use to represent whether or not the CakeType is an ice cream cake? Use this to define another field.

On your paper, you should have:

```
# a CakeType is a flavor, number of layers, and whether or not it is an
ice cream cake.
data CakeType:
  | cake(
      flavor      :: String,
      layers      :: Number,
      is-iceCream :: Boolean)
end
```

This is the code that defines the `CakeType data` structure. It tells the computer what a `CakeType` is and what goes into it. It also defines its ***constructor*** function, called `cake`. To make a CakeType, you *must* call the constructor function with three things: a `flavor`, which is a String, `layers`, a Number, and `is-iceCream`, which is a Boolean. Remember that order matters! For now, these are the only things that we're going to keep track of in a CakeType, but you can imagine how you might extend it to include other information.

Stress the importance of being able to define your own data types to students: no longer are they bound by the single values of numbers, strings, or Booleans! Pyret allows you to define brand new Data Structures, containing any combination of values.

Open the Bakery Starter File and look at lines 3–8. Do they match what you have on your paper?

Now take a look farther down, at line 10: `birthday-cake = cake("Vanilla", 4, false)`

- What is the name of this variable?
- What is the flavor of `birthday-cake`?
- How many layers does `birthday-cake` have?
- Finally, is `birthday-cake` an ice cream cake, or not?

Below the data definition for CakeType there are four CakeTypes defined:

- `birthday-cake`
- `chocolate-cake`
- `strawberry-cake`
- `red-velvet-cake`

Ask students questions about these CakeTypes to get them thinking about how they would define their own.

On line 14, define another CakeType, which you can name however you like (but choose something descriptive, like `pb-cake`, `lemon-cake`, etc.) To start,

- How would you define this variable?
- What function is used to make a Cake?
- Which thing comes first in a Cake structure?

Now what do you expect to happen when you type the name of your new CakeType into the Interactions Area? Click "Run" and try it out.

Have students walk you through the process of defining a new value and making a `CakeType` with whatever flavor, etc. they like.

```
pb-cake = cake("Peanut Butter", 2, true)
```

Define two new values for some of your favorite cakes. You can give them whatever names you prefer. You can make any kind of `CakeType` that you want, as long as your structure has the right types in the right orders!

Repetition is key in this lesson. Have students identify each part of the `CakeType` for every one they've defined. What is the flavor of their first `CakeType`? Its number of layers? Ensure that students are using their inputs in the right order!

At this point, you've worked with two different ***Data Structures***: JumperStates and `CakeTypes`, and you've created different examples, or ***instances***, of these structures. Instances are concrete uses of a data type, just as 3 is a concrete Number (where Number is the type). Here, `CakeType` is the type, and `cake("Chocolate", 8, false)` is a concrete cake with specific values for each field. In programming, we create instances much more often than we create new data structures. For now, the important point is to recognize the difference between a structure *definition* (the `data....` piece of code) and specific ***instances*** of a data structure (like `birthday-cake`, or `jumper(44, 75)`.

Common Misconceptions

Students often struggle with the difference between the *definition* of a data structure and ***instances*** (items created from) that data structure. When students define `CakeType`, they haven't created any specific cakes. They have defined a type that they can use to define specific cakes. If they have a specific cake, they can ask questions of it such as "is this a chocolate cake?"and produce an answer. If all they have is the `CakeType` definition, they can't answer such questions. Some people like the analogy of a cookie cutter here – `CakeType` defines a cookie cutter, but doesn't produce any cookies. To get a cookie, you use the cake constructor to define a specific cake with specific values for the fields.

Synthesize

Based on these instances of CakeTypes you just wrote:

- What is the name of the function that creates a CakeType?
- What is the Domain of this function?

- How many things are in the domain?

The three things in the domain of cake are, in fact, the three things that we have already listed on Data Structure: CakeType! With data structures, the order is very important: we always want the first string in cake to be the CakeType's flavor, the first number to be its number of layers, etc.

After clicking the "Run" button, in Pyret, type `birthday-cake` into the Interactions Area and hit enter. What do you get back?

Let's make sense of this output. What happens when you type just a number into the Interactions Area? We get that same number back! What about Strings? Images? Booleans? If we don't do anything to our input, or use any function on it, we get back exactly what we put in! Here, you put in a `CakeType`, let's see what we get back. At first glance, it looks like a function call was the answer! But there's a few things different about what appears in the output. First, it isn't the same color as a normal function call, which is the first hint that something's different. Second, we can *click* on it, and see that this value is storing three other values in its ***fields*** — the flavor, layers, and whether or not it's ice cream. This compound value that's printed is an ***instance*** of a `CakeType`. It's a value in its own right, so when we type in `birthday-cake` it shows us this value (just like with numbers and strings).

Remind students that values will always evaluate to themselves. 4 evaluates to 4, the string "pizza" evaluates to "pizza", and birthday-cake evaluates to cake("Vanilla", 4, false)

Getting data from a structure *40 minutes*

Overview

Students are introduced to the syntax of ***dot accessors***, which allow them retrieve data from instances.

Launch

Suppose you want to get the flavor *out* of `chocolate-cake`. You don't care about the message, color, or anything else — you just want to know the flavor. Pyret has syntax for doing precisely that: `.flavor`.

If you type `chocolate-cake.flavor` into the Interactions Area, what should it evaluate to? Try it out!

- What kind of thing did it return: A Number, String, Image, Boolean, or structure?
- Practice taking the flavor out of every `CakeType` you have defined, using `.flavor`

Of course, there are ways to access any part of a `CakeType`, not just the flavor! What do you think you would get if you typed `chocolate-cake.layers` in the Interactions Area?

Try using the dot-accessors `.layers` and `.is-iceCream` on your CakeTypes! Do they do what you expect?

A way to prompt students to use these accessors is to ask: "How do you get the flavor out of a CakeType?" or "How do you get the layers out of a CakeType?" Throughout the course you can set up a call and response system with students, where the question "How do you get the X out of a Y?" will prompt the name of the accessor.

The syntax for getting a field from a structure is known as a ***dot accessor***. They allow you to specify exactly what part of a structure you want. If we want to know if we can fit a certain CakeType through a doorway, we probably care only whether the number of layers is less than a certain amount. Likewise, if we want to know whether or not a character in our game has lost, we need to know only if her health is less than 0: we might not care what her location is, or the color of her armor. Programmers use accessors a lot, because they often need to know only one piece of information from a complex data structure.

Our CakeType structure is defined using `data CakeType:` and the `cake(...)` lines, which tell the computer what things make up that structure, and what order and type each thing is. In return, we get new functions to use. Until we write these lines, we don't have `cake(...)` (to make a Cake), `.flavor` (to get the flavor out of the Cake), `.layers`, or any other dot-accessors, because Pyret doesn't know what a CakeType is — *we haven't defined it*.

To see this for yourself, type a pound sign (#) before the line which begins with `cake(...)` and each of the fields. This comments out the definition, so that the computer ignores it. Hit run, and see what happens.

Investigate

Of course, when programmers work with data structures, they don't just define them and create instances. They also write functions that use and produce structures. Let's get started writing some functions for CakeTypes.

Turn to Word Problem: taller-than in your workbook. Write the contract and purpose statement for a function called taller-than, which consumes two CakeTypes, and produces true if the first CakeType is taller than the second.

- What is the domain for this function?
- What is the range of taller-than?
- Which part(s) of the CakeTypes will you need to check to determine if one is taller than the other?

```
# taller-than :: CakeType, CakeType -> Boolean
# consumes two CakeTypes and produces true if the number of
# layers in the first is greater than the number of
# layers in the second
```

For your first example, try comparing `birthday-cake` and `chocolate-cake`. Do we care about what flavor either of these CakeTypes are? What about whether or not one of them is an ice cream cake? All we need to figure out which one is taller is their number of layers.

How do you get the number of layers out of `birthday-cake`? What about `chocolate-cake`? Write your first example to figure out if `birthday-cake` has a greater number of layers than `chocolate-cake`.

```
examples:
    taller-than(birthday-cake, chocolate-cake) is
    birthday-cake.layers > chocolate-cake.layers
end
```

- Write one more example for the function taller-than, this time using it to compare any two CakeTypes you defined earlier.
- Next, circle and label what changes between the two examples. How many variables will this function need? Then write the definition, using your examples to help you.

After replacing the changing things with variables, your definition should look similar to:

```
fun taller-than(a-cake1, a-cake2):
  a-cake1.layers > a-cake2.layers
end
```

Turn to Word Problem: will-melt in your workbook. Your bakery needs to know if certain CakeTypes needs to be refrigerated. If the temperature is greater than 32 degrees AND the given CakeType is an ice cream cake, the function should return true.

- Fill out the ***Contract*** and ***Purpose Statement*** for the function.
- Write two examples for how one would use `will-melt`.
- Circle and label what varies between those examples and label it with a ***variable*** name.
- Define the function.

Give students plenty of time to practice using dot-accessors, extracting pieces of the Cake structures and writing expressions that compute with them.

Synthesize

Optional: In the Bakery Starter File, extend the `CakeType` data structure to include one more field: a message, represented as a String. (Make sure you remember to change each CakeType instance below the data definition: if a CakeType now contains four fields, each instance will need to include all four fields!) Next, write a function called `make-birthday-cake`, which takes in a string representing someone's name, and produces a 2-layer, chocolate CakeType with "Happy birthday [Name]!" as the message.

Since this function returns a CakeType, remind students that they'll need to use the cake constructor function to produce a CakeType.

Closing *5 minutes*

Data Structures are a powerful tool for representing complex data in a computer program. Simple video games, like Pong, might need to keep track of only a few numbers at once, such as the position of the ball, position of each paddle, and the score. But if a game has many different enemies, each with its own position and health, or multiple levels with their own background images, the game can get very complicated very fast, and structures are a great way to manage and make sense of all the data. Programmers can do a LOT with data structures, and in the upcoming lessons you'll start creating your own structures to make a customized animation.

Additional Exercises

- Students can practice their vocabulary on Vocabulary Practice