










Defining Functions

Students learn a structured approach to problem solving called the “Design Recipe”. They then use these functions to create images, and learn how to apply them to enhance their scatterplots.

Prerequisites	Applying Functions																		
Relevant Standards <div><div>OK K12CS CSTA NGSS CC-Math</div></div>	Select one or more standards from the menu on the left (⌘-click on Mac, Ctrl-click elsewhere).																		
Lesson Goals	<p>Students will be able to...</p> <ul style="list-style-type: none">• define one-argument functions that consume a Number and produce an Image• define one-argument functions that consume a String and produce an Image• define one-argument functions that consume a Row and produce an Image• create custom scatter plots, using functions they have defined• define one-argument functions that make Images from Numbers, Strings, and even Rows• create custom scatter plots using those functions																		
Student-facing Lesson Goals	<ul style="list-style-type: none">• Let's learn how to write our own functions in Pyret.																		
Materials	<ul style="list-style-type: none">• Lesson Slides (Google Slides)• Computer for each student (or pair), with access to the internet• Student workbook, and something to write with• All students should log into CPO and open the "Animals Starter File" they saved from the prior lesson. If they don't have the file, they can open a new one																		
Preparation	<ul style="list-style-type: none">• Make sure all materials have been gathered• Decide how students will be grouped in pairs																		
Supplemental Resources																			
Language Table	<table><tr><th>Types</th><th>Functions</th><th>Values</th></tr><tr><td>Number</td><td>num-sqrt, num-sqr</td><td>4, -1.2, 2/3</td></tr><tr><td>String</td><td>string-repeat, string-contains</td><td>"hello", "91"</td></tr><tr><td>Boolean</td><td>==, <, <=, >=, string-equal</td><td>true, false</td></tr><tr><td>Image</td><td>triangle, circle, star, rectangle, ellipse, square, text, overlay, bar-chart, pie-chart, bar-chart-summarized, pie-chart-summarized</td><td>  </td></tr><tr><td>Table</td><td>count, .row-n, .order-by, .filter, .build-column</td><td></td></tr></table>	Types	Functions	Values	Number	num-sqrt, num-sqr	4, -1.2, 2/3	String	string-repeat, string-contains	"hello", "91"	Boolean	==, <, <=, >=, string-equal	true, false	Image	triangle, circle, star, rectangle, ellipse, square, text, overlay, bar-chart, pie-chart, bar-chart-summarized, pie-chart-summarized	  	Table	count, .row-n, .order-by, .filter, .build-column	
Types	Functions	Values																	
Number	num-sqrt, num-sqr	4, -1.2, 2/3																	
String	string-repeat, string-contains	"hello", "91"																	
Boolean	==, <, <=, >=, string-equal	true, false																	
Image	triangle, circle, star, rectangle, ellipse, square, text, overlay, bar-chart, pie-chart, bar-chart-summarized, pie-chart-summarized	  																	
Table	count, .row-n, .order-by, .filter, .build-column																		

Glossary

design recipe :: a sequence of steps that helps people document, test, and write functions

Overview

Students have learned to define *values* (e.g. - `name = "Maya"` , `x = 5` , etc). Students should have defined `animalA` and `animalB` to be two different rows in the animals table. If they haven't, make sure they do this now.

Launch

Suppose we want to make a solid, green triangle of size 10. What would we type? What if we wanted to make one of size 20? 25? 1000?

```
triangle(10, "solid", "green")
triangle(20, "solid", "green")
triangle(25, "solid", "green")
triangle(1000, "solid", "green")
```

This is a lot of redundant typing, when the only thing changing is the size of the triangle! It would be convenient to define a *shortcut* , which only needs the size. Suppose we call it `gt` for short:

```
gt(10)
gt(20)
gt(25)
gt(1000)
```

We don't need to tell `gt` whether the shape is `"solid"` or `"outline"` , and we don't need to tell it what color to use. We will define our shortcut so it already knows these things, and all it needs is the size. This is a lot like defining values, which we already know how to do. But values don't change, so our triangles would always be the same size. Instead of defining values, we need to define *functions* .

To build our own functions, we'll use a series of steps called the *Design Recipe*. The Design Recipe is a way to think through the behavior of a function, to make sure we don't make any mistakes with the animals that depend on us! The Design Recipe has three steps, and we'll go through them together for our first function.

Turn to [The Design Recipe \(Page 23\)](#) in your Student Workbook, and read the word problem at the top of the page.

Step 1: Contract and Purpose

The first thing we do is write a Contract for this function. You already know a lot about contracts: they tell us the Name, Domain and Range of the function. Our function is named `gt` , and it consumes a Number. It makes triangles, so the output will be an Image. A Purpose Statement is just a description of what the function does:

```
# gt :: (size :: Number) -> Image
# Consumes a size, and produces a solid green triangle of that size.
```

Since the contract and purpose statement are notes for humans, we add the `#` symbol at the front of the line to turn them into comments.

Be sure to check students' contracts and purpose statements before having them move on!

Step 2: Write Examples

Examples are a way for us to tell the computer how our function should behave for a specific input. We can write as many examples as we want, but they must all be wrapped in an `examples:` block and an `end` statement. Examples start with the name of the function we're writing, followed by an example input. Suppose we write `gt(10)` . What work do we have to do, in order to produce the right shape as a result? What if we write `gt(20)` ?

```
# gt :: (size :: Number) -> Image
# Consumes a size, and produces a solid green triangle of that size.
examples:
```

```

gt(10) is triangle(10, "solid", "green")
gt(10) is triangle(10, "solid", "green")
end

```

Step 3: Define the Function

We start with the `fun` keyword (short for “function”), followed by the name of our function and a set of parentheses. This is exactly how all of our examples started, too. But instead of writing `10` or `20`, we’ll use the label from our Domain. Then we add a colon (`:`) in place of `is`, and write out the work we did to get the answers for our examples. Finally, we finish with the `end` keyword.

```

# gt :: (size :: Number) -> Image
# Consumes a size, and produces a solid green triangle of that size.
examples:
  gt(10) is triangle(10, "solid", "green")
  gt(10) is triangle(10, "solid", "green")
end
fun gt(size):
  triangle(size, "solid", "green")
end

```

Investigate

Type your function definition into the Definitions Area. Be sure to include the Contract, Purpose Statement, Examples *and* your Definition! Once you have typed everything in, click “Run” and evaluate `gt(10)` in the Interactions Area. What did you get back?

Once we have defined a function, we can use it as our shortcut! This makes it easy to write simpler code, by moving the complexity into a function that can be tested and re-used whenever we like.

- Use the Design Recipe to solve the word problem at the bottom of [The Design Recipe \(Page 23\)](#).
- Type in the Contract, Purpose Statement, Examples and Definition into the Definitions Area.
- Click “Run”, and make sure all your examples pass!
- Type `bc(20)` into the Interactions Area. What happens?

Synthesize

Ask students what happens if they change one of the examples to be incorrect: `gt(10) is triangle(99, "solid", "green")`

Defining Functions over Other Datatypes

20 minutes

Overview

Students deepen their understanding of function definition and the Design Recipe, by solving different kinds of problems.

Launch

Functions can consume values besides Numbers. For example, we might want to define a function called `sticker` that consumes a `Color`, and draws a star of that color:

```

sticker("blue") is star(50, "solid", "blue")
sticker("yellow") is star(50, "solid", "yellow")

```

Or a function called `nametag` that consumes a `Row` from the `animals` table, and draws that animal’s name in purple letters.

```

nametag(animalA) is text(animalA["name"], 10, "purple")

```

```
nametag(animalB) is text(animalB["name"], 10, "purple")
```

Investigate

Turn to [The Design Recipe \(Page 24\)](#), and use the Design Recipe to write both of these functions.

Custom Scatter Plot Images

15 minutes

Overview

Students discover *functions that consume other functions*, and compose a scatter plot function with one of the functions they've already defined.

Launch

Students have used Pyret functions that use Numbers, Strings, Images, and even Tables and Rows. Now they've written functions of their own that work with these datatypes. However, Pyret functions can even use *other functions*! Have students at the Contract for `image-scatter-plot`:

```
image-scatter-plot :: (t :: Table, xs :: String, ys :: String, f :: (Row -> Image)) -> Image
```

This function looks a lot like the regular `scatter-plot` function. It takes in a table, and the names of columns to use for x- and y-values. Take a closer look at the third input...

```
...f :: (Row -> Image)...
```

That looks like the contract for a function! Indeed, the third input to `image-scatter-plot` is named `f`, which itself is a function that consumes Rows and produces Images. In fact, students have just defined a function that does exactly that!

Investigate

- Type `image-scatter-plot(animals-table, "pounds", "weeks", nametag)` into the Interactions Area.
- What did you get?
- What other scatter plots could we create?

Note: the optional lesson [If Expressions](#) goes deeper into basic programming constructs, using `image-scatter-plot` to motivate more complex (and exciting!) plots.

Synthesize

Functions are powerful tools, for both mathematics and programming. They allow us to create reusable chunks of logic that can be tested to ensure correctness, and can be used over and over to solve different kinds of problems. A little later on, you'll learn how to combine, or *compose* functions together, in order to handle more complex problems.

Additional Exercises:

- [The Design Recipe](#)