Lecture 3.2 – Functions

Specific Learning Objectives:

1.2.1 – Understand the way computers execute commands.

1.2.2 – Create functions in R.

1.2.3 – Use functions to reduce repetitive procedures in a script.

1.2.4 – Use functions to automate and standardize the production of a product (e.g. a graph, an analysis).

1.2.5 – Create a function that vectorizes a calculation.

3.5 – Think and work independently with code.

Scoping: Looking for things

- In order to run a function, R must first find it! The process of searching for objects is called **scoping**.

    - Scoping searches through all the objects trying to match the name that you request and look up the code in order to run the function.

success:

```
> sum(1:10)
[1] 55
```

- If R finds the function, it will run the code and produce the output. If it doesn't, it will return the error "could not find function".

failure:

```
> sum2(1:10)
Error in sum2(1:10) : could not find function "sum2"
```

Scoping and Environments

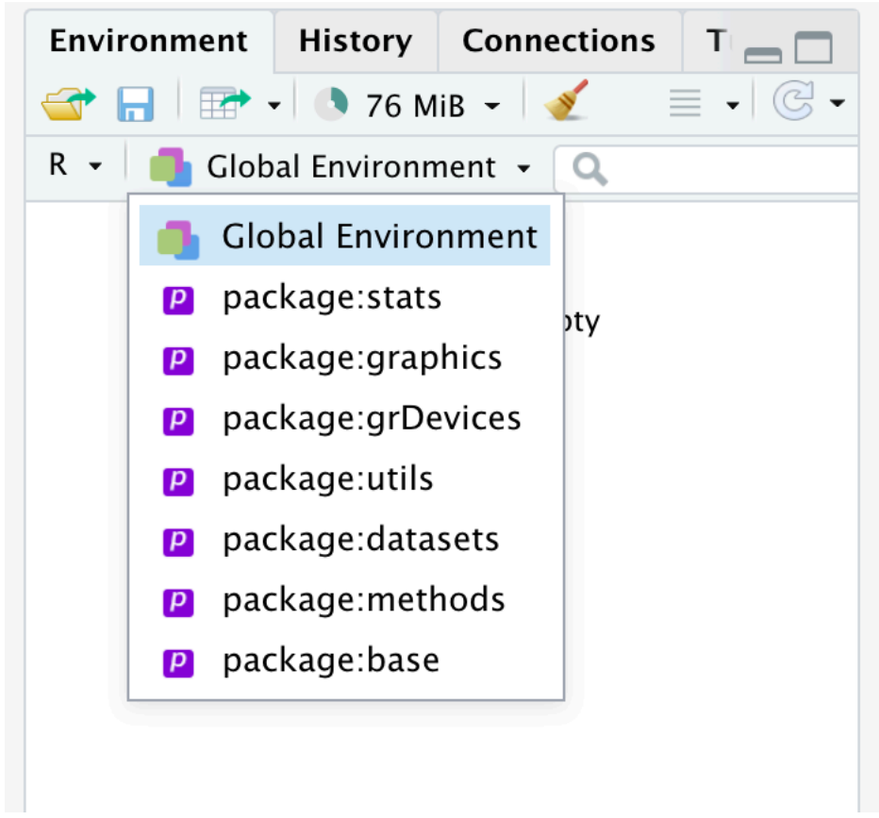
- R has **environments**, or hierarchical collections of objects, to make scoping a little more efficient.

    - The top environment is called the **global environment** and contains all the other environments.

    - Packages create their own environments every time they are loaded.

    - R uses **lexical scoping** to search for objects (including functions) inside each environment throughout the hierarchy.

    - If a package isn't loaded, then the environment doesn't exist!

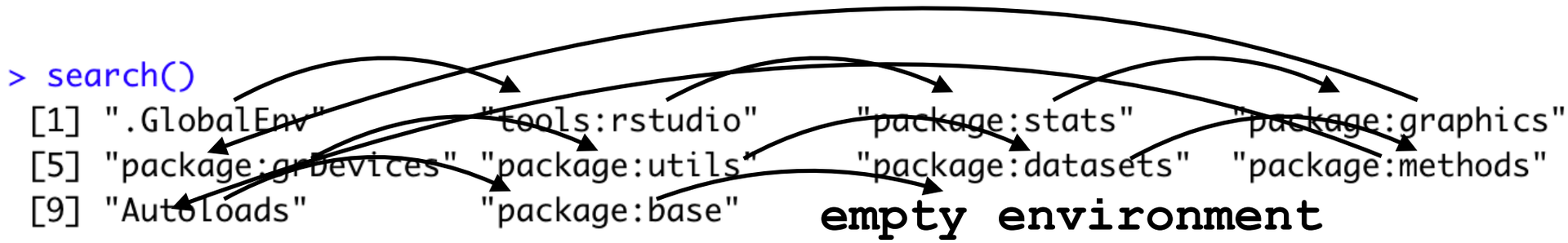


You can check your loaded packages/environments in RStudio!

Scoping and Environments

- Lexical scoping works through the environments in a certain order.

  - Use **`search()`** to see the order of the environments.



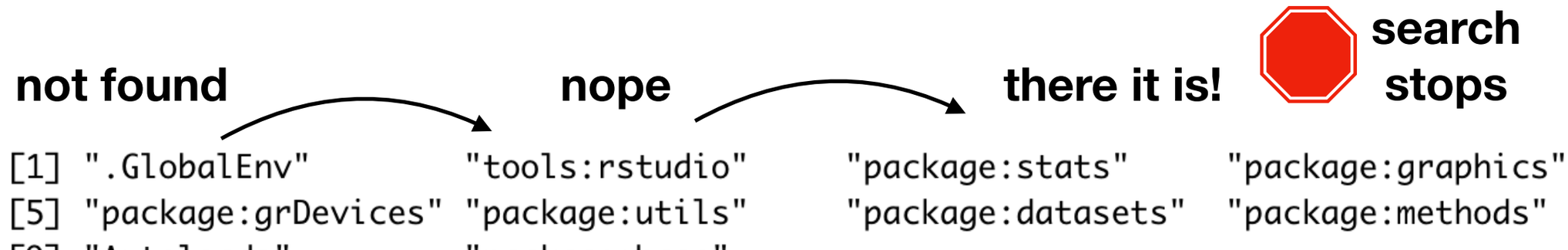
  - This order changes when you load a package!

```
> library(ggplot2)
> search()
 [1] ".GlobalEnv"         "package:ggplot2"    "tools:rstudio"      "package:stats"
 [5] "package:graphics"   "package:grDevices" "package:utils"      "package:datasets"
 [9] "package:methods"    "Autoloads"          "package:base"
```

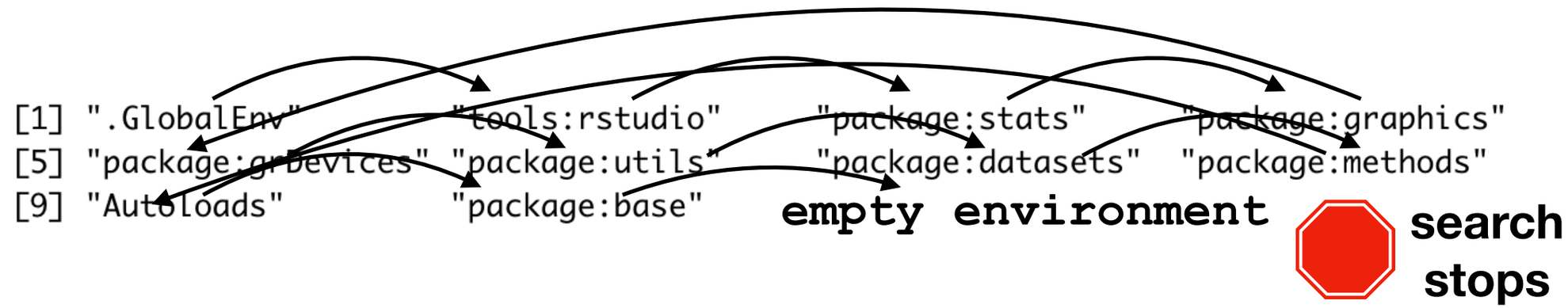
Scoping: Looking for things

- Take a look at how this works in our previous example:

```
> sum(1:10)
[1] 55
```




```
> sum2(1:10)
```

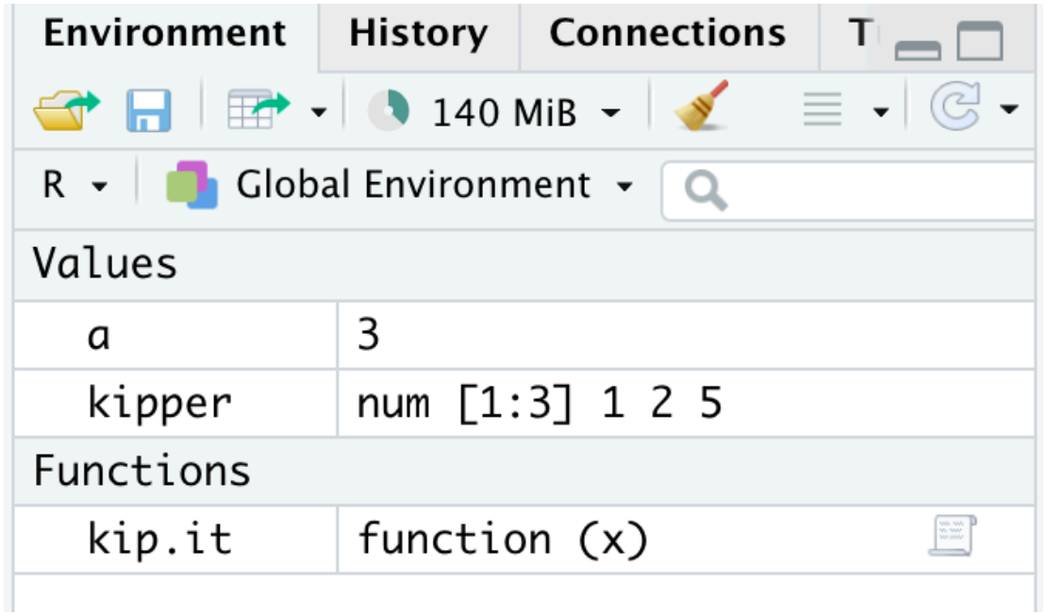



```
Error in sum2(1:10) : could not find function "sum2"
```

Investigating Environments

- The global environment is special and reserved for user-defined objects.
  - To list objects in the global environment, use **`ls()`**.

```
> a <- 3
> kipper <- c(1,2,5)
> kip.it <- function(x) x + 5
> ls()
[1] "a"      "kip.it" "kipper"
>
```



- Package environments can also be investigated using ls(), but need an argument.
  - To list objects in a package, use **`ls("package:packagename")`**.

```
> ls("package:stats")
  [1] "acf"                 "acf2AR"              "add.scope"              "add1"
  [5] "addmargins"          "aggregate"           "aggregate.data.frame"   "aggregate.ts"
  [9] "AIC"                 "alias"               "anova"                  "ansari.test"
 [13] "aov"                 "approx"              "approxfun"              "ar"
```

Check Your Understanding

Add the packages `dplyr` and `ggplot2` to your global environment with `library()`.

To find the function `mean()` in the base package, what environments does R search in what order?

What Use are Functions?

- The one and only thing computers are really good at doing is ***repetition***. Repeating things exactly the same way thousands of times is what computers do best!

    - In order to take advantage of this feature, we need a way of giving a *standard set of instructions* to the computer so it can do the repetition without us having to tell it what to do after every step.

    - **Functions** are a way to do this! Functions are a set of instructions that the computer can take and run without you having to intervene.

    - *Think of functions like recipes*: they tell you all the steps to do to make a cake, but without the original author having to stand over your shoulder!

Functions and their parts

Recipe		Function
Chocolate Cake	Name	`cakefunction <- function`

- 2 cups flour
- 2 large eggs
- 10 oz melted chocolate
- 1 cup milk
	Inputs	`(x,`
		` y,`
		` kind)`

1. Preheat oven to 350°F.
2. Combine eggs, milk, and chocolate and mix until smooth.
3. Mix in flour 1/2 cup at a time until smooth. Pour into cake pan.
3. Bake until done.
	Instructions	`{`
		`  c <- x + y`
		`  b <- x * y`
		`  h <- c + b`
		`  if(class(h) == kind) {`
		`  print("yes")`
		`  } else {`
		`  print("no")`
		`  }`
		`}`

Output: Chocolate Cake	Output	**Output: yes or no**

Constructing Functions

Give your function a unique name

use `function()` to define the function

Include all the inputs you want to pass to the function

```
myfunction <- function(x, y){
  c <- x + y
  b <- x * y
}
```

Include all instructions like you would int a script, between the curly braces!

Run your new function by using it like any other function:

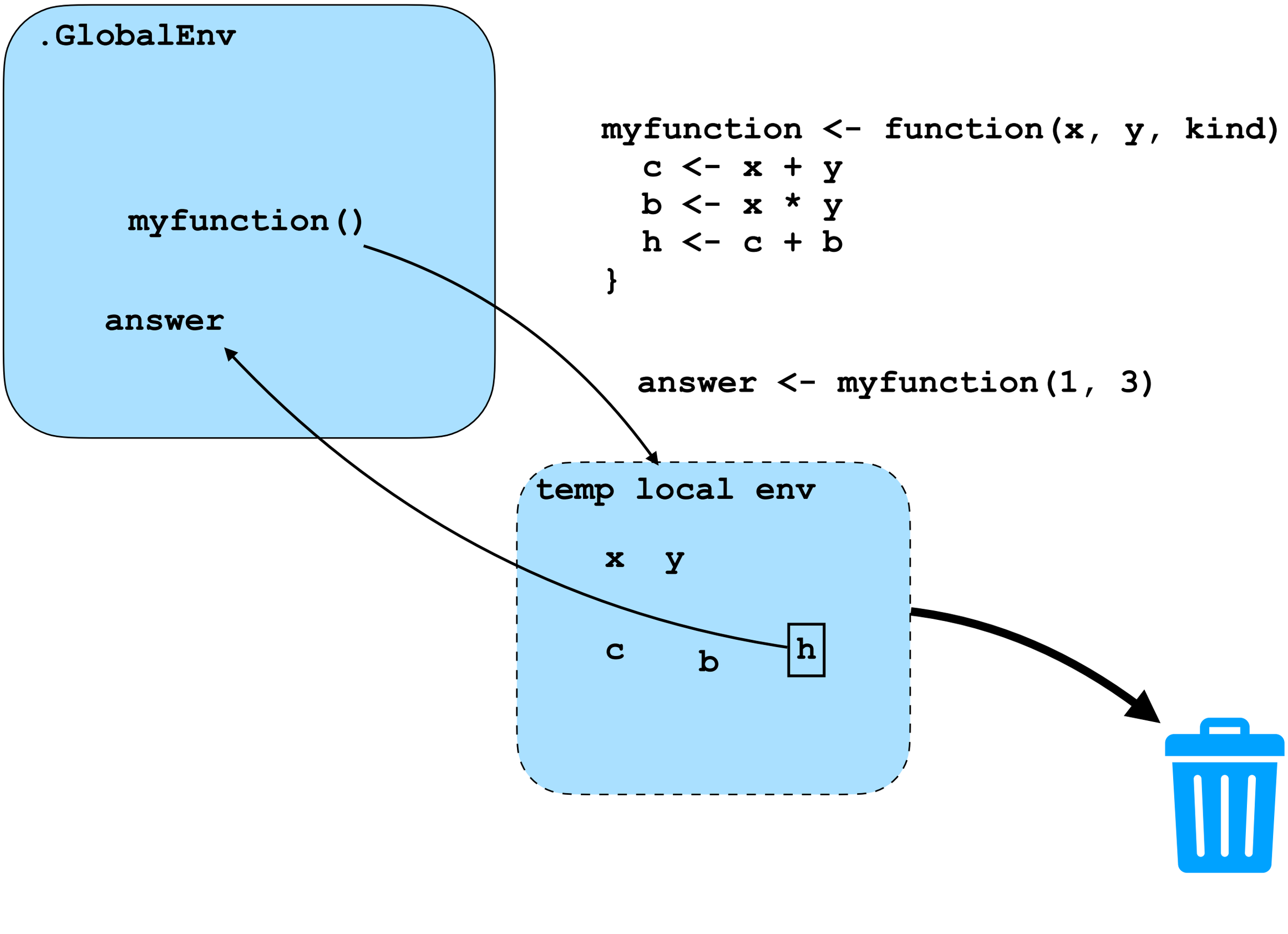
```
myfunction(1, 3)
           ↑  ↑
           x  y
```

Check Your Understanding

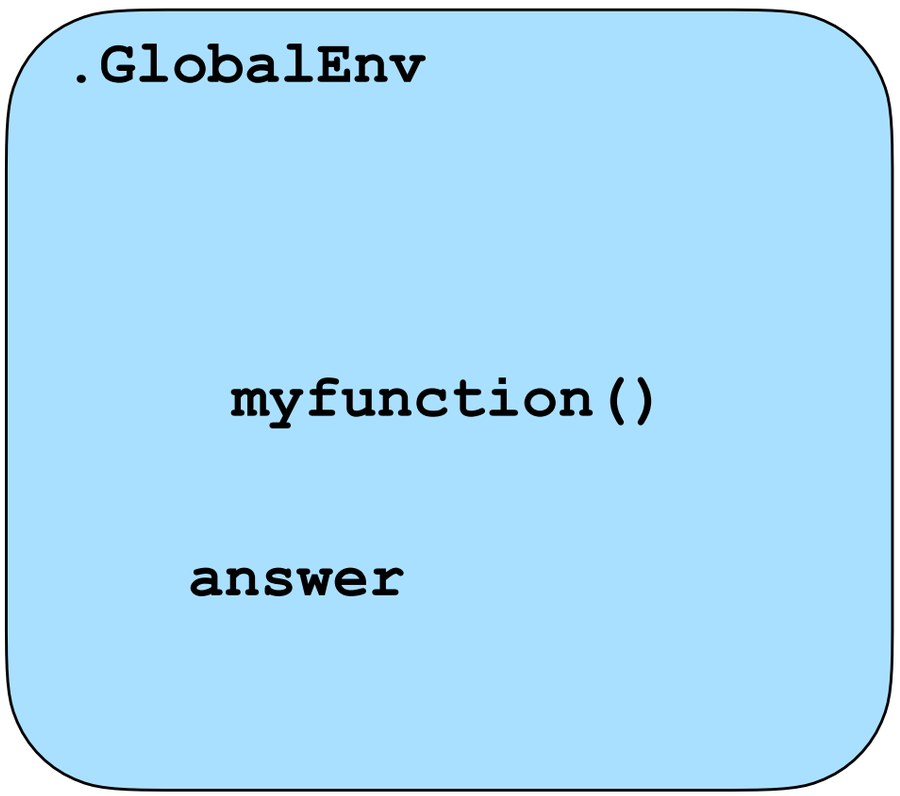
Write a function named `add.these` that takes two input arguments (a and b) and then returns the sum of a and b as an output.

Include a line of code to see if the function works.

Functions and Environments



Functions and Environments

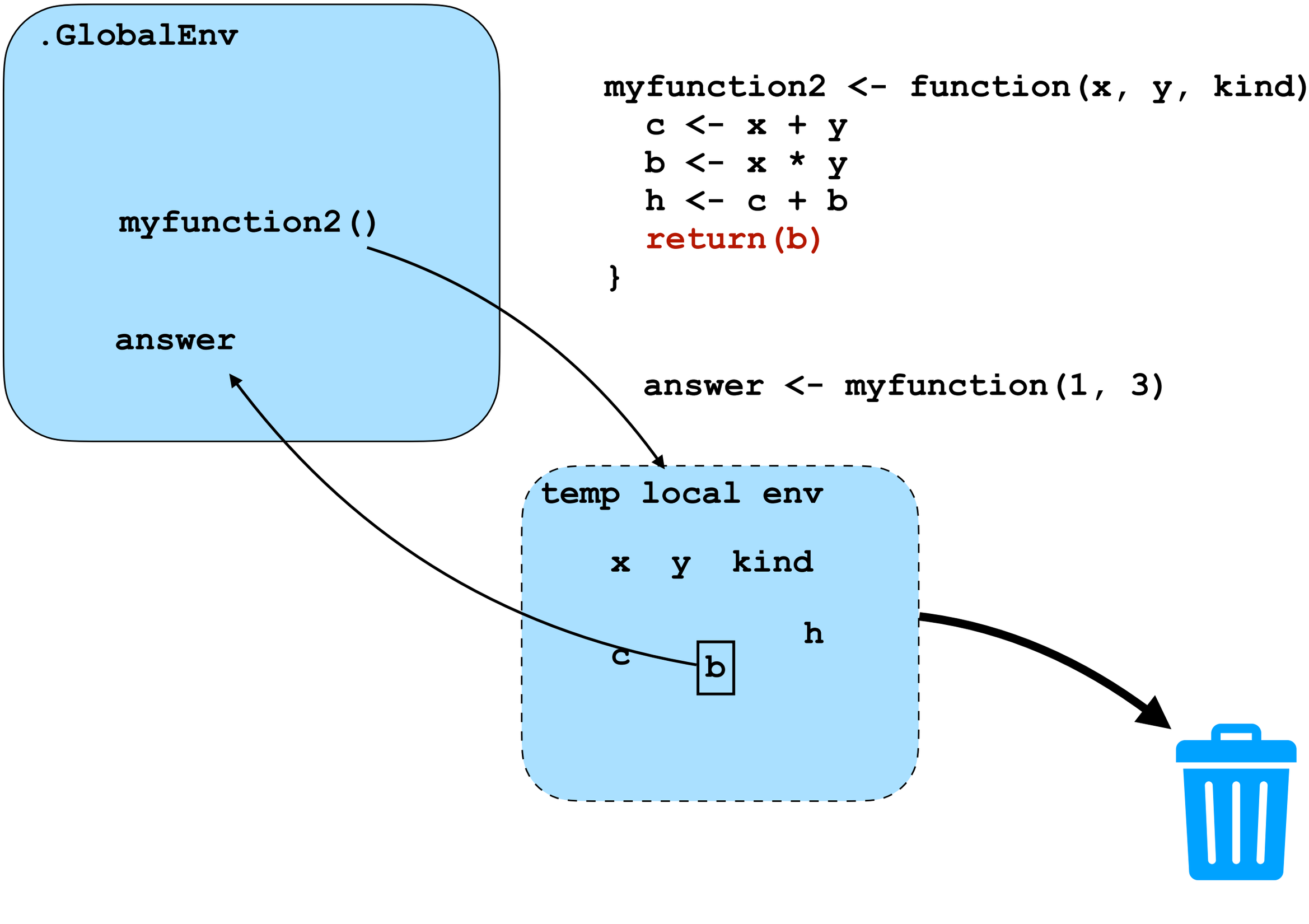



```
myfunction <- function(x, y, kind){
  c <- x + y
  b <- x * y
  h <- c + b
}
```

Where are `c` and `b`?

Functions only return the last object generated or what you tell it to!

Functions and Environments



Check Your Understanding

For the function `whichisit()`:

```
whichisit <- function(j){
  a <- j + 1
  h <- j*10
  c <- 2*(2+j)+10
  d <- j+3
}
```

Running `whichisit(4)` would have what output?

a) 7

c) 22

b) 5

d) 40

Code it and try!

Do It Yourself Functions

- When should you consider turning code into a function? Think repetition!

  - **The ‘Rule of Three’.** Every time you have to copy and paste code a third time, you should write a function.

  - **When you want to loop.** When you’re thinking about turning something into a loop to repeat, you probably want to write a function.

  - **Make things look the same.** When you have figures/charts/tables that you want to look similar without writing long strings of code every time.

Do It Yourself Functions: Example

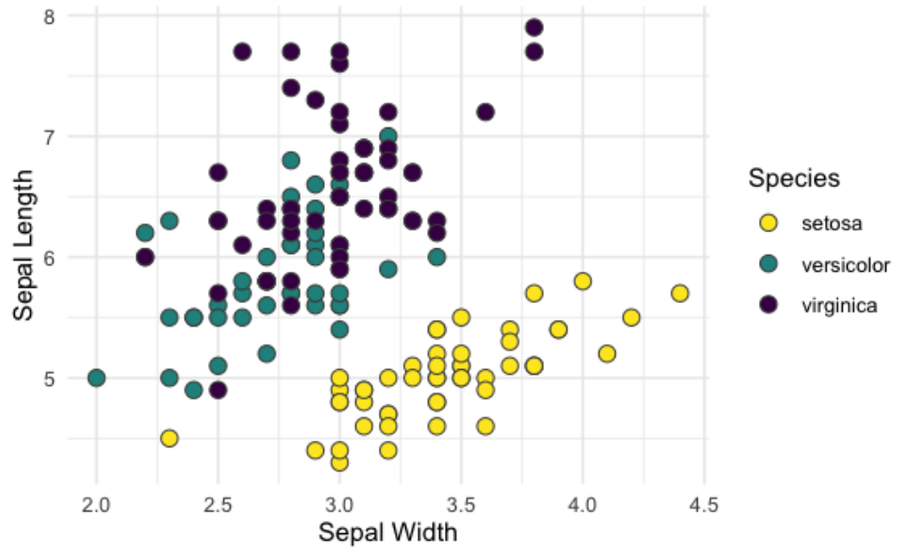
Original copied plots:

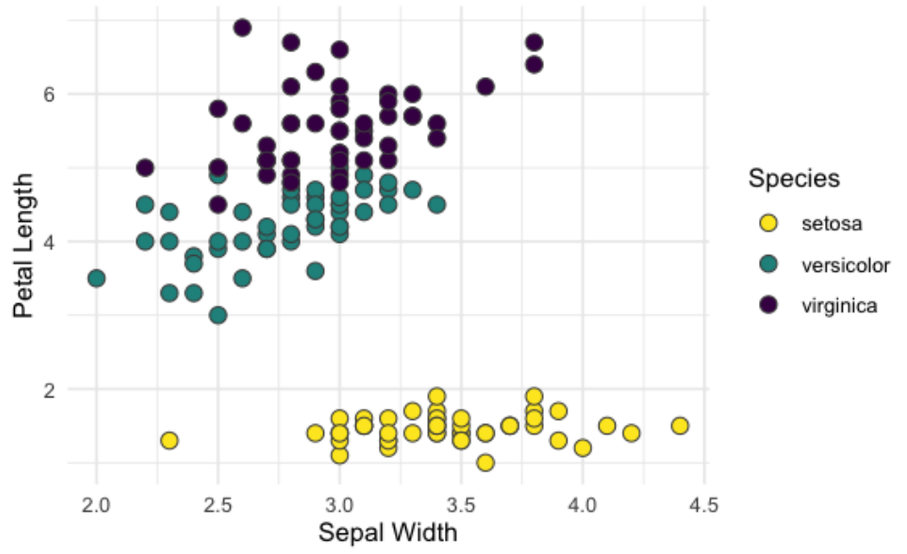
```
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, fill=Species)) +
  geom_point(size = 3, color = "gray30", pch = 21) +
  xlab("Sepal Width") + ylab("Sepal Length") +
  scale_fill_viridis(discrete = TRUE, direction = -1) +
  theme_minimal()

ggplot(iris, aes(x = Sepal.Width, y = Petal.Length, fill = Species)) +
  geom_point(size = 3, color = "gray30", pch = 21) +
  xlab("Sepal Width") + ylab("Petal Length") +
  scale_fill_viridis(discrete = TRUE, direction = -1) +
  theme_minimal()
```

oops

```
ggplot(iris, aes(x = Petal.Width, y = Sepal.Length, fill = Species)) +
  geom_point(size = 3, color = "gray30", pch = 21) +
  xlab("Sepal Width") + ylab("Sepal Length") +
  scale_fill_viridis(discrete = TRUE, direction = -1) +
  theme_minimal()
```

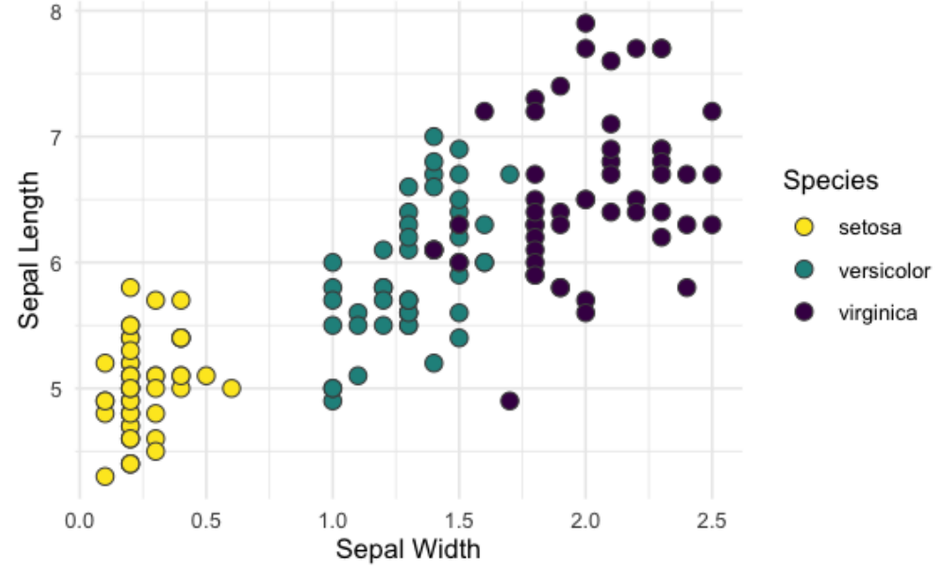




New function:

```
iris.plot <- function(xpos, ypos){
  ggplot(iris, aes_string(x = xpos, y = ypos, fill = "Species")) +
  geom_point(size = 3, color = "gray30", pch = 21) +
  xlab(sub("[.]", " ", xpos)) + ylab(sub("[.]", " ", ypos)) +
  scale_fill_viridis(discrete = TRUE, direction = -1) +
  theme_minimal()
}

iris.plot("Sepal.Width", "Sepal.Length")
iris.plot("Sepal.Width", "Petal.Length")
iris.plot("Petal.Width", "Sepal.Length")
```



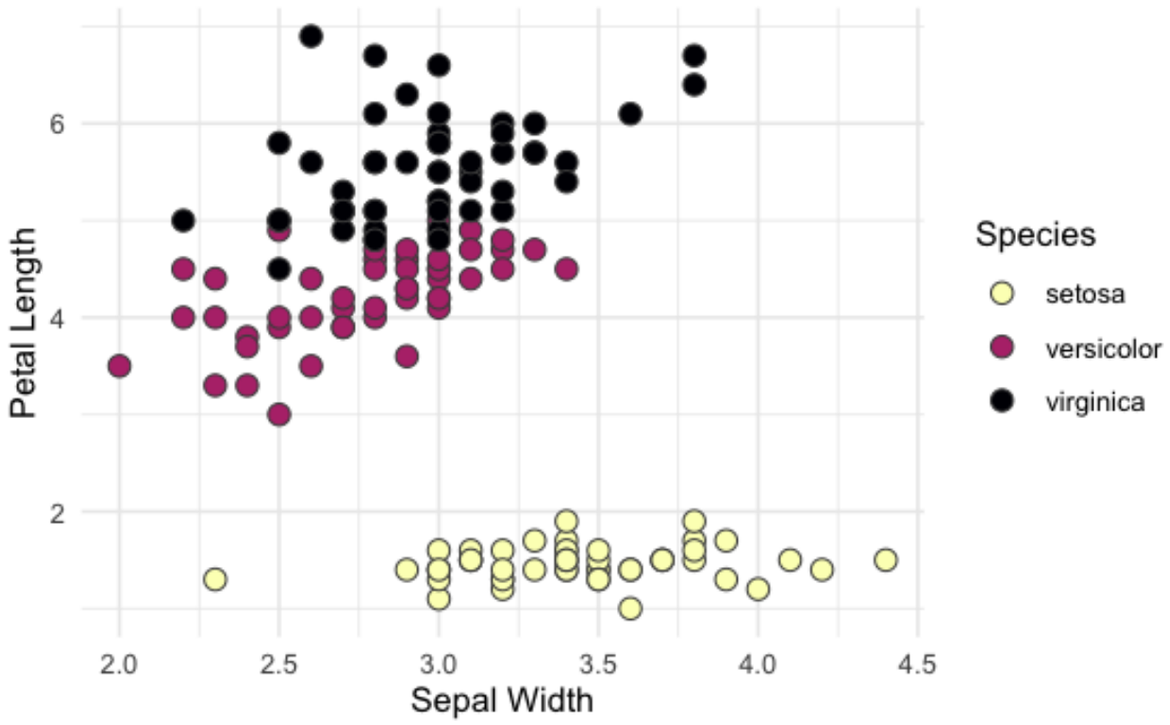
Adding Flexibility

Add options in passes!
Get stuff to work and
then add something else!

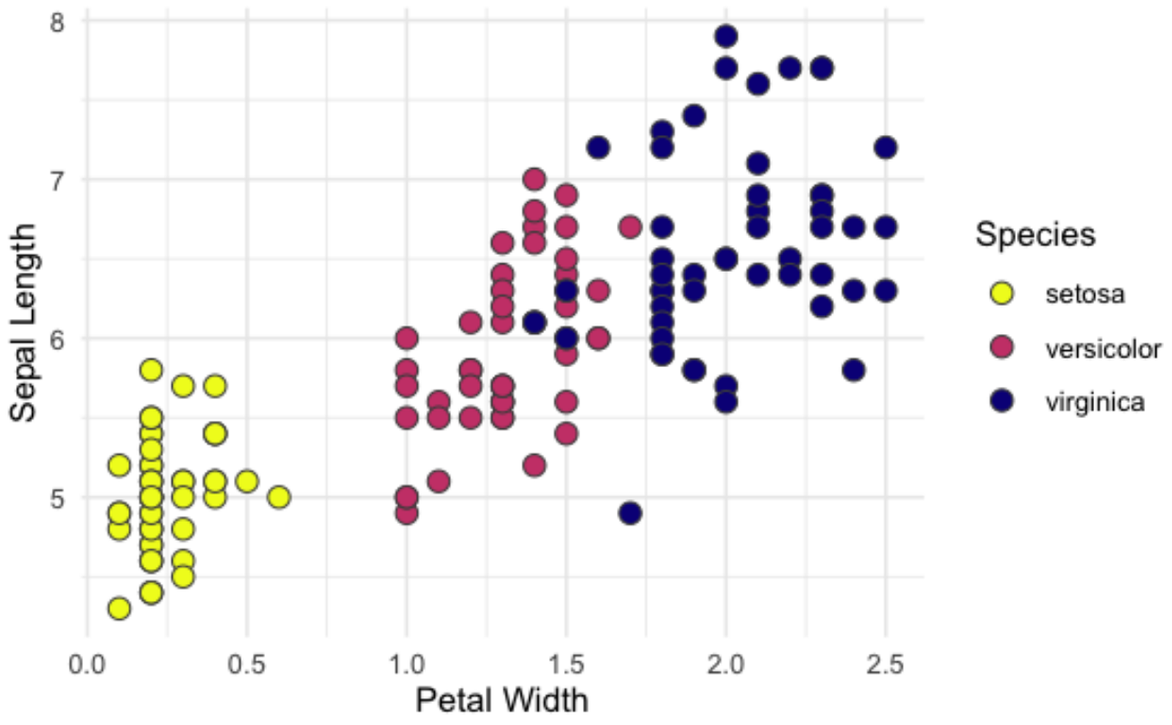
Add option to change viridis palette:

```
iris.plot <- function(xpos, ypos, opt){
  ggplot(iris, aes_string(x = xpos, y = ypos, fill = "Species")) +
  geom_point(size = 3, color = "gray30", pch = 21) +
  xlab(sub("[.]", " ", xpos)) + ylab(sub("[.]", " ", ypos)) +
  scale_fill_viridis(discrete = TRUE, direction = -1, option = opt) +
  theme_minimal()
}
```

```
iris.plot("Sepal.Width", "Petal.Length", opt = "A")
```




```
iris.plot("Petal.Width", "Sepal.Length", opt = "C")
```



Check Your Understanding

Take the following code (see markdown) and make a function that will plot individual stations red. Inputs should be the station number and the output should be a plot with that station's points colored red.

```
plot(x = quakes$depth, y = quakes$mag,
     pch = 21, col = "gray40", bg = "gray80")
points(x = quakes$depth[quakes$stations==12],
 y = quakes$mag[quakes$stations==12],
 pch = 21, bg = "red")
```

As a bonus, you could add an option to change the color of the station's point from red to user-defined.

Action Items

1. **Complete Assignment 3.2.**

2. **Read Davies Ch. 10.1 for next time.**