loops and functions in R

Berry Boessenkool, berry-b@gmx.de
Jannes Breier, jbreier@gfz-potsdam.de

SWC lesson material

These slides and tasks are a subset of Berry's teaching material at
github.com/brry/course

R installation instructions: github.com/brry/course#install

PDF created on 2019-11-27, 07:16

Outline

print("Hello world!")

- Berry Boessenkool → Geoecology @ Potsdam University

`print("Hello world!")`

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fan

`print("Hello world!")`

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fanatic

`print("Hello world!")`

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fanatic since 2010

`print("Hello world!")`

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fanatic since 2010
- Developer of `rdwd`,

`print("Hello world!")`

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fanatic since 2010
- Developer of `rdwd`, Freelance trainer & consultant 

`print("Hello world!")`

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fanatic since 2010
- Developer of `rdwd`, Freelance trainer & consultant 

- Jannes Breier → Geoecology @ Potsdam University

print("Hello world!")

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fanatic since 2010
- Developer of `rdwd`, Freelance trainer & consultant 

- Jannes Breier → Geoecology @ Potsdam University
- Berry taught me R in 2013 ☺

`print("Hello world!")`

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fanatic since 2010
- Developer of `rdwd`, Freelance trainer & consultant 

- Jannes Breier → Geoecology @ Potsdam University
- Berry taught me R in 2013 ☺
- Research Software Engineer at GFZ, GFZ Helmholtz-Zentrum Potsdam Sec.4.4: Hydrology

print("Hello world!")

- Berry Boessenkool → Geoecology @ Potsdam University
- R Fanatic since 2010
- Developer of `rdwd`, Freelance trainer & consultant 

- Jannes Breier → Geoecology @ Potsdam University
- Berry taught me R in 2013 ☺
- Research Software Engineer at GFZ, GFZ Helmholtz-Zentrum Potsdam Sec.4.4: Hydrology

- If we're proceeding too fast, please interrupt!

Outline

For loops

Execute a block of code several times, with different input values.

Syntax: `for(aRunningVariable in aSequence){ doSomething }`

For loops

Execute a block of code several times, with different input values.
Syntax: `for(aRunningVariable in aSequence){ doSomething }`
Often, i (for index) is used, thus `for(i in 1:n) doThis(i)`

For loops

Execute a block of code several times, with different input values.
Syntax: `for(aRunningVariable in aSequence){ doSomething }`
Often, i (for index) is used, thus `for(i in 1:n) doThis(i)`

```
help("for") # needs quotation marks!
```

For loops

Execute a block of code several times, with different input values.
Syntax: `for(aRunningVariable in aSequence){ doSomething }`
Often, i (for index) is used, thus `for(i in 1:n) doThis(i)`

```
help("for") # needs quotation marks!
```

```
print(1:2)
print(1:5)
print(1:9)
```

For loops

Execute a block of code several times, with different input values.
Syntax: `for(aRunningVariable in aSequence){ doSomething }`
Often, i (for index) is used, thus `for(i in 1:n) doThis(i)`

```
help("for") # needs quotation marks!
```

```
print(1:2)
print(1:5)
print(1:9)
```

This is easier and less prone to human errors with:

For loops

Execute a block of code several times, with different input values.
Syntax: `for(aRunningVariable in aSequence){ doSomething }`
Often, i (for index) is used, thus `for(i in 1:n) doThis(i)`

```
help("for") # needs quotation marks!
```

```
print(1:2)
print(1:5)
print(1:9)
```

This is easier and less prone to human errors with:

```
for(i in c(2,5,9) ) { print(1:i) }
```

```
## [1] 1 2
## [1] 1 2 3 4 5
## [1] 1 2 3 4 5 6 7 8 9
```

For loops: fill a vector

```
v <- vector(mode="numeric", length=20)
v

##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

For loops: fill a vector

```
v <- vector(mode="numeric", length=20)
v

##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
for(i in 3:17)  {  v[i] <- (i+2)^2  }
```

For loops: fill a vector

```
v <- vector(mode="numeric", length=20)
v

##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
for(i in 3:17)  {  v[i] <- (i+2)^2  }
```

```
v # this code was executed once for each i

##  [1]   0   0  25  36  49  64  81 100 121
## [10] 144 169 196 225 256 289 324 361   0
## [19]   0   0
```

For loops: fill a vector

```
v <- vector(mode="numeric", length=20)
v

##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

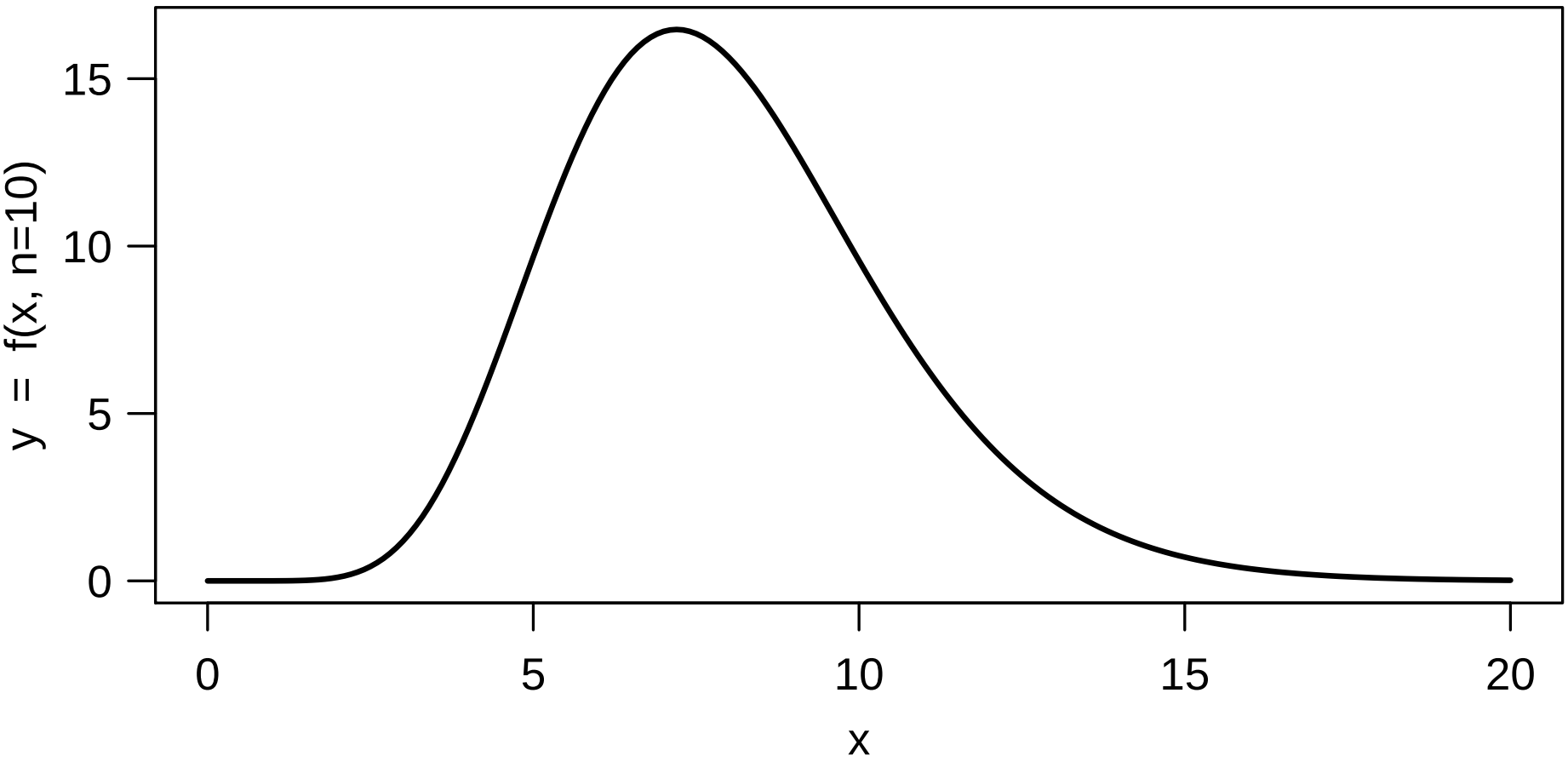
```
for(i in 3:17)  {  v[i] <- (i+2)^2  }
```

```
v # this code was executed once for each i

##  [1]   0   0  25  36  49  64  81 100 121
## [10] 144 169 196 225 256 289 324 361   0
## [19]   0   0
```

In R, `for` loops are slow. Always try to vectorize (the best option, not always possible) or use `lapply` (saves you the initiation of the empty vector, easier to parallize).

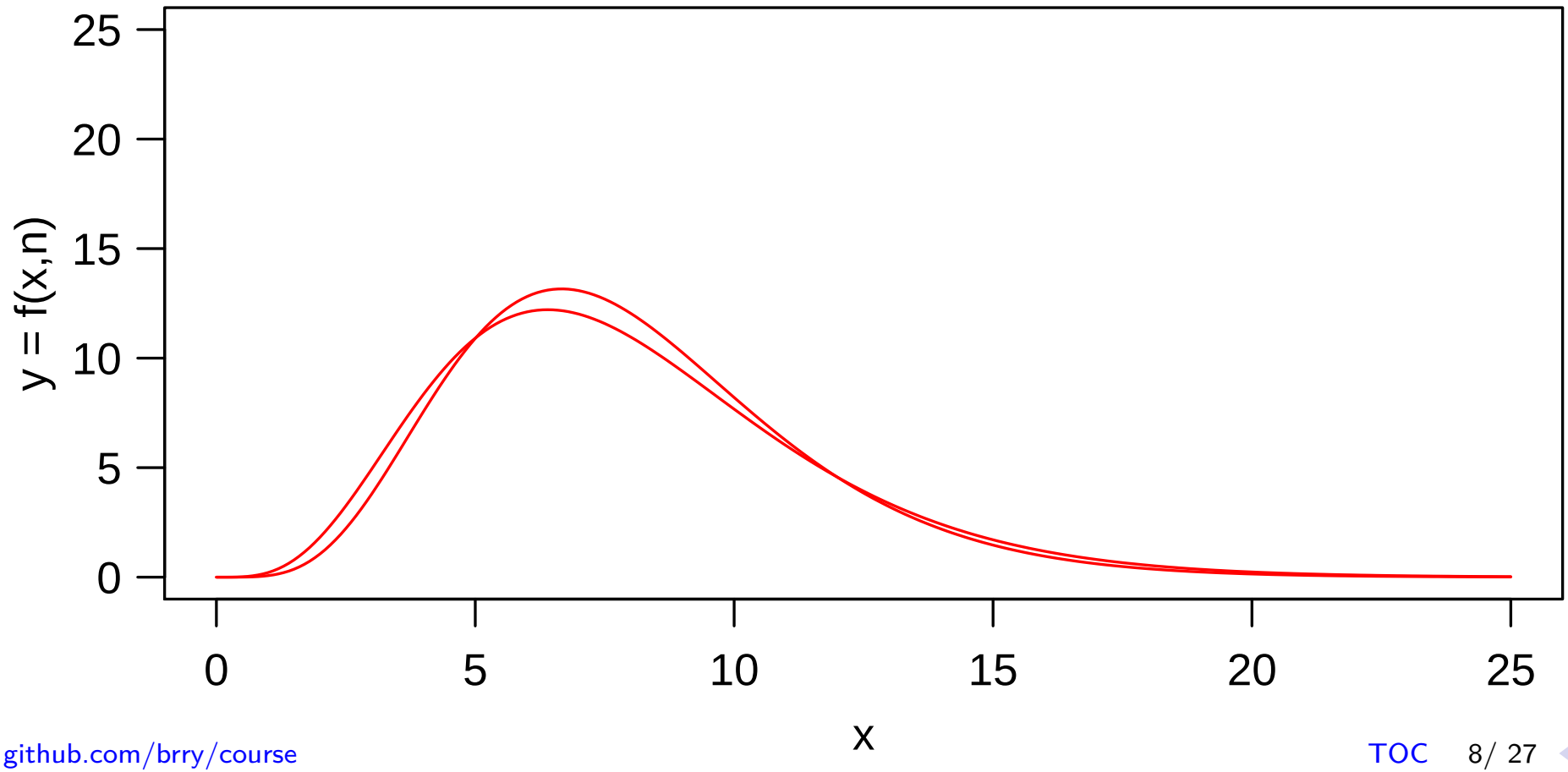
For loops: execute code multiple times I

$$y = f(x, n) = \frac{12.5 * n}{(n-1)!} * \left(\frac{nx}{8}\right)^{(n-1)} * e^{-\frac{nx}{8}}$$



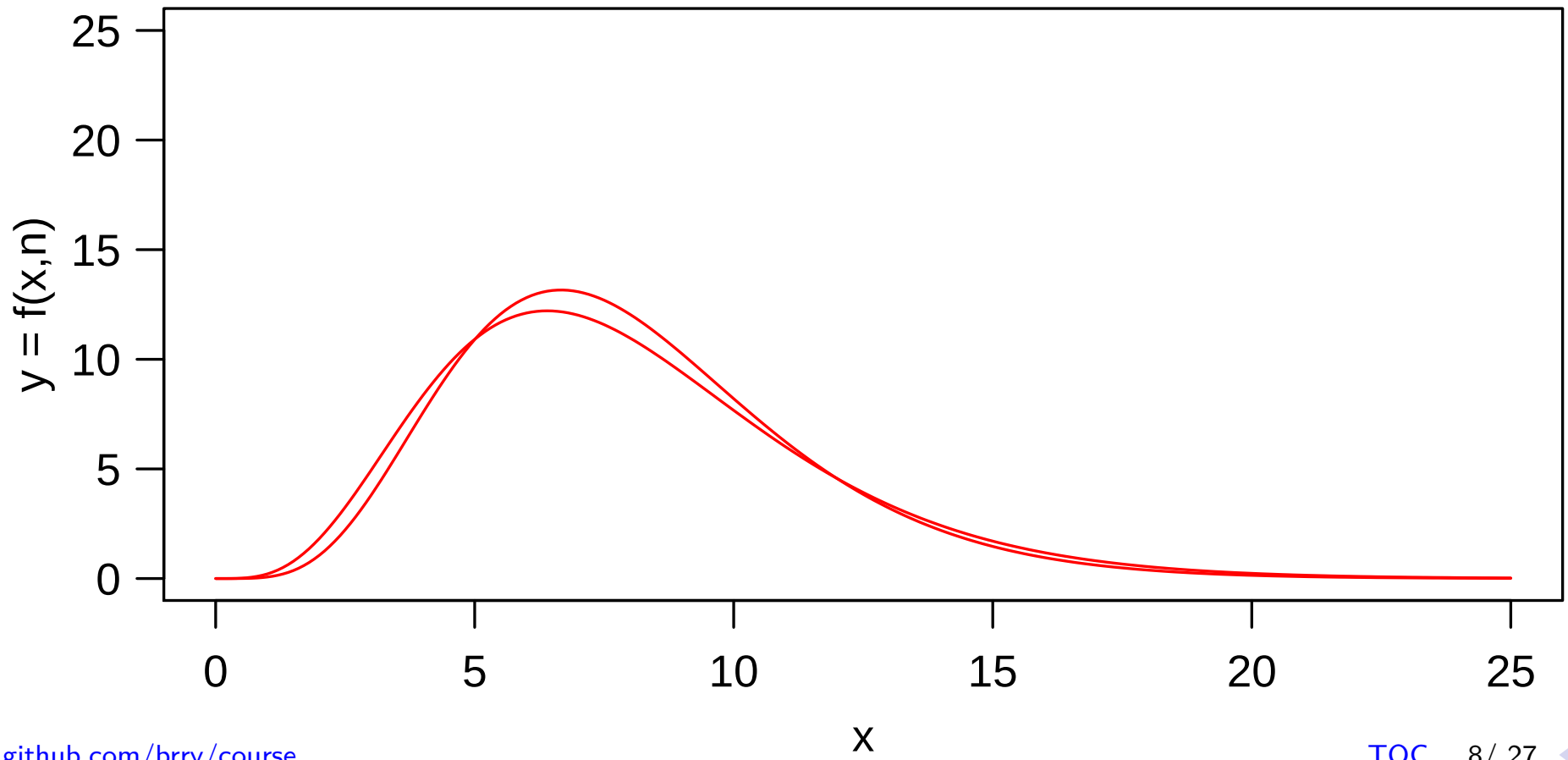
For loops: execute code multiple times II

```
x <- seq(0,25,0.1)
plot(x,x, type="n", ylab="y = f(x,n)")
lines(x, 12.5*5/factorial(5-1)*(x/8*5)^(5-1)*exp(-x/8*5), col=2)
lines(x, 12.5*6/factorial(6-1)*(x/8*6)^(6-1)*exp(-x/8*6), col=2)
```



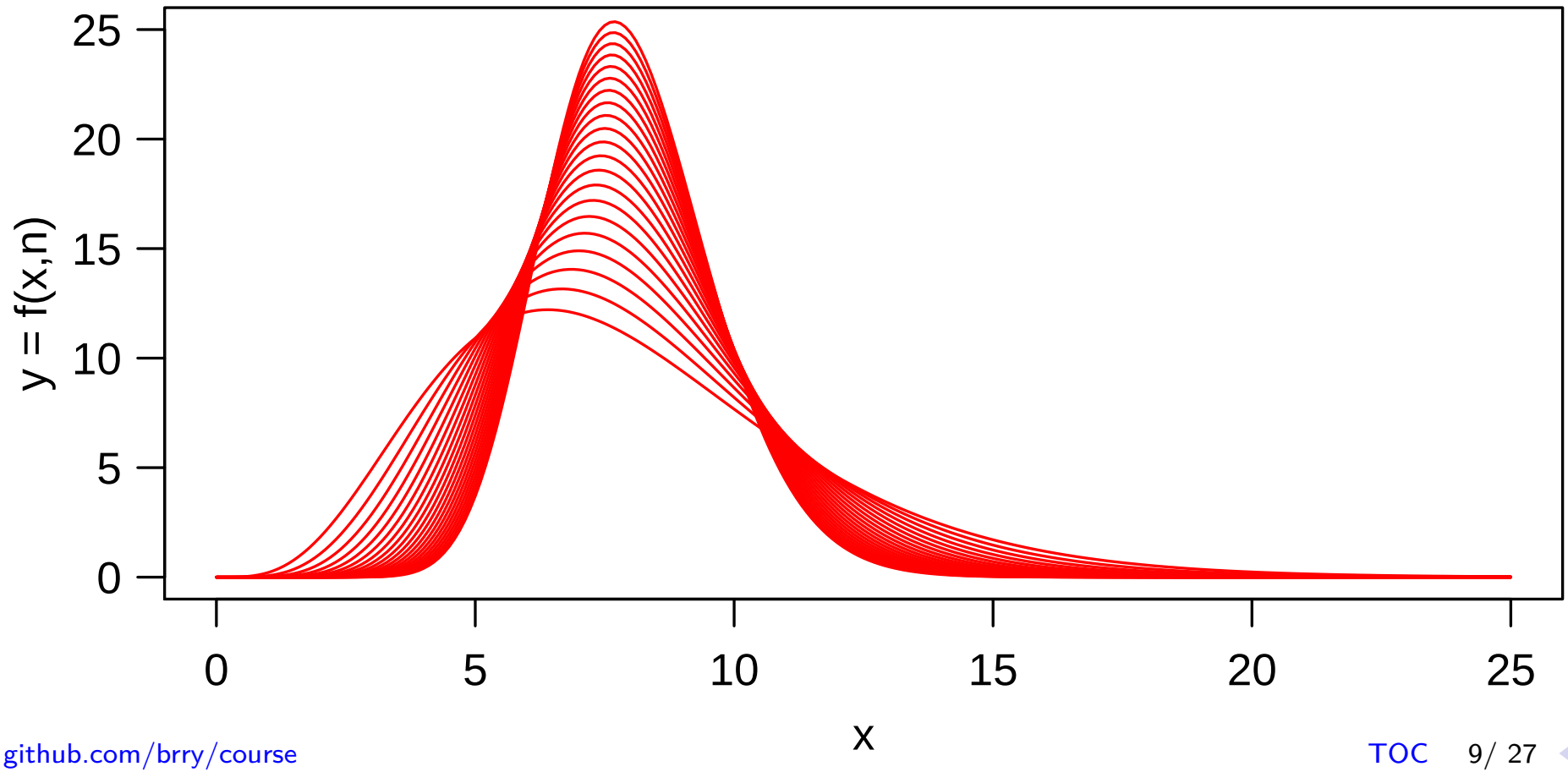
For loops: execute code multiple times II

```
x <- seq(0,25,0.1)
plot(x,x, type="n", ylab="y = f(x,n)")
lines(x, 12.5*5/factorial(5-1)*(x/8*5)^(5-1)*exp(-x/8*5), col=2)
lines(x, 12.5*6/factorial(6-1)*(x/8*6)^(6-1)*exp(-x/8*6), col=2)
```



For loops: execute code multiple times III

```
x <- seq(0,25,0.1)
plot(x,x, type="n", ylab="y = f(x,n)")
for (n in 5:25)
lines(x, 12.5*n/factorial(n-1)*(x/8*n)^(n-1)*exp(-x/8*n), col=2)
```



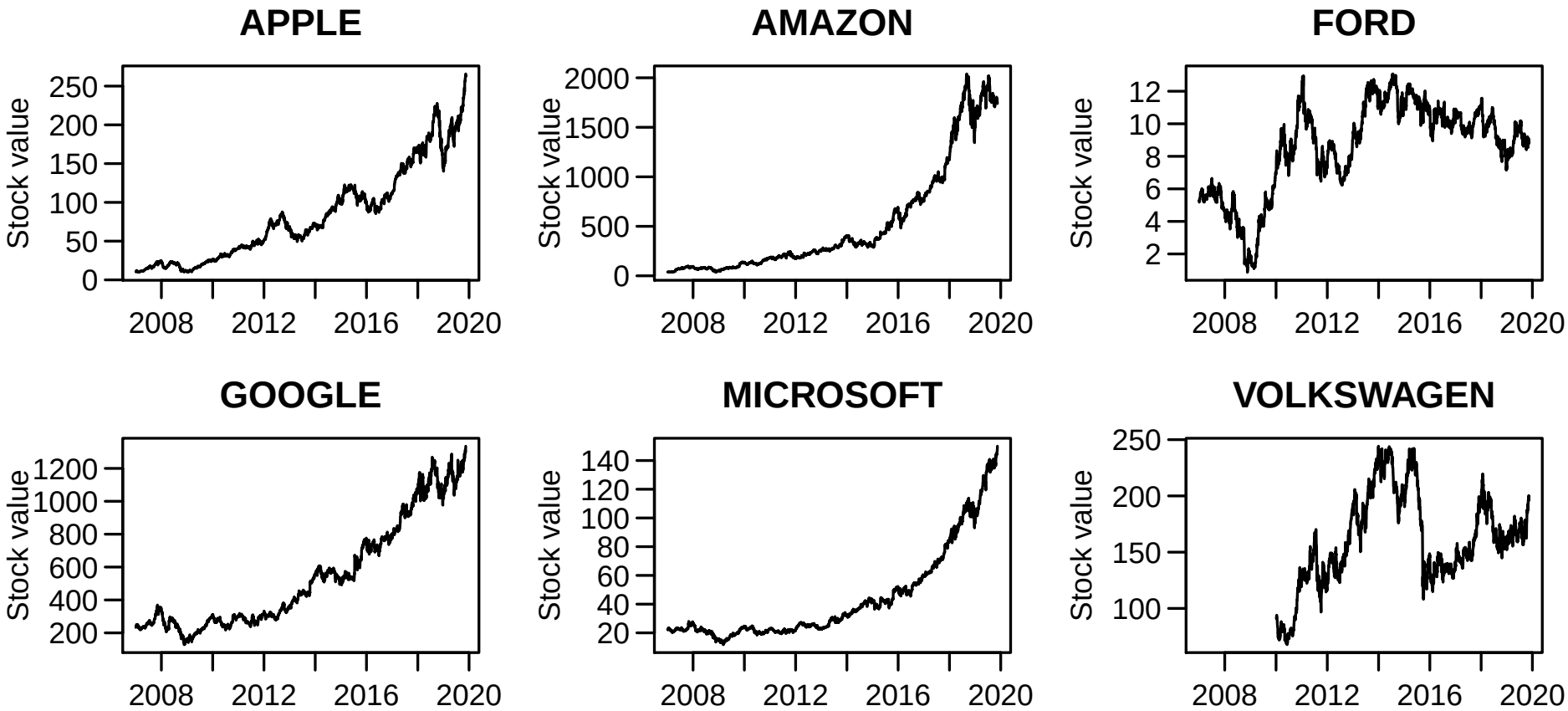
For loops exercise

Exercise 1: for loop

1. Read stocks.txt *(rightclick **Raw**, save as)*, so that there are no factors in the data.frame
2. Change the first column type from `char` to `date` with `?as.Date`
3. What do you get with `plot(stocks[ ,c(1,2)])`? Make it a line graph by setting the argument `type`.
4. With a `for` loop, plot each stock time series, i.e. plot the *i*th column over the first column.
5. BONUS 1: Use good annotations (`main, ylab, xlab`)
6. BONUS 2: Turn y axis labels upright (`las`)
7. BONUS 3: With `par(mfrow...`, set up a two by three panel plot

For loops exercise solution

```
stocks <- read.table("data/stocks.txt", header=T, stringsAsFactors=FALSE)
stocks$Date <- as.Date(stocks$Date)
for(i in 2:7) plot(stocks[ ,c(1,i)], type="l", main=colnames(stocks)[i],
                   ylab="Stock value", las=1)
```



for -> lapply

lapply: apply a function to elements of a list (or vector)
list: R object containing other objects

for -> lapply

lapply: apply a function to elements of a list (or vector)
list: R object containing other objects

```
files <- dir("../rawdata", pattern="*.csv", full=TRUE)
```

for -> lapply

lapply: apply a function to elements of a list (or vector)
list: R object containing other objects

```
files <- dir("../rawdata", pattern="*.csv", full=TRUE)
```

```
# bad and slow way:
ldfs <- list() # initiate empty list
for(i in 1:length(files))
   ldfs[[i]] <- read.csv(files[i], as.is=TRUE)
```

for -> lapply

lapply: apply a function to elements of a list (or vector)
list: R object containing other objects

```
files <- dir("../rawdata", pattern="*.csv", full=TRUE)
```

```
# bad and slow way:
ldfs <- list() # initiate empty list
for(i in 1:length(files))
   ldfs[[i]] <- read.csv(files[i], as.is=TRUE)
```

```
# much better way: apply function to each file
ldfs <- lapply(X=files, FUN=read.csv, as.is=TRUE)
```

for -> lapply

lapply: apply a function to elements of a list (or vector)
list: R object containing other objects

```
files <- dir("../rawdata", pattern="*.csv", full=TRUE)
```

```
# bad and slow way:
ldfs <- list() # initiate empty list
for(i in 1:length(files))
   ldfs[[i]] <- read.csv(files[i], as.is=TRUE)
```

```
# much better way: apply function to each file
ldfs <- lapply(X=files, FUN=read.csv, as.is=TRUE)
```

```
# progress bar with remaining time (+ parallelized!)
library("pbapply")
ldfs <- pblapply(X=files, FUN=read.csv, as.is=TRUE)
ldfs <- pblapply(X=files, FUN=read.csv, as.is=TRUE, cl=8)
```

Outline

Functions I

http://r4ds.had.co.nz/functions.html

Functions I

http://r4ds.had.co.nz/functions.html

```
?"function"
```

Functions I

http://r4ds.had.co.nz/functions.html

```
?"function"
```

Syntax:

```
Functionobjectname <- function(argument1, argument2, ...)
                                {"DoSomething"}
```

Functions I

http://r4ds.had.co.nz/functions.html

```
?"function"
```

Syntax:

```
Functionobjectname <- function(argument1, argument2, ...)
                                    {"DoSomething"}
```

```
myfunct <- function(grappig)
       {plot(grappig, type="l"); return(grappig*7)  }
```

Functions I

http://r4ds.had.co.nz/functions.html

```
?"function"
```

Syntax:

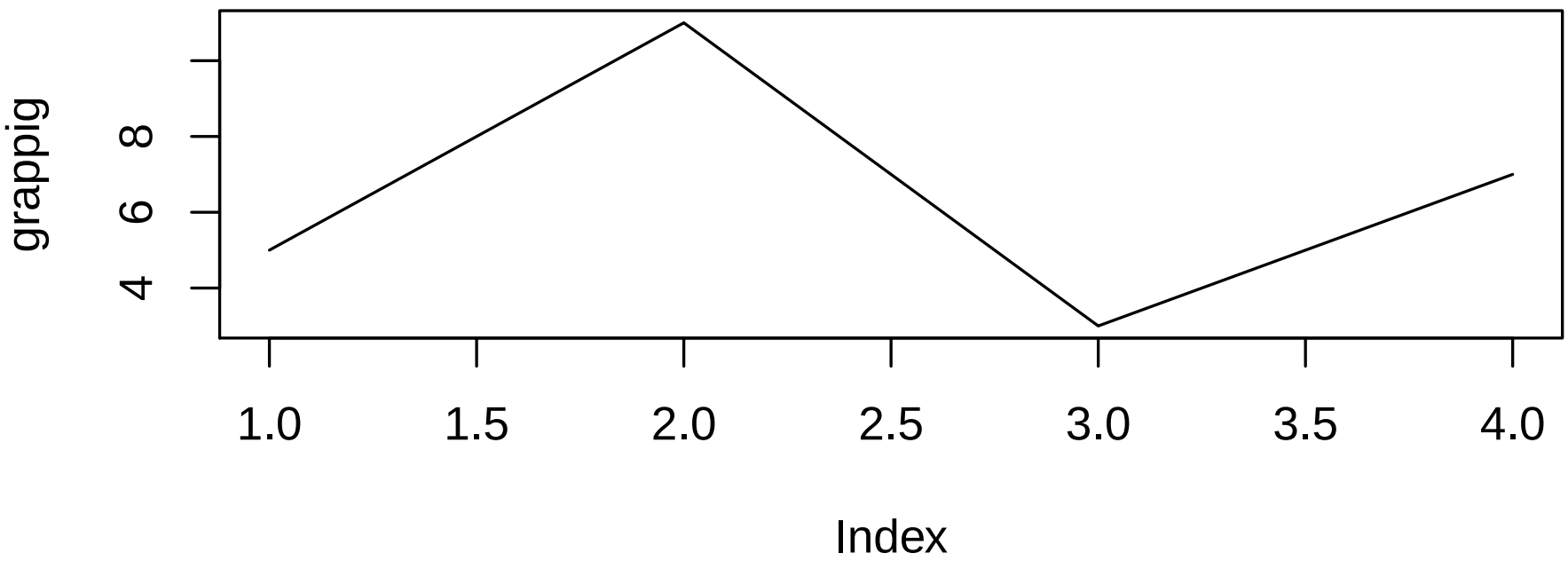
```
Functionobjectname <- function(argument1, argument2, ...)
                                    {"DoSomething"}
```

```
myfunct <- function(grappig)
       {plot(grappig, type="l"); return(grappig*7)  }
```

After `return()`ing, the execution of the function is terminated, so it should only be positioned at the end. It can also be left away, the last instruction ("expression") will then be returned.

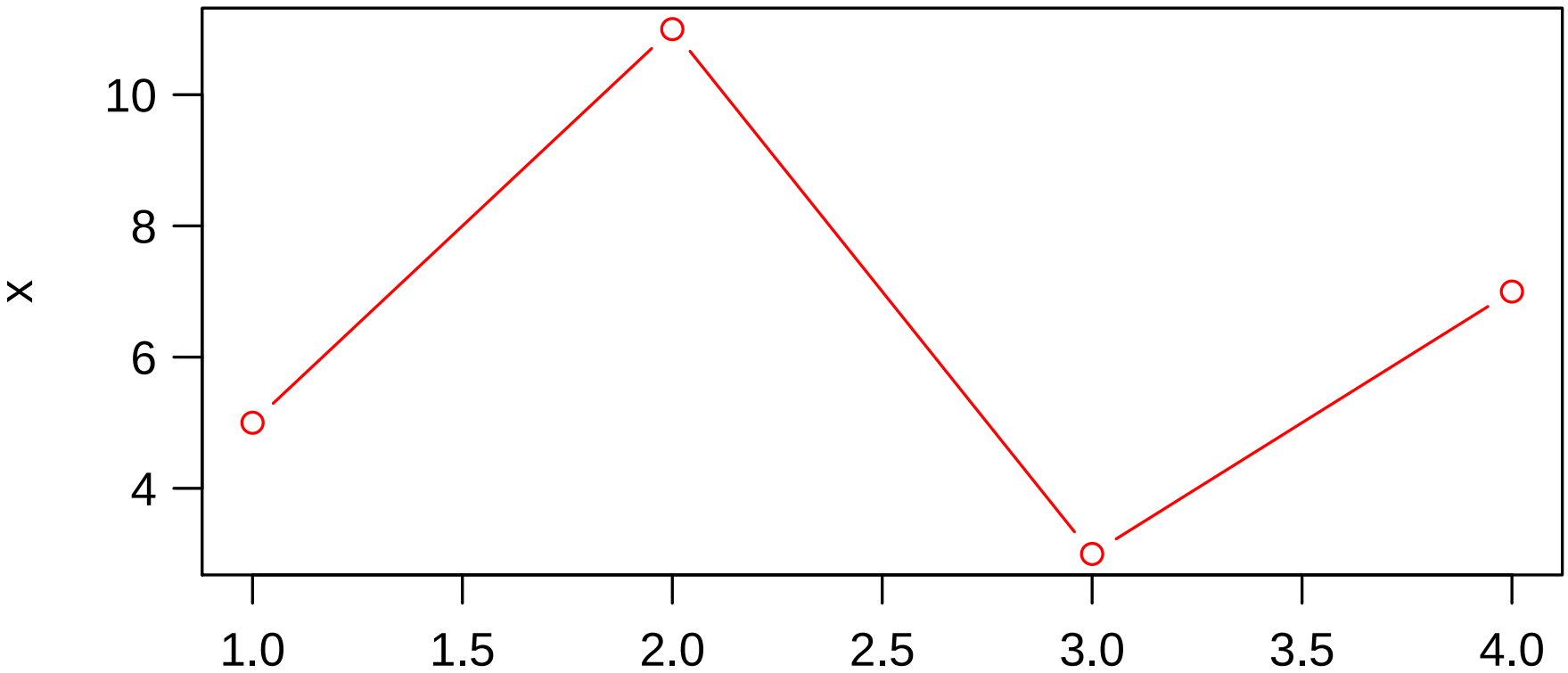
Functions II

```
myfunct(   c(5,11,3,7)  )

## [1] 35 77 21 49
```



Functions with more arguments + default values

```
myfunct <- function(x, type="b", ...) plot(x, type=type, ...)
# type="b" is now the default, thus used unless specified
# The ellipsis (...) passes arguments to other functions
myfunct(  c(5,11,3,7) , col="red", las=1)
```



Conditional execution

Syntax: `if(this_is_true) {do_something}`

Conditional execution

Syntax: `if(this_is_true) {do_something}`

```
if(this_is_true) {do_something} else {do_other_thing}
```

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

```
7-3 > 2
class(7-3 > 2 )
if(7-3 > 2) 18
if(7-3 > 5) 18
if(7-3 > 5) 18 else 17
```

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

```
7-3 > 2                          TRUE
class(7-3 > 2 )
if(7-3 > 2) 18
if(7-3 > 5) 18
if(7-3 > 5) 18 else 17
```

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

```
7-3 > 2
class(7-3 > 2 )
if(7-3 > 2) 18
if(7-3 > 5) 18
if(7-3 > 5) 18 else 17
```

```
TRUE
logical = truth value, boolean
```

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

`7-3 > 2`	TRUE
`class(7-3 > 2 )`	logical = truth value, boolean
`if(7-3 > 2) 18`	Condition is TRUE, so 18 is returned
`if(7-3 > 5) 18`	
`if(7-3 > 5) 18 else 17`	

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

`7-3 > 2`	TRUE
`class(7-3 > 2 )`	logical = truth value, boolean
`if(7-3 > 2) 18`	Condition is TRUE, so 18 is returned
`if(7-3 > 5) 18`	Condition is FALSE, so nothing happens
`if(7-3 > 5) 18 else 17`	

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

`7-3 > 2`	TRUE
`class(7-3 > 2 )`	logical = truth value, boolean
`if(7-3 > 2) 18`	Condition is TRUE, so 18 is returned
`if(7-3 > 5) 18`	Condition is FALSE, so nothing happens
`if(7-3 > 5) 18 else 17`	Condition FALSE, so 17 is returned.

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

`7-3 > 2`	TRUE
`class(7-3 > 2 )`	logical = truth value, boolean
`if(7-3 > 2) 18`	Condition is TRUE, so 18 is returned
`if(7-3 > 5) 18`	Condition is FALSE, so nothing happens
`if(7-3 > 5) 18 else 17`	Condition FALSE, so 17 is returned.

```
if(length(input)>1)
  stop("length must be 1, not ", length(input))
```

Conditional execution

Syntax: `if(this_is_true) {do_something}`
`if(this_is_true) {do_something} else {do_other_thing}`

If condition == TRUE, then expression1 is evaluated,
if condition == FALSE, then expression2 is evaluated.

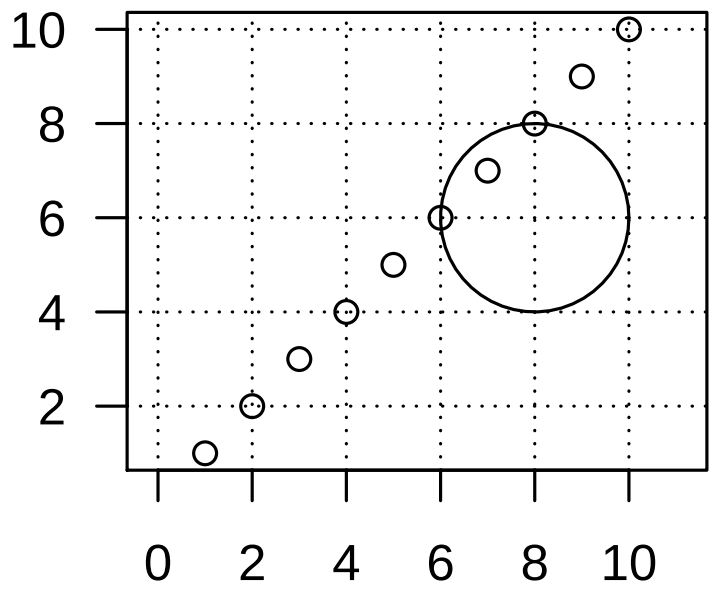
`7-3 > 2`	TRUE
`class(7-3 > 2 )`	logical = truth value, boolean
`if(7-3 > 2) 18`	Condition is TRUE, so 18 is returned
`if(7-3 > 5) 18`	Condition is FALSE, so nothing happens
`if(7-3 > 5) 18 else 17`	Condition FALSE, so 17 is returned.

```
if(length(input)>1)
  stop("length must be 1, not ", length(input))
```

`stop`: Interrupts function execution and gives error
`warning`: continues but gives warning
`message`: to inform instead of worry the user

Exercise: add circles with given radius

```
plot(1:10, asp=1) # aspect ratio y/x of graph range
grid(col=1)
# the next part sould go into a function:
x <- 8 ;  y <- 6 ;  r <-2
p <- seq(0, 2*pi, len=50)
cx <- x+r*cos(p) ;  cy <- y+r*sin(p)
polygon(cx, cy)
```



Time to practice programming

Exercise 2: Writing functions

Write a function that

1. - draws a circle with a certain radius at user-specified locations of an existing plot (see last slide).
2. - uses ellipsis (`...`) to allow the user to customize the appearance
3. - checks all the arguments and gives useful warnings if the wrong type of input is provided
4. - has useful explanations for each argument (documentation)
5. - has readable indentation, spacing and comments explaining the code
6. Now let your neighbor use it without explaining how it is to be used (this should be inferred from the code and comments!)
7. Use your neighbor's function with a vector to draw several circles at once. (unintended use?) What happens?
8. BONUS: Learn writing packages at packdev.R *(rightclick **Raw**, save as)*

```
# Small helper function drawing circles into existing graphics
circle <- function(
  x, # x-coordinate of points, numeric value of length 1
  y, # ditto for y
  r, # radius of the circle, in the graphic's units
  locnum=100, # number of points on circle (more means smoother but slower)
  ...) # Further Arguments passed to polygon, like col, border, lwd
 {

 #
```

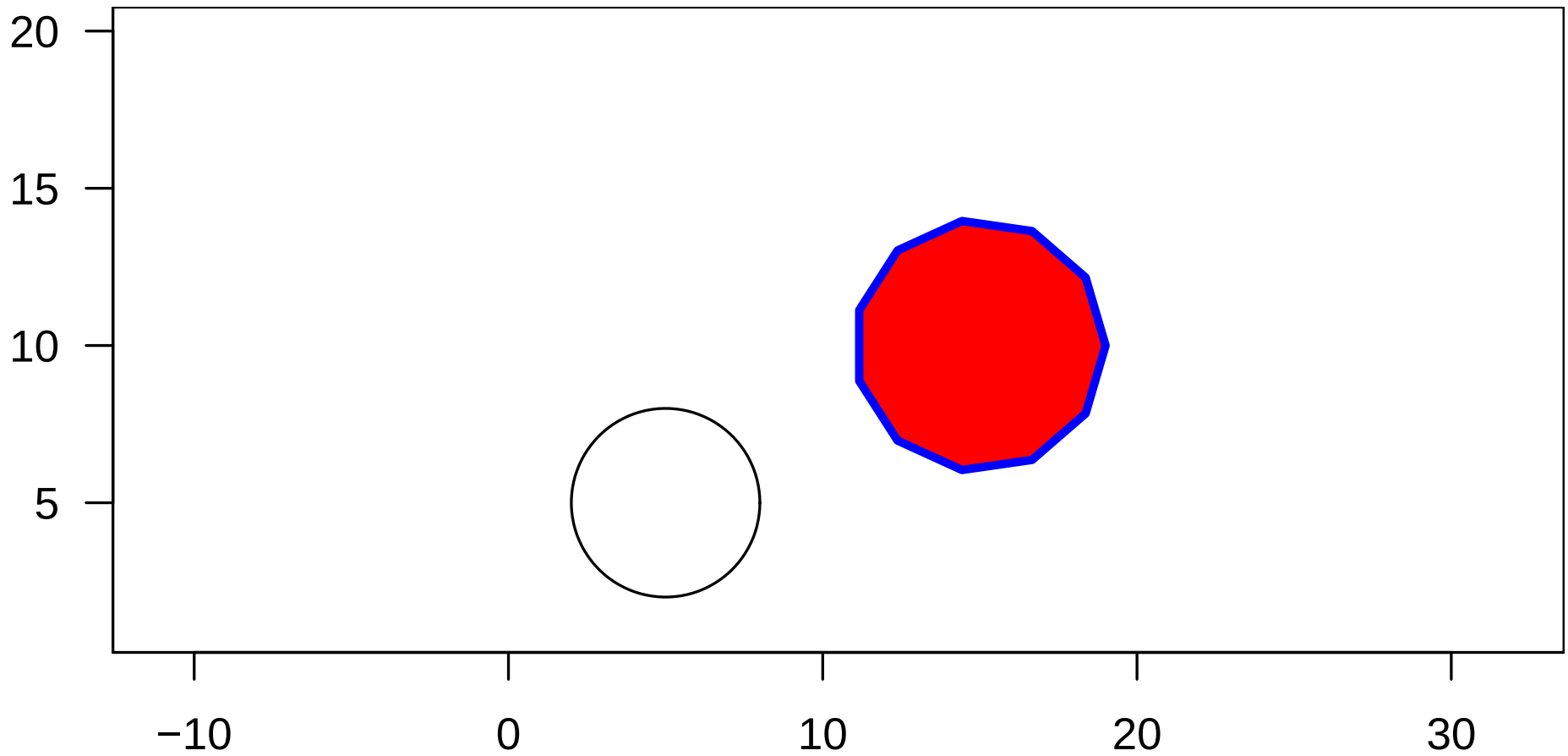
```r
# Small helper function drawing circles into existing graphics
circle <- function(
  x, # x-coordinate of points, numeric value of length 1
  y, # ditto for y
  r, # radius of the circle, in the graphic's units
  locnum=100, # number of points on circle (more means smoother but slower)
  ...) # Further Arguments passed to polygon, like col, border, lwd
 {

 # input checking - only one circle can be drawn:
 if(length(x) >1 | length(y) >1 | length(r) >1 | length(locnum) >1)
   {
   warning("Only the first element of the vectors is used.")
   x <- x[1]; y <- y[1]; r <- r[1]; locnum <- locnum[1]
   }

 #
```

```r
# Small helper function drawing circles into existing graphics
circle <- function(
  x, # x-coordinate of points, numeric value of length 1
  y, # ditto for y
  r, # radius of the circle, in the graphic's units
  locnum=100, # number of points on circle (more means smoother but slower)
  ...) # Further Arguments passed to polygon, like col, border, lwd
 {

 # input checking - only one circle can be drawn:
 if(length(x) >1 | length(y) >1 | length(r) >1 | length(locnum) >1)
   {
   warning("Only the first element of the vectors is used.")
   x <- x[1]; y <- y[1]; r <- r[1]; locnum <- locnum[1]
   }

 if(!is.numeric(x)) stop("x must be numeric, not ", class(x))
 if(!is.numeric(y)) stop("y must be numeric, not ", class(y))
 if(!is.numeric(r)) stop("r must be numeric, not ", class(r))

 #
```

```r
# Small helper function drawing circles into existing graphics
circle <- function(
  x, # x-coordinate of points, numeric value of length 1
  y, # ditto for y
  r, # radius of the circle, in the graphic's units
  locnum=100, # number of points on circle (more means smoother but slower)
  ...) # Further Arguments passed to polygon, like col, border, lwd
 {

 # input checking - only one circle can be drawn:
 if(length(x) >1 | length(y) >1 | length(r) >1 | length(locnum) >1)
   {
   warning("Only the first element of the vectors is used.")
   x <- x[1]; y <- y[1]; r <- r[1]; locnum <- locnum[1]
   }

 if(!is.numeric(x)) stop("x must be numeric, not ", class(x))
 if(!is.numeric(y)) stop("y must be numeric, not ", class(y))
 if(!is.numeric(r)) stop("r must be numeric, not ", class(r))

 # prepare circle line coordinates:
 cx <- x+r*cos( seq(0,2*pi,len=locnum) )
 cy <- y+r*sin( seq(0,2*pi,len=locnum) )
 polygon(cx, cy, ...) # actually draw it
 }
# Note: if circles look like ellipsis, use plot(... asp=1)
```

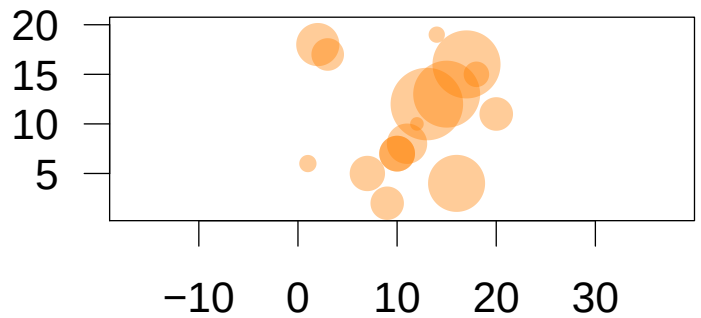
Solution for exercise 2 II: functions

```
plot(1:20, type="n", asp=1, cex=2)
circle(5,5, r=3)
circle(15,10, r=4, locnum=12, col=2, border=4, lwd=3)
```



Solution for exercise 2 III: functions

```
# can not be vectorized:
x <- sample(1:20, 15) ;  y <- sample(1:20, 15) ; r <- runif(20)*4
circle(x,y,r, col=rgb(1,0.5,0,alpha=0.4), border=NA)

## Warning in circle(x, y, r, col = rgb(1, 0.5, 0, alpha = 0.4), border = NA):
Only the first element of the vectors is used.

for(i in 1:15) circle(x[i],y[i],r[i], col=rgb(1,0.5,0,alpha=0.4), border=NA)
```



Outline

Debugging

- Your code throws an error. You didn't call the mentioned function. Obviously, your code calls some function calling some function calling some function calling [you get the idea] which in the end creates an error. To trace back this path, you can use `traceback()`.

Debugging

- Your code throws an error. You didn't call the mentioned function. Obviously, your code calls some function calling some function calling some function calling [you get the idea] which in the end creates an error. To trace back this path, you can use `traceback()`.
- Now that you know where the error originates from, you set `options(error=recover)`. You run your code again, but this time R waits at the level creating an error. You examine the environment within the function, play around with the objects and internal function code, until the bug has been fixed. You have just debugged a function.

Debugging

- Your code throws an error. You didn't call the mentioned function. Obviously, your code calls some function calling some function calling some function calling [you get the idea] which in the end creates an error. To trace back this path, you can use `traceback()`.
- Now that you know where the error originates from, you set `options(error=recover)`. You run your code again, but this time R waits at the level creating an error. You examine the environment within the function, play around with the objects and internal function code, until the bug has been fixed. You have just debugged a function.
- You want to step into the function you are developing at a specific point. You add `browser()` at that point of the code.

Debugging: useful functions + Resources

Debugging: useful functions + Resources

source("projectFuns.R") execute complete file

Debugging: useful functions + Resources

source("projectFuns.R")	execute complete file
traceback()	find error source in sequence of function calls

Debugging: useful functions + Resources

source("projectFuns.R")	execute complete file
traceback()	find error source in sequence of function calls
options(warn=2)	warnings to error. default 0

Debugging: useful functions + Resources

source("projectFuns.R")	execute complete file
traceback()	find error source in sequence of function calls
options(warn=2)	warnings to error. default 0
browser()	go into function environment: `n`, `s`, `f`, `c`, `Q`

Debugging: useful functions + Resources

source("projectFuns.R")	execute complete file
traceback()	find error source in sequence of function calls
options(warn=2)	warnings to error. default 0
browser()	go into function environment: `n`, `s`, `f`, `c`, `Q`
options(error=recover)	open interactive session where error occurred

Debugging: useful functions + Resources

source("projectFuns.R")	execute complete file
traceback()	find error source in sequence of function calls
options(warn=2)	warnings to error. default 0
browser()	go into function environment: `n`, `s`, `f`, `c`, `Q`
options(error=recover)	open interactive session where error occurred
debug(funct)	toggle linewise function execution

Debugging: useful functions + Resources

source("projectFuns.R")	execute complete file
traceback()	find error source in sequence of function calls
options(warn=2)	warnings to error. default 0
browser()	go into function environment: `n`, `s`, `f`, `c`, `Q`
options(error=recover)	open interactive session where error occurred
debug(funct)	toggle linewise function execution
undebug(funct)	after calling and fixing funct

Debugging: useful functions + Resources

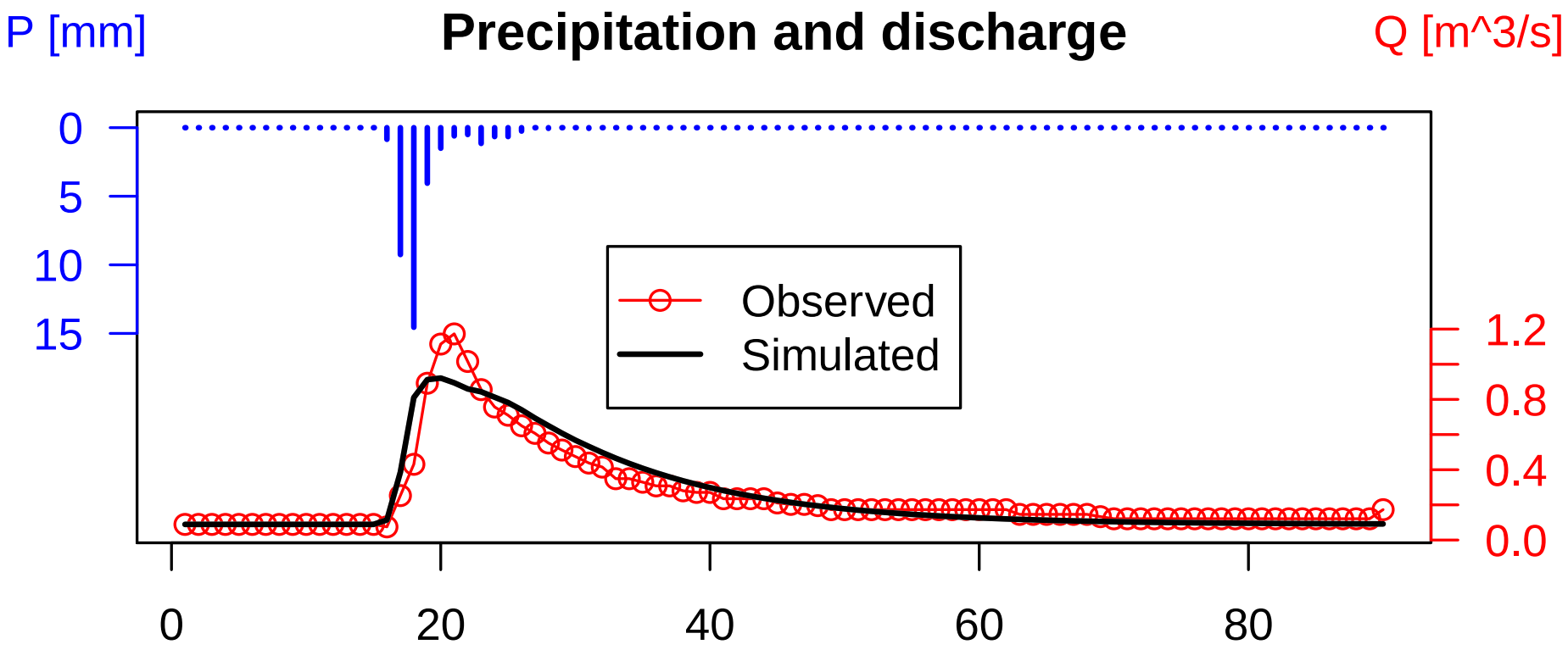
source("projectFuns.R")	execute complete file
traceback()	find error source in sequence of function calls
options(warn=2)	warnings to error. default 0
browser()	go into function environment: `n`, `s`, `f`, `c`, `Q`
options(error=recover)	open interactive session where error occurred
debug(funct)	toggle linewise function execution
undebug(funct)	after calling and fixing funct

R. Peng (2002): Interactive Debugging Tools in R
D. Murdoch (2010): Debugging in R
H. Wickham (2015): Advanced R: debugging
Example: Pete Werner Blog Post (2013)

Practice debugging, lsc_functions.R *(rightclick **Raw**, save as)*

Exercise 3: Debugging

1. Load your package and the datasets. Correct the functions until `lsc(calib$P, calib$Q, area=1.6)` returns the result below.
2. BONUS: commit each change to git.



Solution for exercise 3: Debugging

- `stupid error you can easily remove` - traceback - find location of error - lsc#73 - just comment it out
- `harder to find but still stupid` - traceback - nse#11 - ditto
- `Error in plot:  need finite 'ylim' value` - browser/options(error=recover) - lsc#105 - NAs in Q - range(Q, na.rm=TRUE) - also in other applicable locations
- `There were 50 or more warnings` - come from rmse being called in optimization - add argument quietNA (or similar) to lsc that is passed to rmse in lsc#79