

MT4113 Lecture 3

Good Programming Practice

Eiren Jacobson

26 September 2018

Topics Covered

- The advantages of using **functions**

Topics Covered

- The advantages of using **functions**
- Different types of **environments**

Topics Covered

- The advantages of using **functions**
- Different types of **environments**
- Understanding the **scope** of variables

Topics Covered

- The advantages of using **functions**
- Different types of **environments**
- Understanding the **scope** of variables
- Software **design** for statisticians

Topics Covered

- The advantages of using **functions**
- Different types of **environments**
- Understanding the **scope** of variables
- Software **design** for statisticians
- Coding **conventions and style**

Functions

Why use functions?

- Don't Repeat Yourself
 - Avoid repeating the same code over and over

Why use functions?

- Don't Repeat Yourself
 - Avoid repeating the same code over and over
- Functions Do One Thing
 - Small chunks with defined inputs and outputs are easier to test

Why use functions?

- Don't Repeat Yourself
 - Avoid repeating the same code over and over
- Functions Do One Thing
 - Small chunks with defined inputs and outputs are easier to test
- Easier to update and propagate changes
 - Only have to change things once

Why use functions?

- Don't Repeat Yourself
 - Avoid repeating the same code over and over
- Functions Do One Thing
 - Small chunks with defined inputs and outputs are easier to test
- Easier to update and propagate changes
 - Only have to change things once
- Divide and conquer the problem
 - Each function is a step of the analysis

Why use functions?

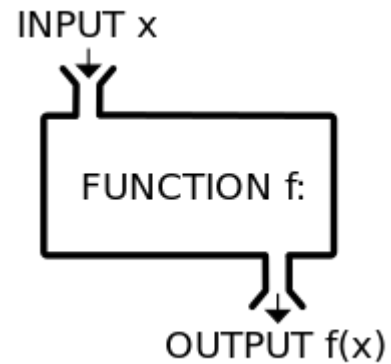
- Don't Repeat Yourself
- Functions Do One Thing
- Easier to update and propagate changes
- Divide and conquer the problem

Why use functions?

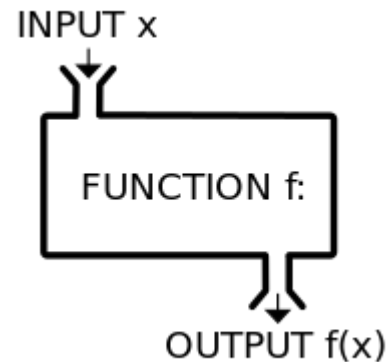
- Don't Repeat Yourself
- Functions Do One Thing
- Easier to update and propagate changes
- Divide and conquer the problem
- *Disadvantage: slight increase in overhead time*

What happens in $f(x)$ stays in $f(x)$

What happens in $f(x)$ stays in $f(x)$

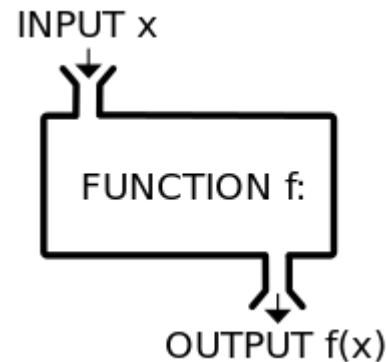


What happens in `f(x)` stays in `f(x)`



- Concept of **encapsulation**
 - send everything into function as arguments
 - return outputs explicitly with `return()` function
 - no global side effects 🚫🌍

What happens in `f(x)` stays in `f(x)`



- Concept of **encapsulation**
 - send everything into function as arguments
 - return outputs explicitly with `return()` function
 - no global side effects 🚫🌍
- Whenever possible, functions should fit on one screen

What happens in $f(x)$ stays in $f(x)$

What happens in `f(x)` stays in `f(x)`

```
x ← 1:25
```

```
func ← function() {  
  sum(x)/length(x)  
}
```

```
func()
```

```
# [1] 13
```

What happens in `f(x)` stays in `f(x)`

```
x ← 1:25  
  
func ← function() {  
  sum(x)/length(x)  
}  
  
func()
```

```
# [1] 13
```

```
# my.mean calculates the mean  
# of a numeric vector x
```

```
my.mean ← function(x) {  
  stopifnot(is.numeric(x))  
  m ← sum(x)/length(x)  
  return(m)  
}
```

```
x ← 1:25  
my.mean(x)
```

```
# [1] 13
```

- **Q:** What are three things that make the example on the right better?

What happens in `f(x)` stays in `f(x)`

- Example from last class:

```
eda ← function (x) {  
  par(mfrow = c(1,3))  
  hist(x, probability = TRUE)  
  lines(density(x))  
  boxplot(x, horizontal = TRUE)  
  rug(x)  
  qqnorm(x)  
  return(summary(x))  
}
```

- Issues: no comments and global change in `par()`

What happens in `f(x)` stays in `f(x)`

```
eda ← function (x) {  
  # Purpose: Exploratory data analysis function  
  # Input: numeric vector x  
  # Output: summary object and 1×3 plot  
  
  # Setup 1×3 graphical device, but make sure  
  # it reverts to current settings on exit  
  old.par ← par(no.readonly = TRUE)  
  on.exit(par(old.par))  
  par(mfrow = c(1,3))  
  .  
  .  
  .  
}
```

What happens in `f(x)` stays in `f(x)`

```
print.and.multiply ← function(x, y) {  
  print(paste('At start of function x=', x, 'y=', y))  
  x ← x*y  
  print(paste('At end of function x=', x))  
  return(x)  
}  
  
first ← 10  
second ← 20  
new.object ← print.and.multiply(first, second)
```

```
## [1] "At start of function x= 10 y= 20"  
## [1] "At end of function x= 200"
```

What happens in `f(x)` stays in `f(x)`

```
print.and.multiply ← function(x, y) {  
  print(paste('At start of function x=', x, 'y=', y))  
  x ← x*y  
  print(paste('At end of function x=', x))  
  return(x)  
}  
  
first ← 10  
second ← 20  
new.object ← print.and.multiply(first, second)
```

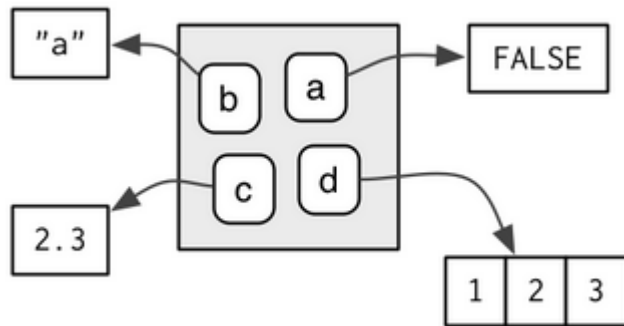
```
## [1] "At start of function x= 10 y= 20"  
## [1] "At end of function x= 200"
```

- **Q:** After running the above code, what is the value of `first`? Of `x`?

Environments

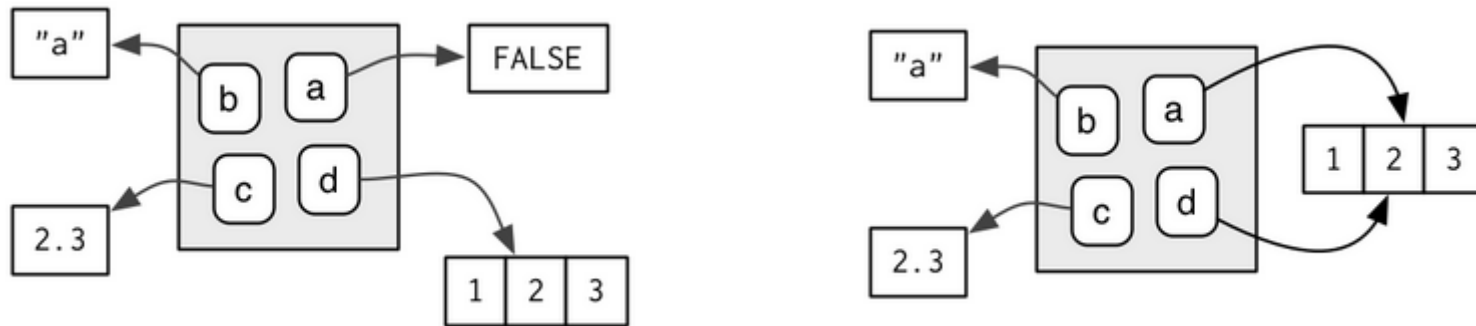
Environments are like address books

- An **environment** associates names with values



Environments are like address books

- An **environment** associates names with values








- Multiple names can point to the same values

What's in your environment?

```
ls()
```

```
## [1] "eda"           "first"          "func"
## [4] "my.mean"       "new.object"     "print.and.multiply"
## [7] "second"        "x"
```

Environment	History	Connections	Git	
<div><div></div><div> Import Dataset ▾</div><div></div></div>				
<div> Global Environment ▾</div>				
Values				
first	10			
new.object	200			
second	20			
x	int [1:25] 1 2 3 4 5 6 7 8 9 10 ...			
Functions				
func	function ()			

Three types we care about

- Base environment
 - Contains packages like `base`, `utils`, `stats`, `graphics`

Three types we care about

- Base environment
 - Contains packages like `base`, `utils`, `stats`, `graphics`
- Global environment
 - Additional variables you have created or packages you have loaded
 - Should "reset" every time you reopen R

Three types we care about

- Base environment
 - Contains packages like `base`, `utils`, `stats`, `graphics`
- Global environment
 - Additional variables you have created or packages you have loaded
 - Should "reset" every time you reopen R
- Current environment
 - Environment inside of a function

Scoping

Name Masking

- If a name isn't defined in the environment, `R` looks up one level

Name Masking

- If a name isn't defined in the environment, `R` looks up one level

```
pi
```

```
## [1] 3.141593
```

Name Masking

- If a name isn't defined in the environment, `R` looks up one level

```
pi
```

```
## [1] 3.141593
```

```
pi ← 1
```

```
pi
```

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

Name Masking

- If a name isn't defined in the environment, R looks up one level

```
pi
```

```
## [1] 3.141593
```

```
pi ← 1
```

```
pi
```

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



Functions vs. variables

- Finding functions works the same way as finding variables

Functions vs. variables

- Finding functions works the same way as finding variables
- You *can* have both `x ← 1:10` and `x ← function(x){x + 10}` ...

Functions vs. variables

- Finding functions works the same way as finding variables
- You *can* have both `x ← 1:10` and `x ← function(x){x + 10}` ...
 - but please don't.

Functions vs. variables

- Finding functions works the same way as finding variables
- You *can* have both `x ← 1:10` and `x ← function(x){x + 10}` ...
 - but please don't.
- Multiple functions can have the same name
 - R will default to the version in the most recently loaded package
 - order of search can be seen using `search()`

A fresh start

- each use of a function is independent of any previous uses

A fresh start

- each use of a function is independent of any previous uses
- a function's environment is wiped clean for each new use

Dynamic lookup

- Lookup happens when code is executed

Dynamic lookup

- Lookup happens when code is executed
- It doesn't matter what the value was when the code was created

```
x ← 15  
f ← function(x){x+1}  
f(x)
```

```
## [1] 16
```

```
x ← 20  
f(x)
```

```
## [1] 21
```

Software Design

Strategies for designing code

- Always a good idea to consider design before implementation
 - strategies include top-down (rigid) and bottom-up (iterative) approaches

Strategies for designing code

- Always a good idea to consider design before implementation
 - strategies include top-down (rigid) and bottom-up (iterative) approaches
- Visual aids like flowcharts can be used for planning and documentation

Strategies for designing code

- Always a good idea to consider design before implementation
 - strategies include top-down (rigid) and bottom-up (iterative) approaches
- Visual aids like flowcharts can be used for planning and documentation
- Outlines or pseudocode are helpful for breaking a big task into manageable bits

Coding Conventions

Conventions and style

- Encompasses everything from file names to spacing around operators

Conventions and style

- Encompasses everything from file names to spacing around operators
- Increases readability

Conventions and style

- Encompasses everything from file names to spacing around operators
- Increases readability
- As with the Oxford comma, people have strong opinions

Conventions and style

- Encompasses everything from file names to spacing around operators
- Increases readability
- As with the Oxford comma, people have strong opinions
- Consistency is most important

Conventions and style

- The things we care about* in this course are:

*HINT: these are things we will be looking for in your assignments!

Conventions and style

- The things we care about* in this course are:
 - Use `←` for assignment, `=` for function arguments, `=` for Boolean

*HINT: these are things we will be looking for in your assignments!

Conventions and style

- The things we care about* in this course are:
 - Use `←` for assignment, `=` for function arguments, `=` for Boolean
 - Start all functions with comments listing purpose, inputs, outputs

*HINT: these are things we will be looking for in your assignments!

Conventions and style

- The things we care about* in this course are:
 - Use `←` for assignment, `=` for function arguments, `==` for Boolean
 - Start all functions with comments listing purpose, inputs, outputs
 - Use indentation and spacing to make code 2-D

*HINT: these are things we will be looking for in your assignments!

Conventions and style

- The things we care about* in this course are:
 - Use `←` for assignment, `=` for function arguments, `==` for Boolean
 - Start all functions with comments listing purpose, inputs, outputs
 - Use indentation and spacing to make code 2-D
 - There should be spaces around operators and after commas

*HINT: these are things we will be looking for in your assignments!

Conventions and style

- The things we care about* in this course are:
 - Use `←` for assignment, `=` for function arguments, `=` for Boolean
 - Start all functions with comments listing purpose, inputs, outputs
 - Use indentation and spacing to make code 2-D
 - There should be spaces around operators and after commas
 - Lines should be <80 characters long

*HINT: these are things we will be looking for in your assignments!

Conventions and style

- The things we care about* in this course are:
 - Use `←` for assignment, `=` for function arguments, `==` for Boolean
 - Start all functions with comments listing purpose, inputs, outputs
 - Use indentation and spacing to make code 2-D
 - There should be spaces around operators and after commas
 - Lines should be <80 characters long
 - Use meaningful object names in a consistent style

*HINT: these are things we will be looking for in your assignments!

Conventions and style

- Style guides are available at
 - <https://google.github.io/styleguide/Rguide.xml>
 - <http://style.tidyverse.org/> (inc. packages to restyle code)