How to make an R script robust

Matthew Suderman

Lecturer in Epigenetic Epidemiology

MRC Integrative Epidemiology Unit

University of BRISTOL

Robust?

"tailored for large puppies with a robust physique"

"human male Asian robust skull displays extreme male traits"

"strong, deep maple flavoured syrup is a firm favourite"

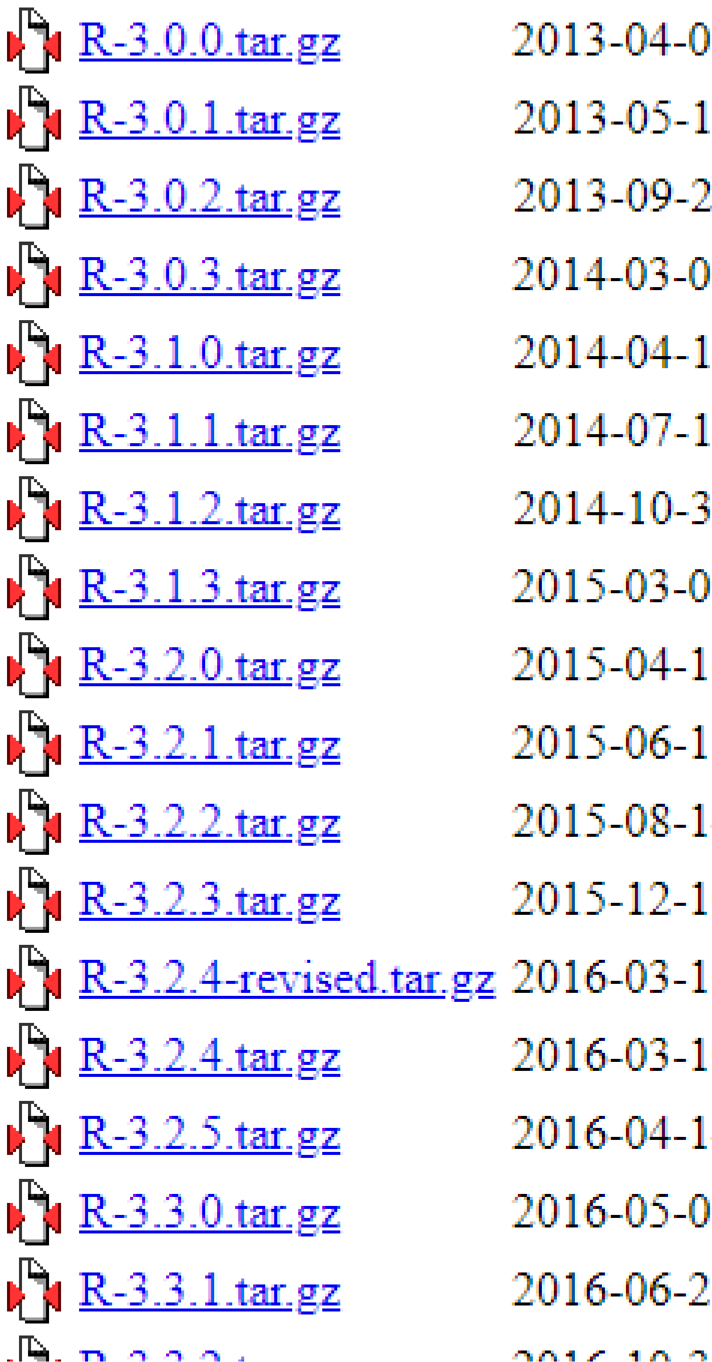
Robust scripts

“robustness is the ability of a computer system to cope with errors during execution and cope with erroneous input.” https://en.wikipedia.org/wiki/Robustness_(computer_science)

Challenges

Challenges

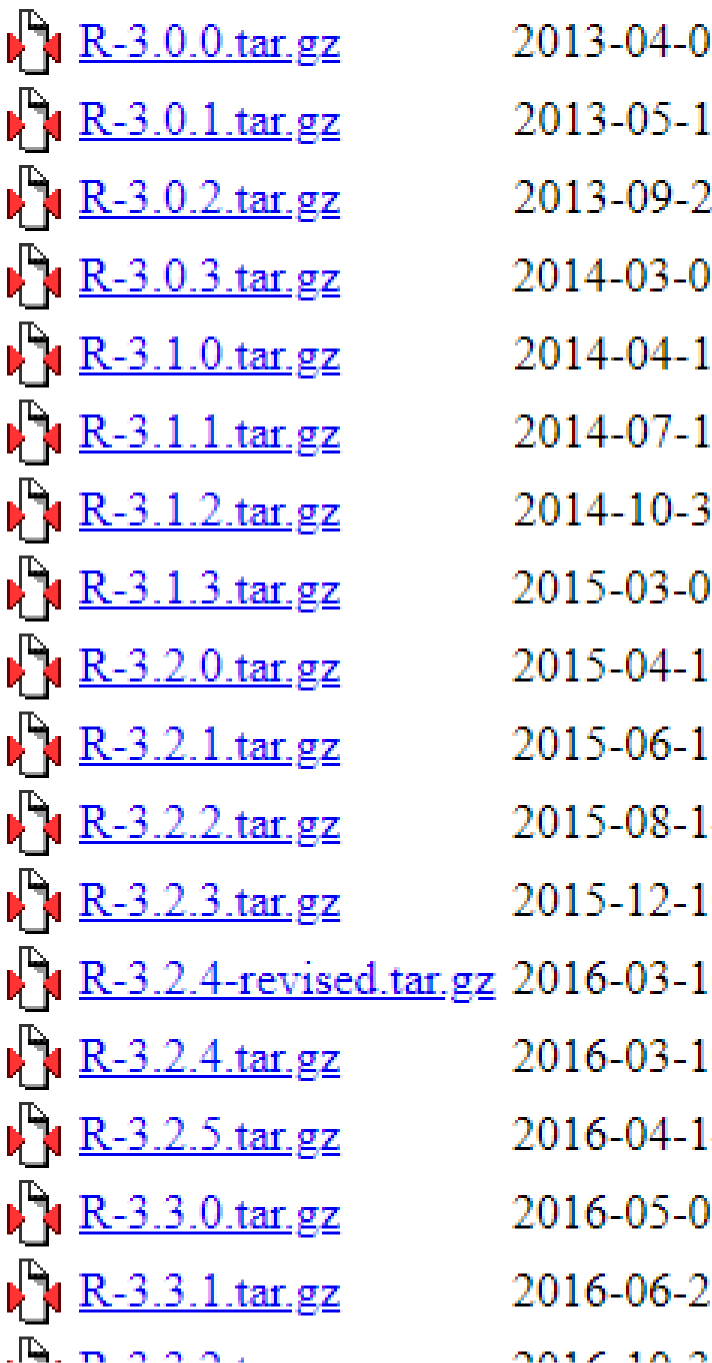
- Versions of R



R-3.0.0.tar.gz	2013-04-0
R-3.0.1.tar.gz	2013-05-1
R-3.0.2.tar.gz	2013-09-2
R-3.0.3.tar.gz	2014-03-0
R-3.1.0.tar.gz	2014-04-1
R-3.1.1.tar.gz	2014-07-1
R-3.1.2.tar.gz	2014-10-3
R-3.1.3.tar.gz	2015-03-0
R-3.2.0.tar.gz	2015-04-1
R-3.2.1.tar.gz	2015-06-1
R-3.2.2.tar.gz	2015-08-1
R-3.2.3.tar.gz	2015-12-1
R-3.2.4-revised.tar.gz	2016-03-1
R-3.2.4.tar.gz	2016-03-1
R-3.2.5.tar.gz	2016-04-1
R-3.3.0.tar.gz	2016-05-0
R-3.3.1.tar.gz	2016-06-2

Challenges

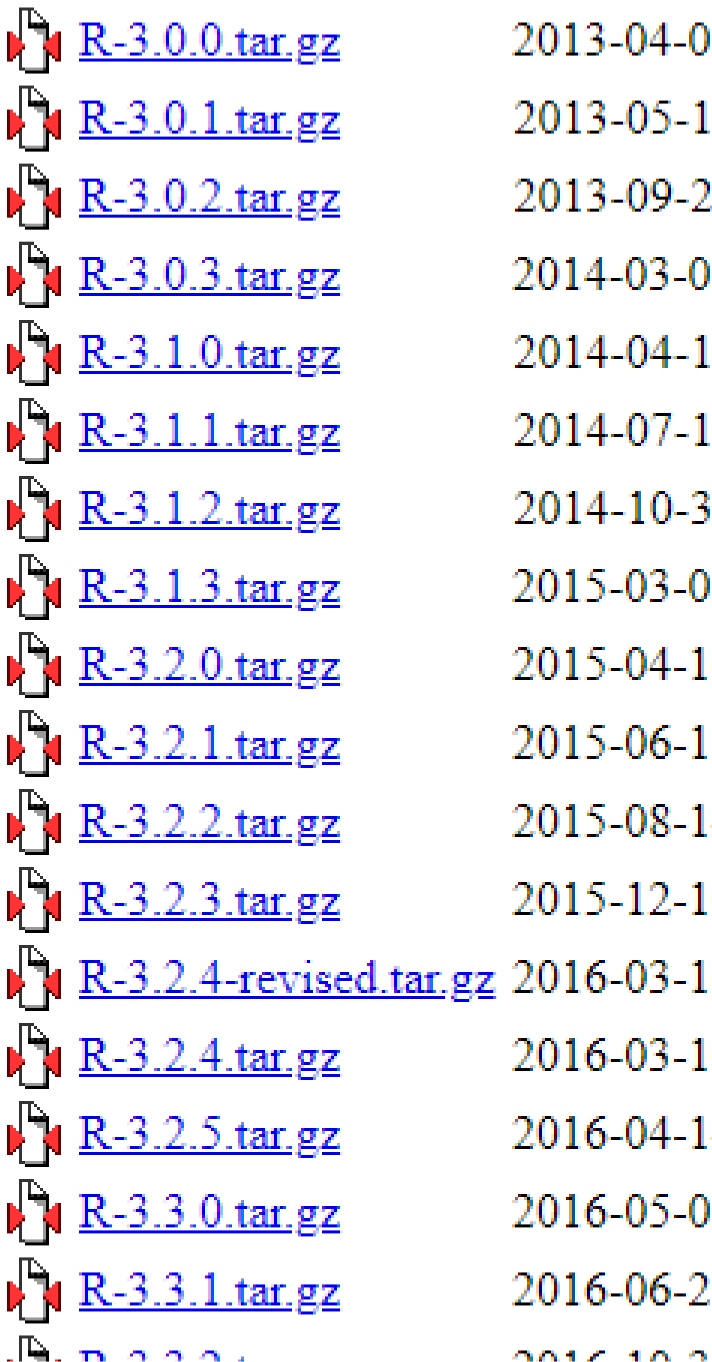
- Versions of R
- Versions of R packages



R-3.0.0.tar.gz	2013-04-0
R-3.0.1.tar.gz	2013-05-1
R-3.0.2.tar.gz	2013-09-2
R-3.0.3.tar.gz	2014-03-0
R-3.1.0.tar.gz	2014-04-1
R-3.1.1.tar.gz	2014-07-1
R-3.1.2.tar.gz	2014-10-3
R-3.1.3.tar.gz	2015-03-0
R-3.2.0.tar.gz	2015-04-1
R-3.2.1.tar.gz	2015-06-1
R-3.2.2.tar.gz	2015-08-1
R-3.2.3.tar.gz	2015-12-1
R-3.2.4-revised.tar.gz	2016-03-1
R-3.2.4.tar.gz	2016-03-1
R-3.2.5.tar.gz	2016-04-1
R-3.3.0.tar.gz	2016-05-0
R-3.3.1.tar.gz	2016-06-2
[illegible]	[illegible]

Challenges

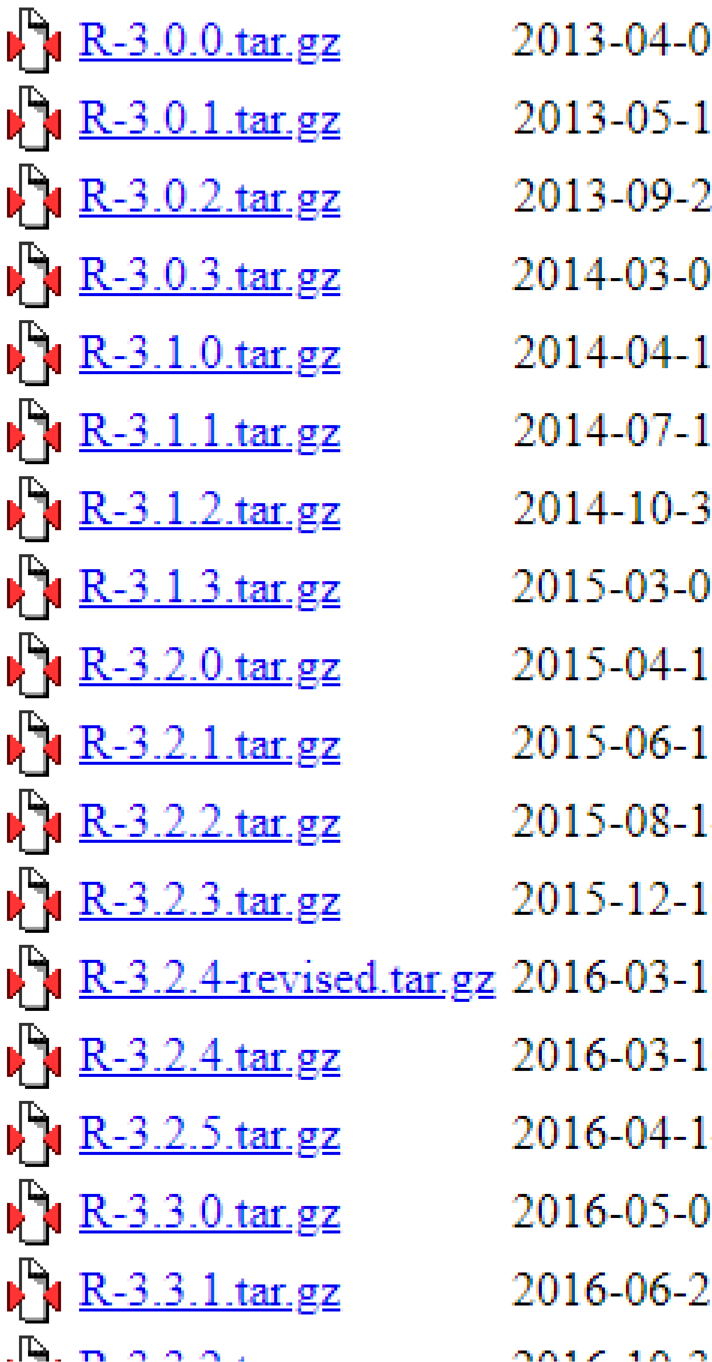
- Versions of R
- Versions of R packages
- Operating systems



R-3.0.0.tar.gz	2013-04-0
R-3.0.1.tar.gz	2013-05-1
R-3.0.2.tar.gz	2013-09-2
R-3.0.3.tar.gz	2014-03-0
R-3.1.0.tar.gz	2014-04-1
R-3.1.1.tar.gz	2014-07-1
R-3.1.2.tar.gz	2014-10-3
R-3.1.3.tar.gz	2015-03-0
R-3.2.0.tar.gz	2015-04-1
R-3.2.1.tar.gz	2015-06-1
R-3.2.2.tar.gz	2015-08-1
R-3.2.3.tar.gz	2015-12-1
R-3.2.4-revised.tar.gz	2016-03-1
R-3.2.4.tar.gz	2016-03-1
R-3.2.5.tar.gz	2016-04-1
R-3.3.0.tar.gz	2016-05-0
R-3.3.1.tar.gz	2016-06-2

Challenges

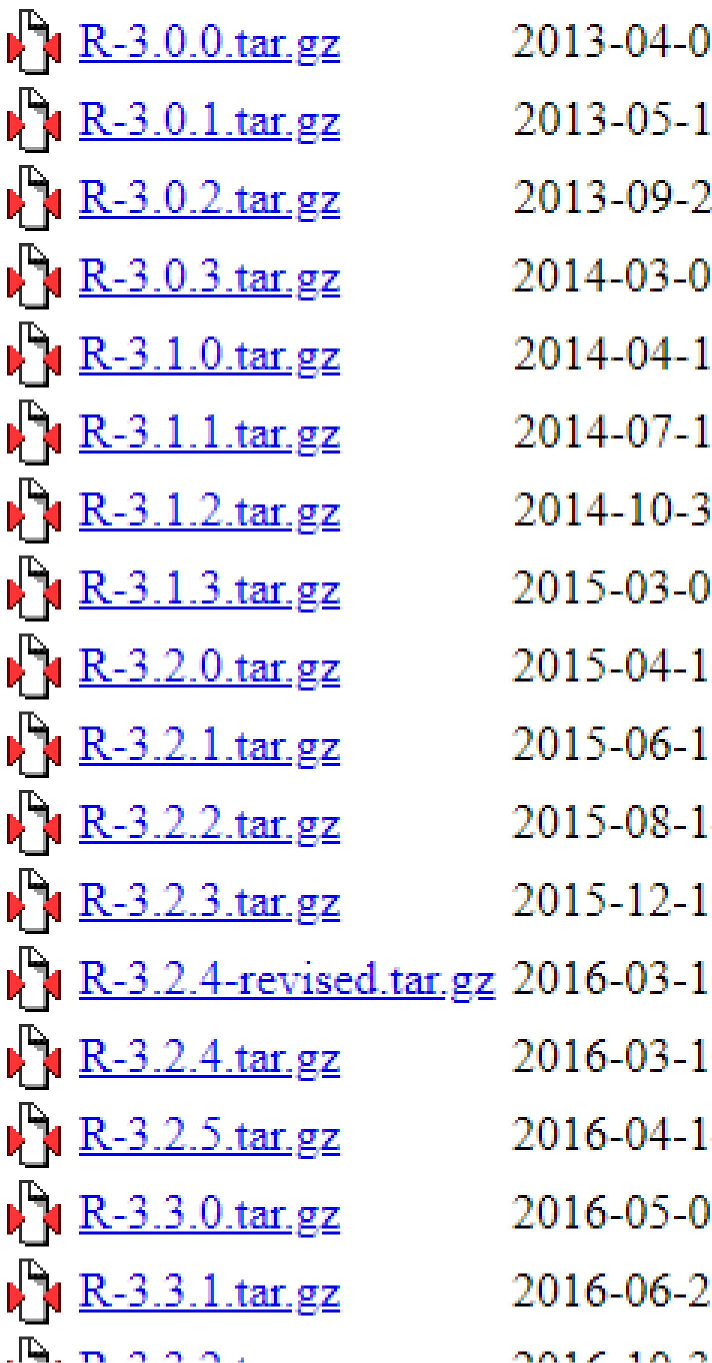
- Versions of R
- Versions of R packages
- Operating systems
- Changing requirements



R-3.0.0.tar.gz	2013-04-0
R-3.0.1.tar.gz	2013-05-1
R-3.0.2.tar.gz	2013-09-2
R-3.0.3.tar.gz	2014-03-0
R-3.1.0.tar.gz	2014-04-1
R-3.1.1.tar.gz	2014-07-1
R-3.1.2.tar.gz	2014-10-3
R-3.1.3.tar.gz	2015-03-0
R-3.2.0.tar.gz	2015-04-1
R-3.2.1.tar.gz	2015-06-1
R-3.2.2.tar.gz	2015-08-1
R-3.2.3.tar.gz	2015-12-1
R-3.2.4-revised.tar.gz	2016-03-1
R-3.2.4.tar.gz	2016-03-1
R-3.2.5.tar.gz	2016-04-1
R-3.3.0.tar.gz	2016-05-0
R-3.3.1.tar.gz	2016-06-2

Challenges

- Versions of R
- Versions of R packages
- Operating systems
- Changing requirements
- Code Reuse



R-3.0.0.tar.gz	2013-04-0
R-3.0.1.tar.gz	2013-05-1
R-3.0.2.tar.gz	2013-09-2
R-3.0.3.tar.gz	2014-03-0
R-3.1.0.tar.gz	2014-04-1
R-3.1.1.tar.gz	2014-07-1
R-3.1.2.tar.gz	2014-10-3
R-3.1.3.tar.gz	2015-03-0
R-3.2.0.tar.gz	2015-04-1
R-3.2.1.tar.gz	2015-06-1
R-3.2.2.tar.gz	2015-08-1
R-3.2.3.tar.gz	2015-12-1
R-3.2.4-revised.tar.gz	2016-03-1
R-3.2.4.tar.gz	2016-03-1
R-3.2.5.tar.gz	2016-04-1
R-3.3.0.tar.gz	2016-05-0
R-3.3.1.tar.gz	2016-06-2

Solution: comments

Commenting in R is very simple.

```r
x <- 3 # all text after the
       # hash symbol is not code
```

Deciding *what* to comment is more tricky.

Roxygen2 provides a useful framework for documenting functions that could be applied to scripts as well.

Solution: comments

Commenting in R is very simple.

```
x <- 3 # all text after the
       # hash symbol is not code
```

Deciding *what* to comment is more tricky.

Roxygen2 provides a useful framework for documenting functions that could be applied to scripts as well.

```
#' My very cool scatterplot
#'
#' @param x x-coordinates of points
#'          (numeric vector)
#' @param y y-coordinates of points
#'          (numeric vector)
#' @param line whether to plot the re
#'             line (logical) (Defaul
#' @return Linear model fit for y~x.
#' @examples
#' x <- c(1,2,3,3)
#' y <- c(2,2,4,5)
#' fit <- myscatterplot(x,y,line=F)
#'
myscatterplot <- function(x,y,line=T)
  plot(x,y,pch=19)
  fit <- lm(y~x)
  if (line)
    abline(fit, col="red", lty="dashe
  return(fit)
}
```

Solution: keeping it short

- Break up long lines

```r
dat.avg <- sapply(by(dat, as.factor(probes$symbol), colMeans), identity)
```

Solution: keeping it short

- Break up long lines

```r
dat.avg <- sapply(by(dat, as.factor(probes$symbol), colMeans), identity)
```

vs

```r
symbols <- as.factor(probes$symbol)
dat.list <- by(dat, symbols, colMeans)
dat.avg <- sapply(dat.list, identity)
```

Solution: keeping it short

- Break up long lines

```
dat.avg <- sapply(by(dat, as.factor(probes$symbol), colMeans), identity)
```

vs

```
symbols <- as.factor(probes$symbol)
dat.list <- by(dat, symbols, colMeans)
dat.avg <- sapply(dat.list, identity)
```

- Make functions/scripts short

Solution: keeping it short

- Break up long lines

```
dat.avg <- sapply(by(dat, as.factor(probes$symbol), colMeans), identity)
```

vs

```
symbols <- as.factor(probes$symbol)
dat.list <- by(dat, symbols, colMeans)
dat.avg <- sapply(dat.list, identity)
```

- Make functions/scripts short

A useful **rule**: the entire script or function should fit on one screen.

Break up long bits of code into functions.

Solution: assertions

It is useful to test assumptions about user input and the values of variables throughout the script.

```r
## ... lots of code

if (!is.numeric(x))
  stop("'x' is not numeric")
if (x < 50)
  stop("'x' is too small")

## ... lots of code
```

R provides a shorthand for this using the `stopifnot()` function.

```r
stopifnot(is.numeric(x) && x >= 50)
```

Solution: assertions

It is useful to test assumptions about user input and the values of variables throughout the script.

```
## ... lots of code

if (!is.numeric(x))
  stop("'x' is not numeric")
if (x < 50)
  stop("'x' is too small")

## ... lots of code
```

R provides a shorthand for this using the `stopifnot()` function.

```
stopifnot(is.numeric(x) && x >= 50)
```

Assertions have 3 benefits:

1. They **catch** errors before they generate mysterious outputs.
2. They force the script writer to **think** more concretely about the values the variables could take.
3. They **document** the script.

Solution: modular development

Split up code as much as possible into functions and possibly even packages.

Solution: modular development

Split up code as much as possible into functions and possibly even packages.

The following code simulates some data and generates two plots.

```
## simulation 1
x1 <- rnorm(100)
y1 <- x1 + rnorm(length(x1))
plot(x1, y1, pch=19)
abline(lm(y1~x1),
       col="red",
       lty="dashed")
##
## simulation 2
x2 <- rnorm(100)
y2 <- x1 + x2 + rnorm(length(x1))
plot(x2, y2, pch=19)
abline(lm(y2~x2),
       col="red",
       lty="dashed")
```

Solution: modular development

Split up code as much as possible into functions and possibly even packages.

The following code simulates some data and generates two plots.

```
## simulation 1
x1 <- rnorm(100)
y1 <- x1 + rnorm(length(x1))
plot(x1, y1, pch=19)
abline(lm(y1~x1),
       col="red",
       lty="dashed")
##
## simulation 2
x2 <- rnorm(100)
y2 <- x1 + x2 + rnorm(length(x1))
plot(x2, y2, pch=19)
abline(lm(y2~x2),
       col="red",
       lty="dashed")
```

The following version is easier to **read**, **modify** and **reuse**.

```
## data simulation
x1 <- rnorm(100)
y1 <- x1 + rnorm(length(x1))
x2 <- rnorm(100)
y2 <- x1 + x2 + rnorm(length(x1))

## scatterplots
myscatter <- function(x,y) {
  plot(x1, y1, pch=19)
  fit <- lm(y~x)
  abline(fit,
         col="red", lty="dashed")
  return(fit)
}
myscatter(x1,y1)
myscatter(x2,y2)
```

Solution: exceptions

The following script will stop when attempting to calculate `r` just before printing the message at the end.

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

## ... lots of code

r <- cor(x,y)
cat("The correlation is", r, "\n")
```

Solution: exceptions

The following script will stop when attempting to calculate `r` just before printing the message at the end.

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

## ... lots of code

r <- cor(x,y)
cat("The correlation is", r, "\n")
```

The following script uses the `tryCatch` function to "catch" the error and set `r` to `NaN` ("Not A Number"). As a result, this script will run all the way to the end.

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

## ... lots of code

r <- tryCatch(cor(x,y),
              error=function(e) NaN)
cat("The correlation is", r, "\n")
```

Solution: debugging

Debugging is identifying errors in the code and fixing them.

If your script or R command has been stopped by an error, the simplest thing to do is to type `trackeback()`.

```r
traceback()
```

Solution: debugging

Debugging is identifying errors in the code and fixing them.

If your script or R command has been stopped by an error, the simplest thing to do is to type `trackeback()`.

```
traceback()
```

Traceback tells you the specific line of code **where** the error was generated.

This may be helpful if the error occured when your script was running a function from an R package.

Solution: debugging, cont

If this doesn't help, you can insert print statements just before the error to check variable values.

For example,

```
## ... lots of code
print("The value of x:")
print(x)
print("the value of y:")
print(y)
## the script stops here
## ... lots of code
```

Solution: debugging, cont

A more advanced alternative is to use the `browser` function.

```
## ... lots of code
browser()
## the script stops here
## ... lots of code
```

Solution: debugging, cont

A more advanced alternative is to use the `browser` function.

```
## ... lots of code
browser()
## the script stops here
## ... lots of code
```

When a script reaches a call to `browser()`, it:

1. pauses the script
2. allows you to run R commands to look at the values of variables
3. allows you to run the rest of the script, pausing at each line