# R Programming: Environments and Scoping Rules

## Stats 102A

Miles Chen

Acknowledgements: Michael Tsiang and Hadley Wickham (all illustrations)

Week 2 Friday

UCLA

Copyright Miles Chen. For personal use only. Do not distribute.

Section 1

# Binding and Scoping

Copyright Miles Chen. For personal use only. Do not distribute.

# Introduction

Having worked with assigning objects and writing functions already, you should have a basic intuitive understanding of environments and whether objects are accessible inside or outside a function.

- All objects created and stored in an R session are collectively called the **workspace**, also known the **global environment**.
- The body of a function creates a **local environment**, so objects created inside the function are accessible only within the function. Objects created locally will not appear in the global environment
- Objects in the local environment will take precedence over objects in the global environment. In particular, a global object will be **masked** by a local object with the same name: The object name within the function will reference the local object instead of the global one.

We want to formalize these ideas and give a deeper understanding of scope and environments to help us build more complex programs.

Copyright Miles Chen. For personal use only. Do not distribute.

# Name Binding

Consider the simple example:

```
x <- c(1, 2, 3)
```

Informally, we often think of this command as creating an object called `x` which contains the vector of values 1, 2, and 3.

Technically, the assignment (`<-`) operator associates, or **binds**, the name `x` to the vector `c(1,2,3)`.

The command above actually does two things:

- Creates a vector object `c(1,2,3)` in memory.
- Binds the object `c(1,2,3)` to a name `x`.

The name, or **variable**, `x` is just a reference, or **pointer**, to the vector, i.e., `x` points to the memory location on your computer that contains the values `c(1,2,3)`.

Copyright Miles Chen. For personal use only. Do not distribute.

# Name Binding

Consider the following commands:

```
x <- c(1, 2, 3)
y <- x
z <- c(1, 2, 3)
```

**Question**: What is the difference between `y` and `z`?

Copyright Miles Chen. For personal use only. Do not distribute.

# Name Binding

Consider the following commands:

```
x <- c(1, 2, 3)
y <- x
z <- c(1, 2, 3)
```

**Question**: What is the difference between `y` and `z`?

Variables in R are references to memory locations of objects, so `x` and `y` actually point to the same location. The `y` variable is another binding to the same object as `x`.

The `z` variable points to a different copy of `c(1,2,3)`.

While memory allocation issues are typically managed by R automatically, basic understanding of memory management can help in writing cleaner and faster code.

Copyright Miles Chen. For personal use only. Do not distribute.

# Copy-on-modify

```
x <- c(1, 2, 3)
y <- x
y[[3]] <- 4
x
```

```
## [1] 1 2 3
```

Modifying `y` did not modify `x`. At first the names `y` and `x` pointed to the same object in memory.

When R saw that we were changing the value of `y`, it created a copy and modified the copy. The name `x` still points to the original object. The name `y` now points to a different object where the third value has been changed to 4.

Copyright Miles Chen. For personal use only. Do not distribute.

# Scope

Assignment is the act of binding a name to a value.

**Scoping** is act of finding the value associated with a name. The **scope** of a variable is the region of the code where the variable is defined.

The basic scoping rules are fairly intuitive. Consider the following code:

```
y <- 1
g <- function(x) {
  y <- 2
  x + y
}
```

**Question**: What is the output of `g(3)`?

Copyright Miles Chen. For personal use only. Do not distribute.

# Name Masking

Names defined inside a function **mask** names defined outside a function.

```
y <- 1
g <- function(x) {
  y <- 2
  x + y  # finds y in the current environment
}
```

What is g(3)?

```
g(3)
```

```
## [1] 5
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Scoping

The main scoping rule (known as **lexical scoping**): If R cannot find a variable in the function body's scope, it will look for it in the next higher scope, and so on.

While it is considered poor technique to reference objects not defined inside a function, R does its best to find the value rather than return an error.

```
y <- 1
f <- function(x) {
  x + y  # can't find y in this environment, searches higher one
}
```

What is `f(3)`?

Copyright Miles Chen. For personal use only. Do not distribute.

# Scoping

The main scoping rule (known as **lexical scoping**): If R cannot find a variable in the function body's scope, it will look for it in the next higher scope, and so on.

While it is considered poor technique to reference objects not defined inside a function, R does its best to find the value rather than return an error.

```
y <- 1
f <- function(x) {
  x + y  # can't find y in this environment, searches higher one
}
```

What is `f(3)`?

```
f(3)
```

```
## [1] 4
```

Copyright Miles Chen. For personal use only. Do not distribute.

# More Scope

Variables defined within a scope only exist in that scope and only for the duration of that scope. If variables exist with the same name outside the scope, those are separate and do not interact with the variables inside the scope, nor are they overwritten.

A function is its own separate environment that only communicates to the outside world via the arguments going in and the returned object going out.

For example, if you execute the command `rm(list = ls())` inside a function (a terrible idea, btw), you would only delete the objects that have been defined inside the function.

The exception is the **super assignment** (`<<-`) operator which searches a higher scope and the global environment. This is a dangerous operator and should not be used regularly.

Copyright Miles Chen. For personal use only. Do not distribute.

# Scoping

```
x <- 1
y <- 1
z <- 1
f <- function() {
    y <- 2  # creates y inside the scope of f()
    g <- function() {  # this function is created inside f()
      z <- 3  # creates z inside the scope for g()
      return(x + y + z) # R uses scope rules to search for these values
      }
    return(g())
}
```

Question: If I run `f()`, what will it return?

Question: After running `f()`, what are the values of x, y, z in the global environment?

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
f()
```

```
## [1] 6
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
f()
```

```
## [1] 6
```

```
c(x, y, z)
```

```
## [1] 1 1 1
```

Copyright Miles Chen. For personal use only. Do not distribute.

Section 2

# Environments

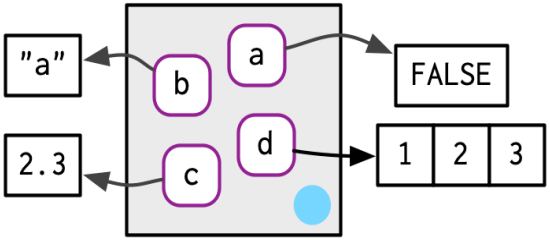
Copyright Miles Chen. For personal use only. Do not distribute.

# Environments

The **environment** is the data structure that powers scoping.

The job of an environment is to associate, or **bind**, a set of names to a set of values. You can think of an environment as a bag of names, each name pointing to an object stored elsewhere in memory.

```
e <- new.env()
e$a <- FALSE
e$b <- "a"
e$c <- 2.3
e$d <- 1:3
```




Copyright Miles Chen. For personal use only. Do not distribute.

# Environment Objects

As with everything in R, environments are also objects. The syntax to work with environment objects is similar to lists in that the double bracket `[[` and `$` operator can be used to get and set values.

```
typeof(e)
```

```
## [1] "environment"
```

```
e$c
```

```
## [1] 2.3
```

```
e[["d"]]
```

```
## [1] 1 2 3
```

```
e[[1]] # Objects inside environments are not ordered, so numeric index produces error
```

```
## Error in e[[1]]: wrong arguments for subsetting an environment
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Environment Objects

The `ls()` and `ls.str()` functions are useful for looking at the objects inside or the structure of the environment.

```
ls(e)
```

```
## [1] "a" "b" "c" "d"
```

```
ls.str(e)
```

```
## a :  logi FALSE
## b :  chr "a"
## c :  num 2.3
## d :  int [1:3] 1 2 3
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Environment Objects

Unlike lists:

- The single square bracket `[` does not work for environments.
- Setting an object to `NULL` does not remove the object.
- Removing objects can be done using the `rm()` function.

```
e$d <- NULL
ls(e)
```

```
## [1] "a" "b" "c" "d"
```

```
rm(d, envir = e)
ls(e)
```

```
## [1] "a" "b" "c"
```

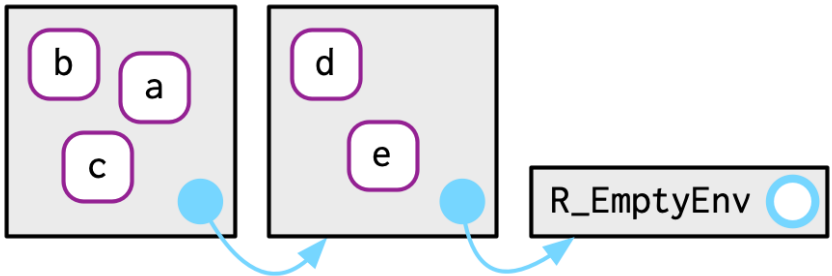
Copyright Miles Chen. For personal use only. Do not distribute.

# Parent Environments

Every environment has a **parent**, another environment. In the diagrams, the pointer to the parent is a small blue circle.

The parent is used to implement **lexical scoping**: If a name is not found in an environment, then R will look in its parent (and so on).

Only one environment does not have a parent: the empty environment.




Copyright Miles Chen. For personal use only. Do not distribute.

# The Important Environments

There are four special environments:

- The `globalenv()`, or **global environment**, also called the **workspace**. This is the environment in which you normally work. The parent of the global environment is the last package that you attached with `library()` or `require()`.
- The `baseenv()`, or **base environment**, is the environment of the base package. Its parent is the empty environment.
- The `emptyenv()`, or **empty environment**, is the ultimate ancestor of all environments, and the only environment without a parent.
- The `environment()` is the current environment.

Copyright Miles Chen. For personal use only. Do not distribute.

# The Search Path

The `search()` function lists all parents of the global environment. This is called the **search path** because objects in these environments can be found from the top-level interactive workspace.

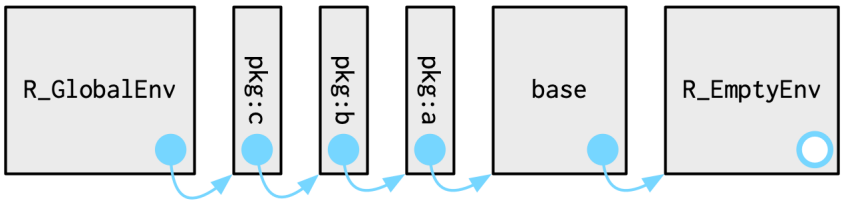
```
search()
```

```
##  [1] ".GlobalEnv"        "package:knitr"     "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

Copyright Miles Chen. For personal use only. Do not distribute.

# The Search Path Visualized

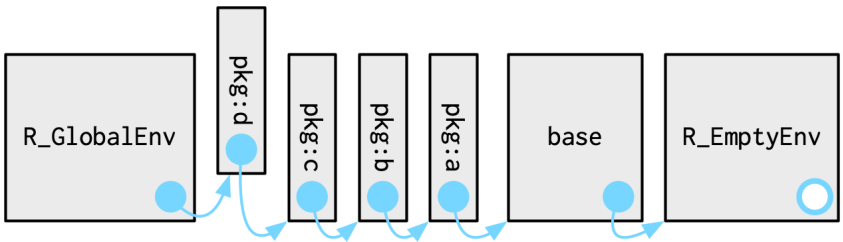
The `globalenv()`, `baseenv()`, the environments on the search path, and `emptyenv()` are connected as shown below.




Copyright Miles Chen. For personal use only. Do not distribute.

# The Search Path Visualized

Each time you load a new package with `library()`, the package environment is inserted between the global environment and the package that was previously at the top of the search path.




Note that the parent of the global environment changes.

Copyright Miles Chen. For personal use only. Do not distribute.

# Function Environments

Most environments are created as a consequence of using functions. This section discusses the four types of environments associated with a function: enclosing, binding, execution, and calling.

The **enclosing** environment is the environment where the function was created. Every function has one and only one enclosing environment. For the three other types of environment, there may be 0, 1, or many environments associated with each function:

- Binding a function to a name with the assignemnt <- operator defines a **binding** environment.
- Calling a function creates a temporary **execution** environment that stores variables created during execution.
- Every execution environment is associated with a **calling** environment, which tells you where the function was called.

Copyright Miles Chen. For personal use only. Do not distribute.

# The Enclosing and Binding environments

When a function is created, it gains a reference to the environment where it was made. This is the enclosing environment and is used for lexical scoping.

When you name a function, the environment where the name exists, is the binding environment.

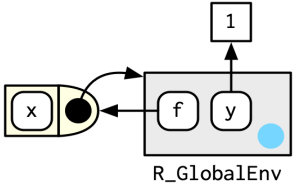
In most scenarios, the enclosing environment and binding environment is the same.

Copyright Miles Chen. For personal use only. Do not distribute.

# Enclosing and Binding Environments

```
y <- 1
f <- function(x) {
  x + y
}
environment(f)
```

```
## <environment: R_GlobalEnv>
```




The function `f` is created in the `globalenv()`, so that is the enclosing environment. The name `f` exists in the `globalenv()` so it is the binding environment.

Copyright Miles Chen. For personal use only. Do not distribute.

# Package Environments

In the search path, we saw that the parent environment of a package varies based on what other packages have been loaded. Does this mean that the package will find different functions if packages are loaded in a different order?

For example, consider the `sd()` function:

```
sd
```

```
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x0000000015b785c0>
## <environment: namespace:stats>
```

The `sd()` function is defined in terms of the `var()` function. How does R make sure that the `sd()` function is not affected by any function called `var()` in the global environment or in one of the attached packages in the search path?

Copyright Miles Chen. For personal use only. Do not distribute.

# Namespaces

The distinction between enclosing and binding environments is particularly important for functions inside packages.

To ensure that every package works the same way regardless of what packages are attached by the user, packages require an internal environment called the **namespace** in which the package functions are defined.

Every function in a package is associated with a pair of environments: the package environment and the namespace environment.

Copyright Miles Chen. For personal use only. Do not distribute.

# Package Environments and the :: Operator

The **package environment** is the external interface to the package. This is the environment which the R user uses to find a function in an attached package. Its parent is determined by the search path, i.e. the order in which packages have been attached.

The **double colon** `::` operator can be used to access a function directly from the package environment (regardless of the search path).

For example, `base::mean()` refers to the `mean()` function from the `base` package, while `mosaic::mean()` refers to the `mean()` function from the `mosaic` package.

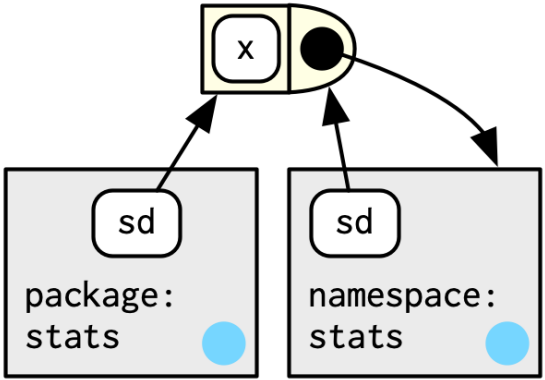
Double colon notation allows you to access objects that may be masked by other objects from packages higher in the search path.

Copyright Miles Chen. For personal use only. Do not distribute.

# Namespace Environments

The **namespace environment** is the internal interface to the package.

The package environment controls how we find the function; the namespace controls how the function finds its variables.




Copyright Miles Chen. For personal use only. Do not distribute.

# Execution Environments

Each time a function is called, a new environment is created to host execution. The parent of the execution environment is the enclosing environment of the function. Once the function has completed, this environment is thrown away.

Let's look at a simple function.

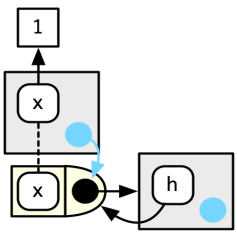
```
h <- function(x) {
  a <- 2
  x + a
}
y <- h(1)
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Execution Environments

```
h <- function(x) {
  a <- 2
  x + a
}
y <- h(1)
```

**1.** Function called with x = 1

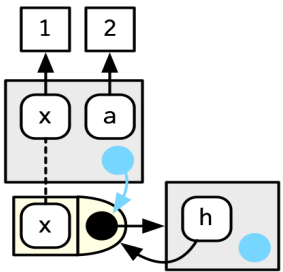



Copyright Miles Chen. For personal use only. Do not distribute.

# Execution Environments

```
h <- function(x) {
  a <- 2
  x + a
}
y <- h(1)
```

**2.** a bound to value 2

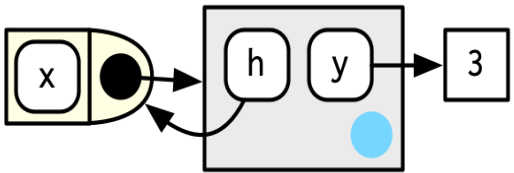



Copyright Miles Chen. For personal use only. Do not distribute.

# Execution Environments

```
h <- function(x) {
  a <- 2
  x + a
}
y <- h(1)
```

**3.** Function completes returning value 3.
Execution environment goes away.




Copyright Miles Chen. For personal use only. Do not distribute.

# Calling Environment

When you call a function, it creates an execution environment. The execution environment has two parent environments: the enclosing environment (where the function is created) and calling environment (where the function is called).

If you created a function in the `globalenv()` and call the function in the `globalenv()`, then these are the same. But in some cases, you might call a function from within another function. In this case, the environments are different.

R's normal scoping rules will use the enclosing environment, so if the function is looking for values, it looks in the enclosing environment.

However, R also supports **dynamic scoping** where you can look for values in the calling environment. This is achieved with the function `get()`

Copyright Miles Chen. For personal use only. Do not distribute.

# Scoping Example 1

```
x <- 0
y <- 10
f <- function() {
  x <- 2
  y <- 100
  h <- function() { # we define h inside f
    x <- 50
    x + y
  }
  h() # we are only calling function h and this is the value it returns
}
```

**Question**: What will `f()` return?

Copyright Miles Chen. For personal use only. Do not distribute.

# Answer

`h()` is defined in `f()`.

`h()` returns the value `x + y`.

It finds `x` in its own execution environment with value 50.

It does not find `y`, so it searches the enclosing environment which is the execution environment of funtion `f()`.

Inside `f()`'s environment, it finds `y <- 100`. It uses that and returns `50 + 100 = 150`

```
f()
```

```
## [1] 150
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Scoping Example 2

```
x <- 0
y <- 10
g <- function() {
  x <- 2
  y <- 100
  h() # we are only calling function h inside g
}
h <- function() { # we define h in global
  x <- 3
  x + y
}
```

**Question**: What will `h()` return?

**Question**: What will `g()` return?

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

`h()` is defined in the global environment.

`h()` returns the value `x + y`.

It finds `x` in its own execution environment with value 3.

It does not find `y`, so it searches the enclosing environment which is the global environment, where `y <- 10`. It uses that and returns `3 + 10 = 13`

```
h()
```

```
## [1] 13
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

`g()` assigns some values in its own environment and returns the value produced by calling `h()`.

`h()` is not defined in `g()`, so R searches the higher scope and finds `h()` defined in the global environment.

`h()` returns the value `x + y`. It finds `x` in its own execution environment with value 3. It does not find `y`, so it searches the enclosing environment which is the global environment, where `y <- 10`.

It uses that and returns `3 + 10 = 13`

```
g()
```

```
## [1] 13
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Dynamic Scoping

```
x <- 0
y <- 10
g <- function() {
  x <- 2
  y <- 100
  h() # we are only calling function h inside g
}
h <- function() { # we define h in global
  x <- 3
  y <- get("y", envir = parent.frame()) # dynamic scoping
  x + y
}
```

**Question**: What will `h()` return?

**Question**: What will `g()` return?

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
h()
```

```
## [1] 13
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
h()
```

```
## [1] 13
```

```
g()
```

```
## [1] 103
```

Copyright Miles Chen. For personal use only. Do not distribute.

Section 3

# Super Assignment

Copyright Miles Chen. For personal use only. Do not distribute.

# Super Assignment

The regular assignment <- operator always creates variables within the current environment.

The **super assignment <<-** operator never creates a variable in the current environment but instead **modifies** an existing variable found in the parent environment.

**Warning**: If (<<-) does not find an existing variable in the parent environment, it will climb the scope ladder until it finds the variable it is looking for. If it reaches the global environment without finding the variable, it **creates** the variable in the global environment.

**Super assignment should generally be avoided.**

Copyright Miles Chen. For personal use only. Do not distribute.

# The Super assignment operator

```
x <- 1; y <- 1; z <- 1
f <- function(){
  y <<- 3
  return(y)
}
```

Copyright Miles Chen. For personal use only. Do not distribute.

# The Super assignment operator

```
x <- 1; y <- 1; z <- 1
f <- function(){
  y <<- 3
  return(y)
}
```

```
f()
```

```
## [1] 3
```

```
c(x, y, z)
```

```
## [1] 1 3 1
```

Copyright Miles Chen. For personal use only. Do not distribute.

# The Super assignment operator

```
x <- 1; y <- 1; z <- 1 # created in the global environment
f <- function() {
    y <<- 2 # modifies the value of y in the scope higher up (global) to 2
    g <- function() { # g() defined inside the environment of f()
      z <<- 3  # modifies the value of z to 3 in the higher scope: f,
               # but does not find z in f, so climbs higher: global
      return(x + y + z) # R uses scope rules to search for these values
      }
    return(g())
}
```

Question: If I run `f()`, what will it return?

Question: After running `f()`, what are the values of x, y, z in the global environment?

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
f()
```

```
## [1] 6
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
f()
```

```
## [1] 6
```

```
c(x, y, z)
```

```
## [1] 1 2 3
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Super assignment can be tricky

```
x <- 1; y <- 1; z <- 1
f <- function(){
    y <- 2  # creates y in the current scope (inside f)
    y <<- 4 # modifies y in the higher scope (global)
    g <- function(){
      z <<- 3  # modifies z in the scope higher up f,
               # but does not find z, so goes to the higher scope (global)
      return(x + y + z) # searches for y in the current scope.
               # does not find y inside g, but does in f. Uses that value.
      }
    return(g())
}
```

Question: If I run `f()`, what will it return?

Question: After running `f()`, what are the values of x, y, z in the global environment?

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
f() # because x = 1, y = 2 (in the scope of f), and z = 3
```

```
## [1] 6
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
f() # because x = 1, y = 2 (in the scope of f), and z = 3
```

```
## [1] 6
```

```
c(x, y, z) # the values of x y and z in the global environment
```

```
## [1] 1 4 3
```

Copyright Miles Chen. For personal use only. Do not distribute.

# More super assignment

```
x <- 1; y <- 1; z <- 1
f <- function(){
    y <- 2
    z <- 10
    y <<- 4 # modifies y in the higher scope (global)
    g <- function(){
      z <<- 3 # modifies z in the scope higher up: f
              # does not touch the z in the global environment
      return(x + y + z) # uses the scoping rules when searching
      }
    return(g())
}
```

Question: If I run `f()`, what will it return?

Question: After running `f()`, what are the values of x, y, z in the global environment?

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
f()
```

```
## [1] 6
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Answers

```
f()
```

```
## [1] 6
```

```
c(x, y, z)
```

```
## [1] 1 4 1
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Avoid Super Assignment

You should avoid using super assignment.

Let's say you want to update some object `foo` by appending the value of x-squared to it

```
# BAD:
add_sq_bad <- function(foo, x){
  foo <<- c(foo, x ^ 2) # combines foo with x^2 and super assigns back to foo
}

foo <- 2
add_sq_bad(foo, 5)
foo # so this seemed to work
```

```
## [1]  2 25
```

Copyright Miles Chen. For personal use only. Do not distribute.

# Dangers of super assignment

```
# Let's try the function with a different object, bar

bar <- 10
add_sq_bad(bar, 6)
```

Question: What is the value of `bar` in the global environment? What is the value of `foo` in the global environment?

Copyright Miles Chen. For personal use only. Do not distribute.

# Dangers of super assignment

```
bar # bar in unchaged
```

```
## [1] 10
```

```
foo  # foo changed. probably not what you wanted
```

```
## [1] 10 36
```

Copyright Miles Chen. For personal use only. Do not distribute.

# A better way to change values in the global environment

```
add_sq_good <- function(baz, x){
  c(baz, x^2)
}

foo <- 2
foo <- add_sq_good(foo, 5)  # the assignment of the output to the object is explicit
foo
```

```
## [1]  2 25
```

```
bar <- 10
bar <- add_sq_good(bar, 5)
bar
```

```
## [1] 10 25
```

Copyright Miles Chen. For personal use only. Do not distribute.