loops and functions in 

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

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 2019-11-15, 10:06

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, 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  }
```

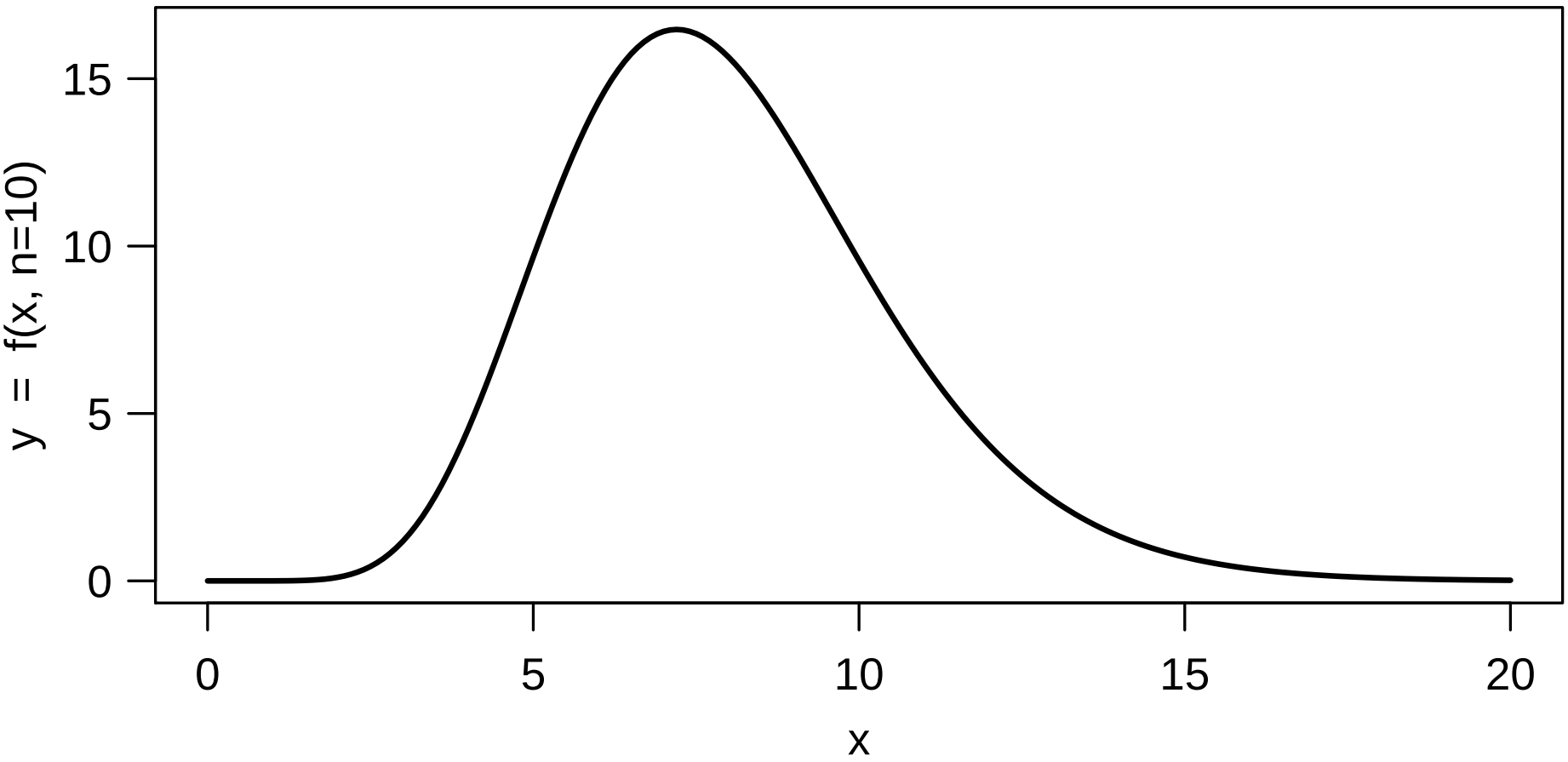
```
v # this code was executed once for each i

##  [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).

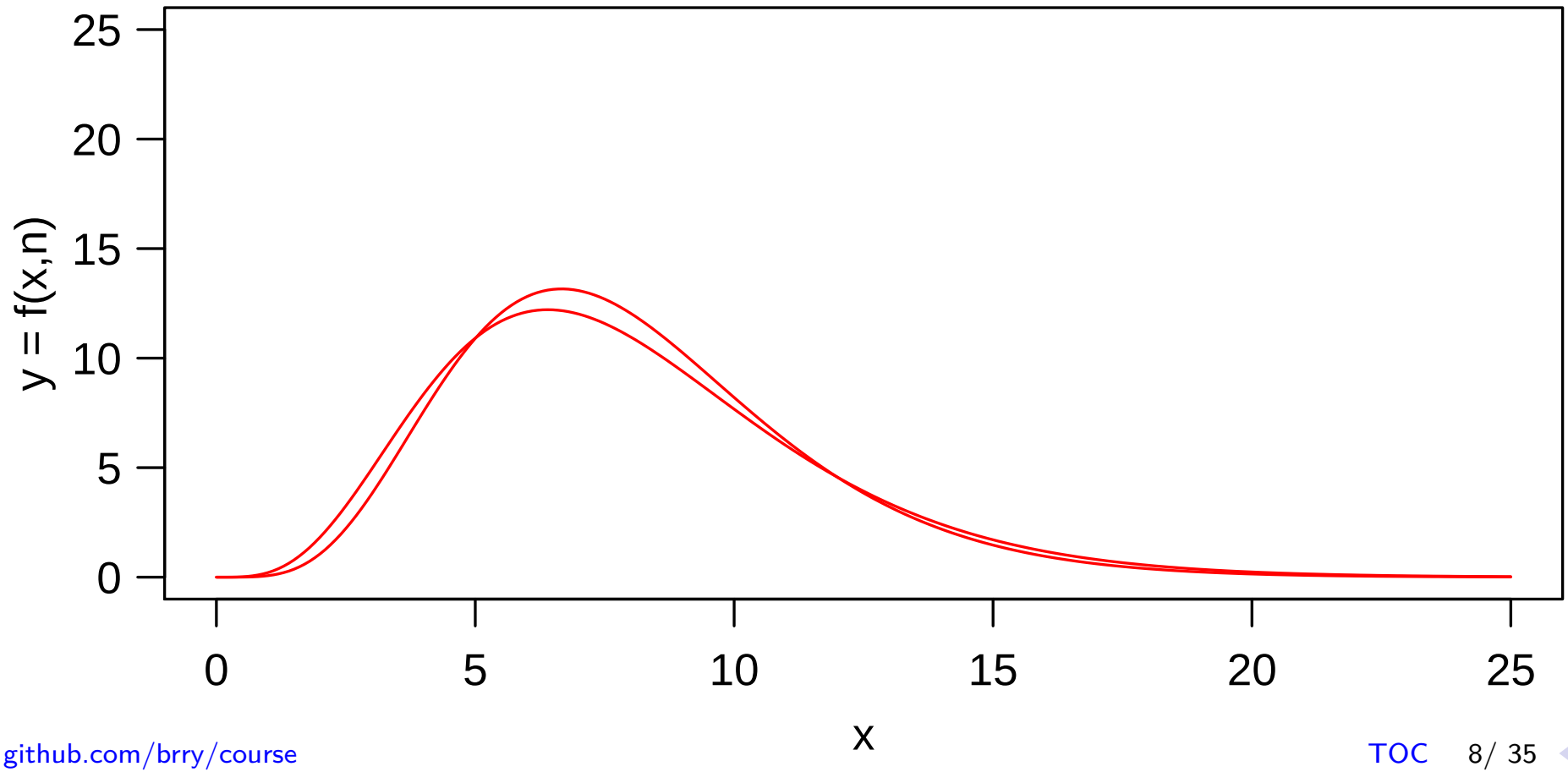
For loops: execute code multiple times I

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



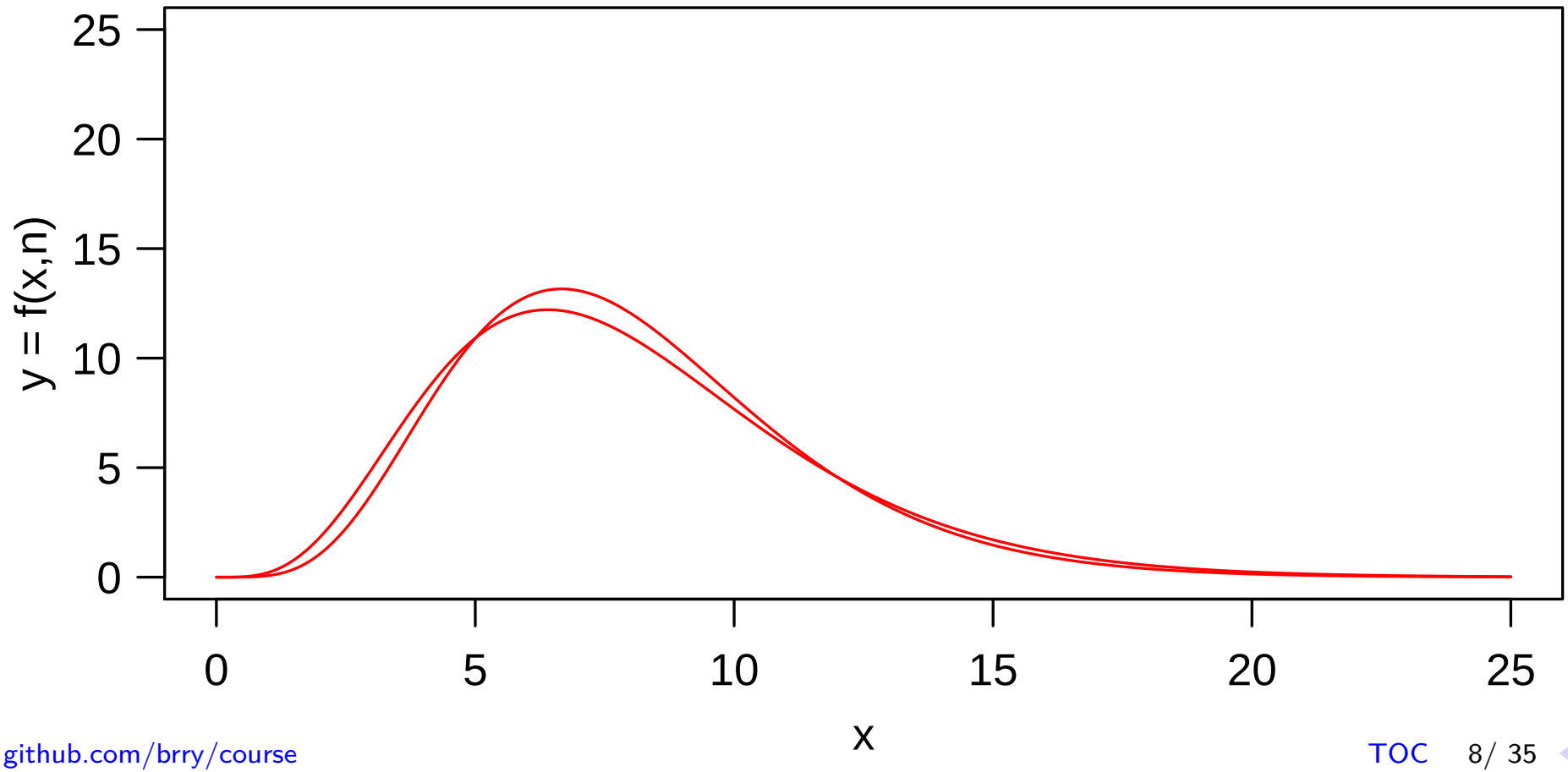
For loops: execute code multiple times II

```
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)
```



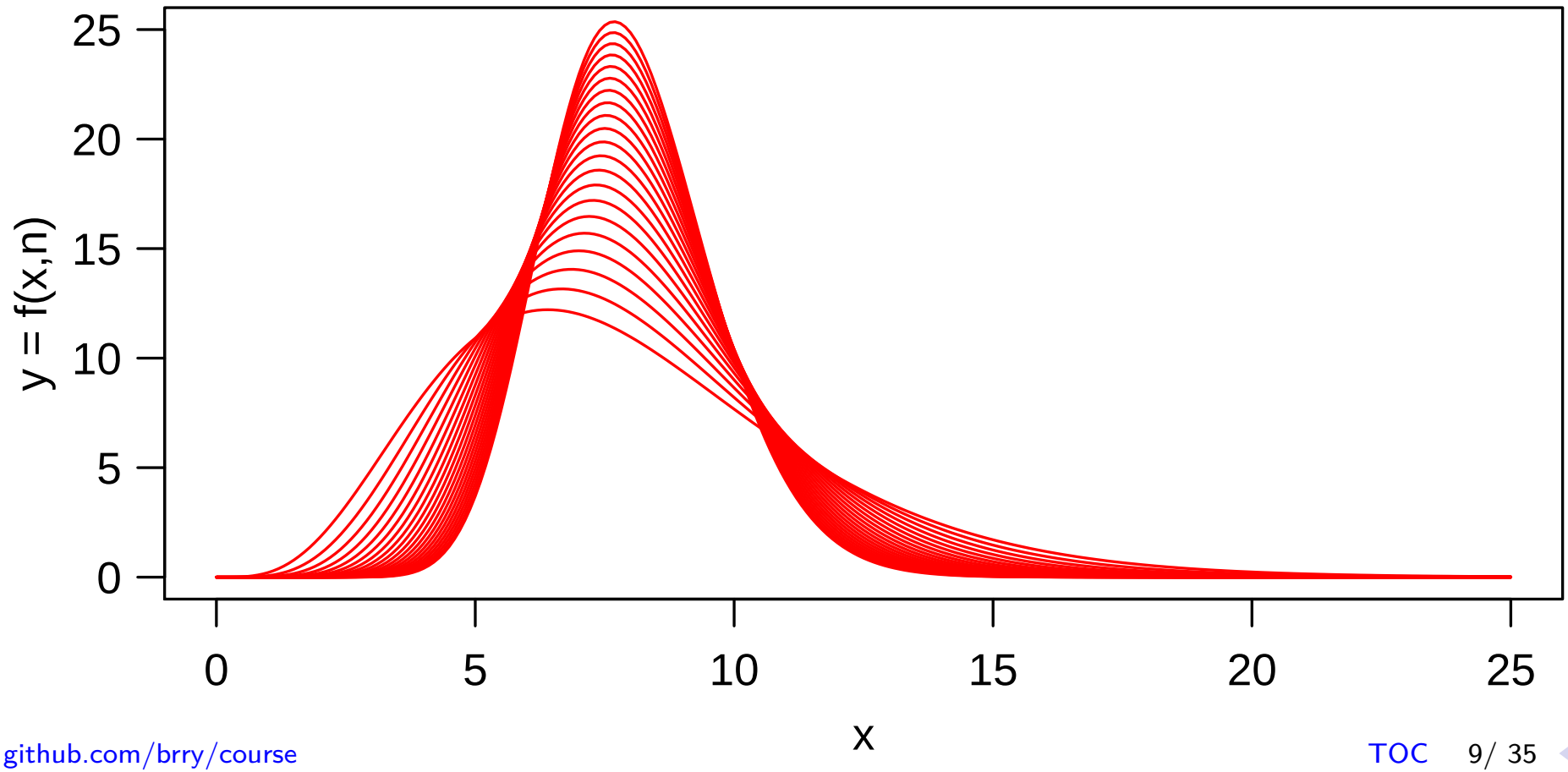
For loops: execute code multiple times II

```
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 shouldn't grow a vector

Bad practice - R needs to recreate the vector each time:

```
output <- NA
for(column in 1:5) output[column] <- median(iris[ ,column])
```

for loops shouldn't grow a vector

Bad practice - R needs to recreate the vector each time:

```
output <- NA
for(column in 1:5) output[column] <- median(iris[ ,column])
```

This internally does the same thing as:

```
output <- vector(mode="numeric", length=0)
for(column in 1:5) output <- c(output, median(iris[ ,column])  )
```

for loops shouldn't grow a vector

Bad practice - R needs to recreate the vector each time:

```
output <- NA
for(column in 1:5) output[column] <- median(iris[ ,column])
```

This internally does the same thing as:

```
output <- vector(mode="numeric", length=0)
for(column in 1:5) output <- c(output, median(iris[ ,column])  )
```

Good practice - first tell R how big the output will be, so it can be adequately allocated in memory:

```
output <- vector(mode="numeric", length=5)
for(column in 1:5) output[column] <- median(iris[ ,column])
```

for loops exercise

Exercise 1: for loops in file creation

We'll write many datasets to disc (and read them back).

1. With `paste0`, print a filename of the structure "mydata_123.txt" using the name, the number and the file ending as inputs. We'll be changing the number later in a loop.
2. Print a data.frame with two columns, each with 10 random numbers: one column from the normal, one from the exponential distribution
3. With `write.table`, write such a table to a file in a subfolder (remember `dir.create`), using the number of rows (e.g. 10) in the filename.
4. BONUS: change the arguments so that row numbers and quotation marks are not printed and tabstops are used for column separation.
5. With a `for`-loop, now write files for different sample sizes, e.g. 10, 20, 50, 100, 500.
6. Using the output of `dir()`, read all the files into a list of data.frames. Remember to first create an empty list of the right length.
7. BONUS: name the list elements according to the filenames.
8. BONUS: now replace the whole construct with an `lapply` loop. Celebrate how much nicer your code looks. Check how you can get element names with `sapply(..., simplify=FALSE)`
9. BONUS: With `unlink`, delete the files from this exercise. This function is vectorizable, so there's no need to do this in a for loop!

for loops exercise solution

```
dir.create("loopexercise")
for(n in c(10,20,50,100,500))
    write.table(x=data.frame(norm=rnorm(n), exp=rexp(n)),
                file=paste0("loopexercise/randomdata_", n, ".txt"),
                quote=F, row.names=F, sep="\t")

fnames <- dir("loopexercise", full=TRUE)
fcontents <- vector("list", length=length(fnames))
for(fnum in seq_along(fnames))
   fcontents[[fnum]] <- read.table(fnames[fnum], header=TRUE)

flist <- sapply(dir("loopexercise", full=TRUE), read.table, header=TRUE,
                simplify=FALSE)

unlink(paste0("loopexercise/randomdata_", c(10,20,50,100,500), ".txt"))
```

`seq_along(n)` is safer than `1:n` in for loops

`seq_along(n)` is safer than `1:n` in `for` loops

```
do_something <- function(x) if(x<1) stop("x must be >=1, not:", x) else x
something <- 1:6
```

`seq_along(n)` is safer than `1:n` in `for` loops

```
do_something <- function(x) if(x<1) stop("x must be >=1, not:", x) else x
something <- 1:6
```

You'll often see the dangerous code `for(i in 1:n)`:

```
for(i in 1:length(something)) do_something(i) # works with current sth
```

`seq_along(n)` is safer than `1:n` in `for` loops

```
do_something <- function(x) if(x<1) stop("x must be >=1, not:", x) else x
something <- 1:6
```

You'll often see the dangerous code `for(i in 1:n)`:

```
for(i in 1:length(something)) do_something(i) # works with current sth
```

Imagine this:

```
something <- which(letters=="4")
for(i in 1:length(something)) do_something(i) # fails! (same code!)

## Error in do_something(i):  x must be >=1, not:0
```

`seq_along(n)` is safer than `1:n` in `for` loops

```
do_something <- function(x) if(x<1) stop("x must be >=1, not:", x) else x
something <- 1:6
```

You'll often see the dangerous code `for(i in 1:n)`:

```
for(i in 1:length(something)) do_something(i) # works with current sth
```

Imagine this:

```
something <- which(letters=="4")
for(i in 1:length(something)) do_something(i) # fails! (same code!)

## Error in do_something(i):  x must be >=1, not:0
```

Safer to use is:

```
for(i in seq_along(something)) do_something(i)
```

seq_along(n) is safer than 1:n in for loops

```
do_something <- function(x) if(x<1) stop("x must be >=1, not:", x) else x
something <- 1:6
```

You'll often see the dangerous code for(i in 1:n):

```
for(i in 1:length(something)) do_something(i) # works with current sth
```

Imagine this:

```
something <- which(letters=="4")
for(i in 1:length(something)) do_something(i) # fails! (same code!)

## Error in do_something(i):  x must be >=1, not:0
```

Safer to use is:

```
for(i in seq_along(something)) do_something(i)
```

Because:

```
1:length(something)     ;   seq_along(something)

## [1] 1 0
## integer(0)
```

stocks data from finance.yahoo.com

```
# Download current datasets:
if(!requireNamespace("quantmod")) install.packages("quantmod")
if(!requireNamespace("pbapply"))  install.packages("pbapply")
dummy <- pbapply::pblapply(c("F","VLKAF", "AMZN","AAPL","GOOG","MSFT"),
   function(x) zoo::write.zoo(x=quantmod::getSymbols(x, auto.assign=FALSE)[,6],
                              file=paste0("data/finance/",x,".txt"), col.names=T))

# read single files to R and merge into one file:
stocks <- lapply(dir("data/finance", full=TRUE),
                 read.table, as.is=TRUE, header=TRUE)
stocks <- Reduce(function(...) merge(..., all=T), stocks)

# Get nicer column names:
names <- sapply(strsplit(colnames(stocks), ".", fix=TRUE),"[", 1)
colnames(stocks) <- c(Index="Date", F="FORD", VLKAF="VOLKSWAGEN",
         AMZN="AMAZON", AAPL="APPLE", GOOG="GOOGLE", MSFT="MICROSOFT")[names]

# Save to disc:
write.table(stocks, file="data/stocks.txt", row.names=F, quote=F)
```

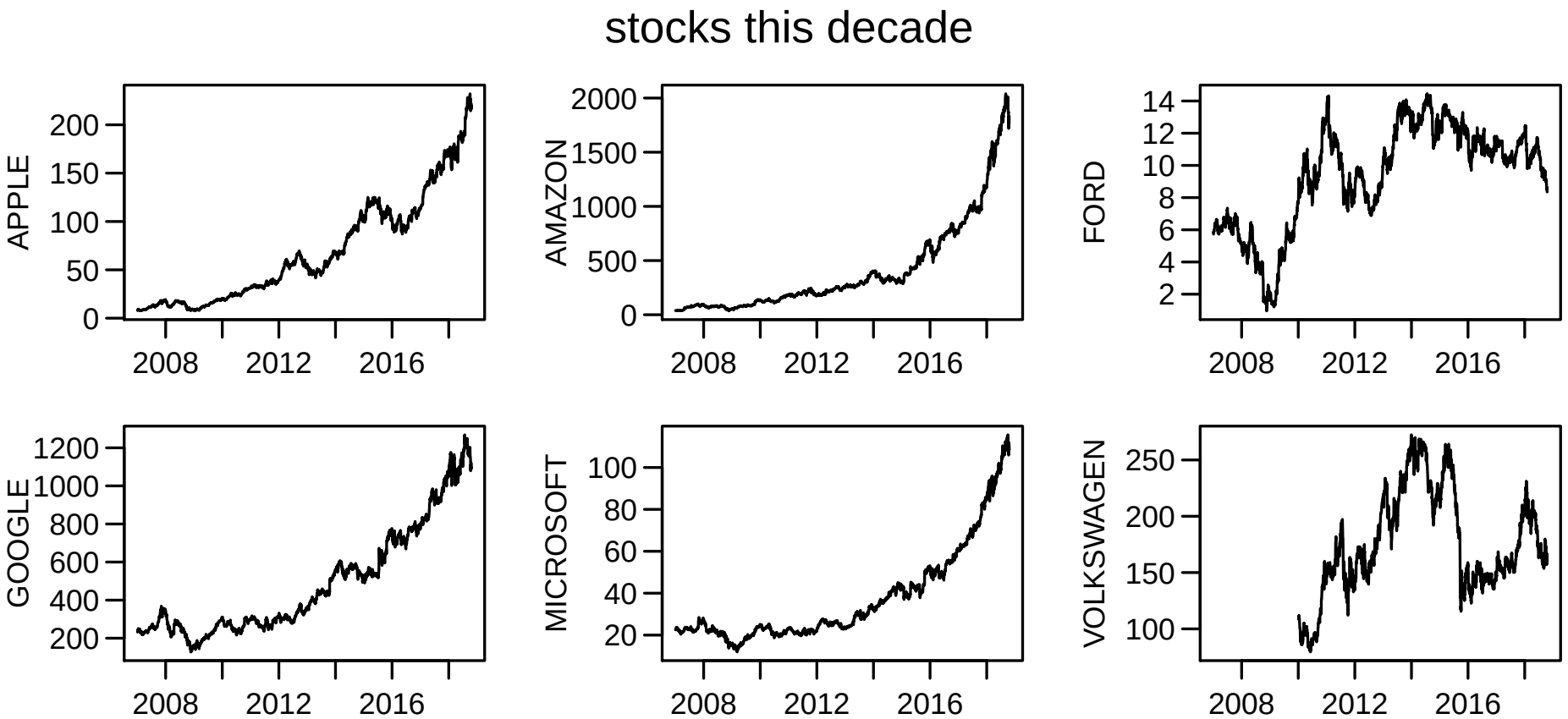
For loops: multipanel graphics: the task

Exercise 2: 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[ ,1:2])`? Make it a line graph.
4. With `par(mfrow...`, set up a two by three panel plot
5. With a for loop, fill those with each stock time series
6. BONUS 1: Make good annotations, including a main title (par oma, `mtext` with the outer argument)
7. BONUS 2: Make the plot margins smaller (par mar), turn y axis labels upright (las) and move the axis labels closer to the plots (mgp).
8. BONUS 3: Understand and comment each line of the data preparation.

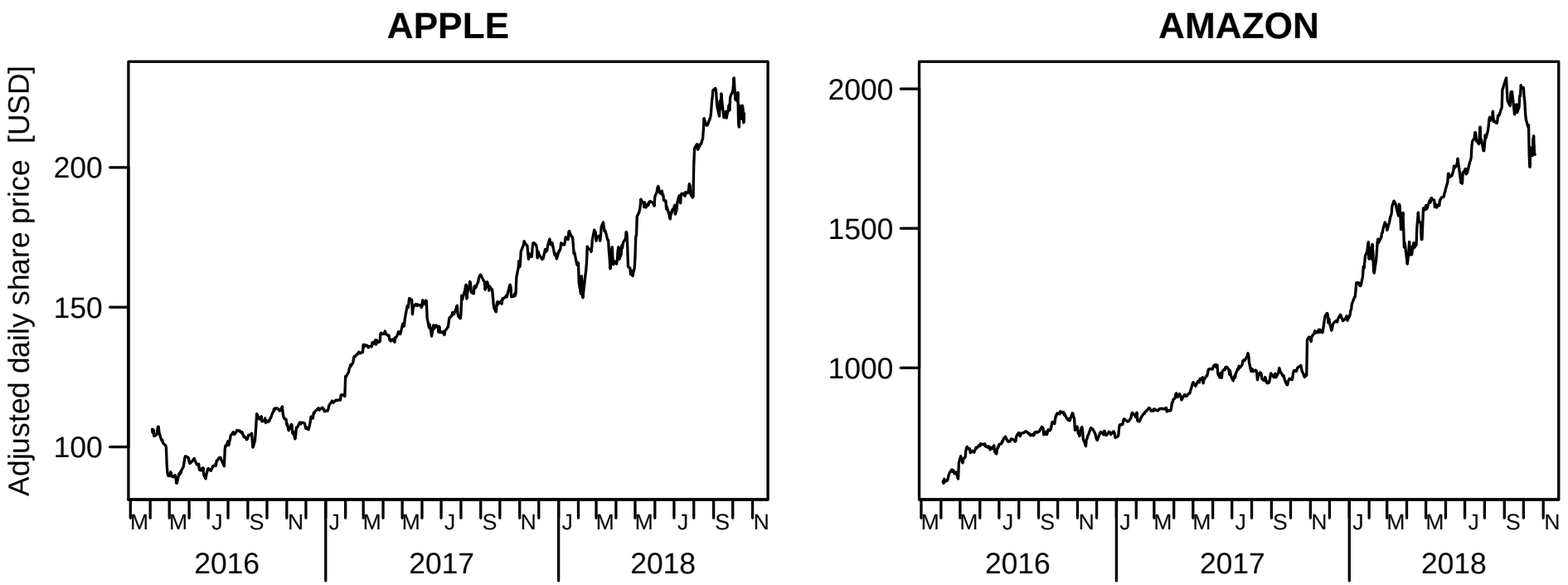
For loops: multipanel graphics: the solution

```
stocks <- read.table("data/stocks.txt", header=T, as.is=T)
stocks$Date <- as.Date(stocks$Date)
par(mfrow=c(2,3), mar=c(2,4,1,1), mgp=c(2.5,0.7,0),oma=c(0,0,2,0),las=1)
for(i in 2:7) plot(stocks[ ,c(1,i)], type="l")
mtext("stocks this decade", line=0, outer=TRUE)
```



For loops: multipanel graphics: beautifuller

```
par(mfrow=c(1,2), mar=c(4,4,2,0.1), mgp=c(2.5,0.7,0), cex=0.7, las=1)
for(i in 2:3)
 {
 plot(stocks[ stocks$Date>as.Date("2016-04-01") , c(1,i) ],
      main=colnames(stocks)[i], xaxt="n", type="l", xlab=" ",
      ylab=if(i==2) "Adjusted daily share price  [USD]" else "")
 berryFunctions::monthAxis(1)
 }
```



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"}
```

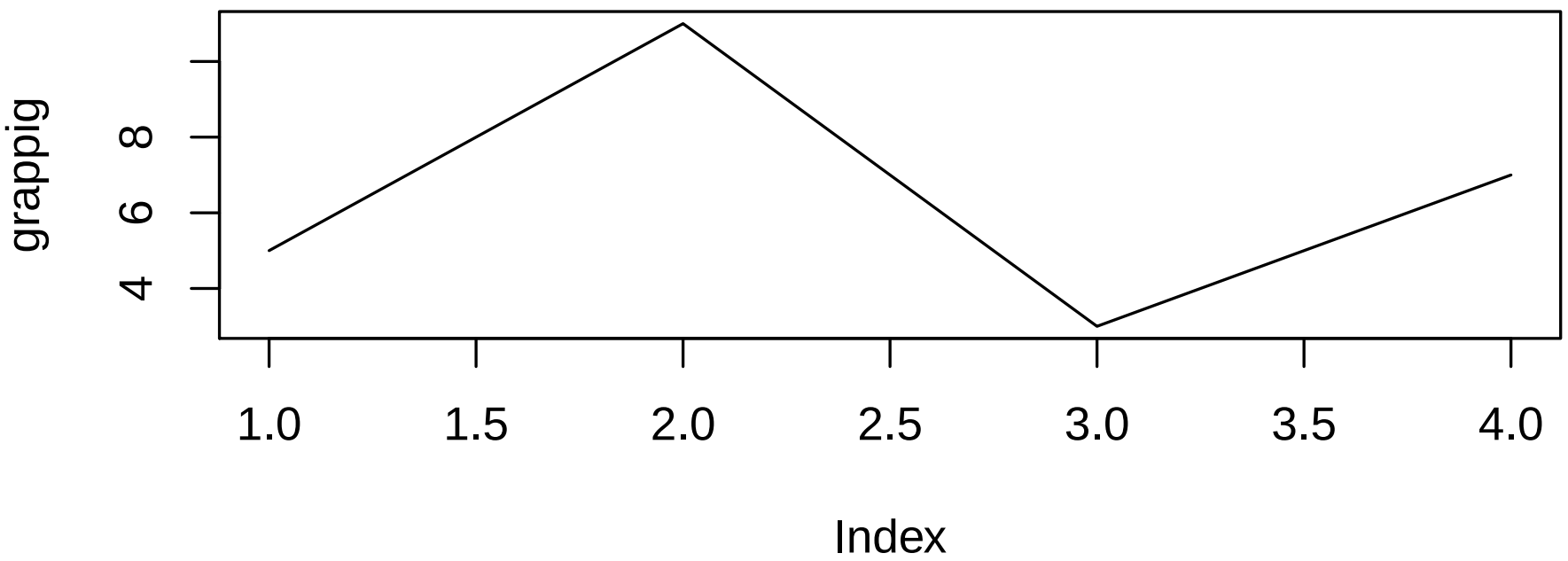
```
myfunct <- function(grappig)
       {plot(grappig, type="l"); return(grappig*7)  }
```

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

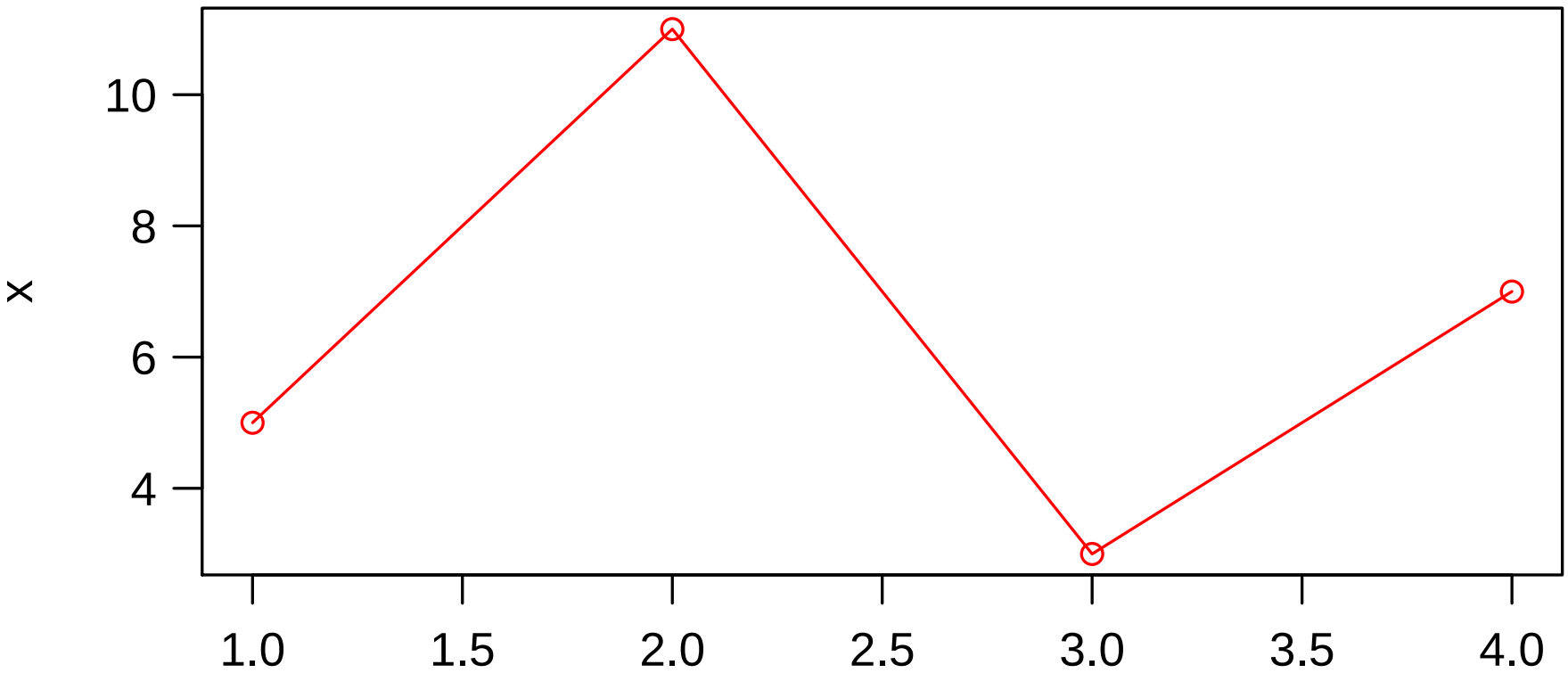
```
myfunct(   c(5,11,3,7)  )

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



Functions with more arguments

```
myfunct <- function(x, type="o", ...) plot(x, type=type, ...)
# type="o" 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)
```



Functions: example I

If you needed to find the zeros of quadratic functions very often, you could use

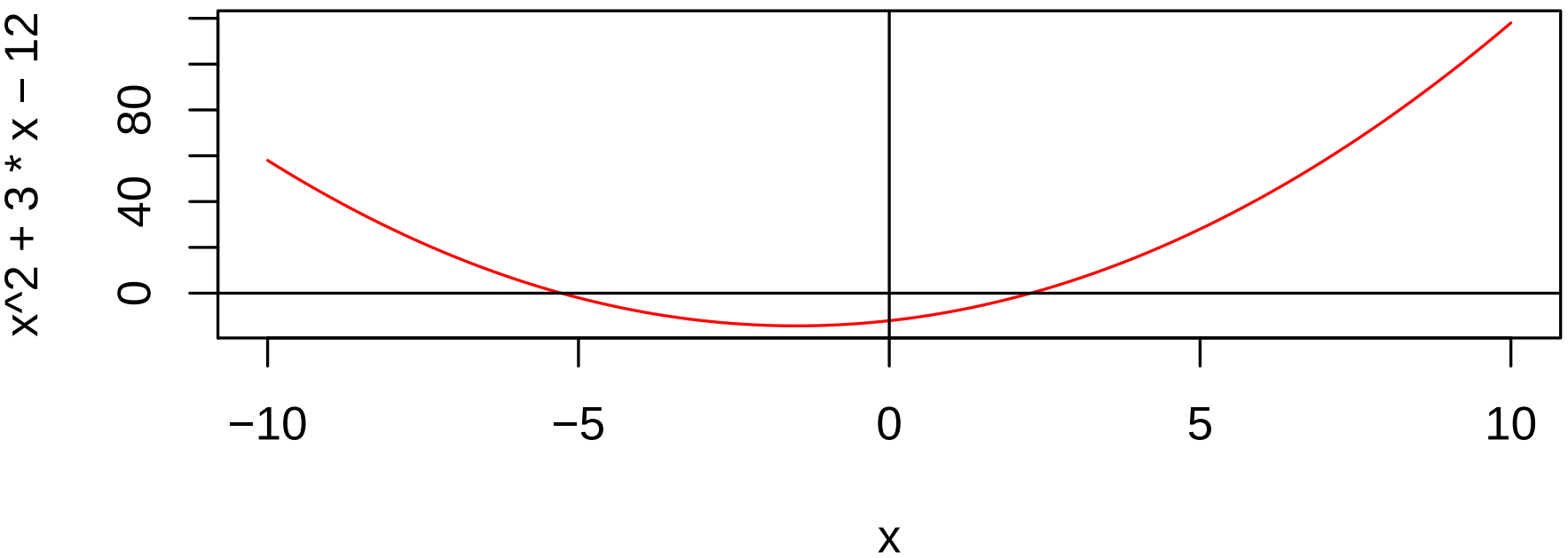
```
pq <- function(p,q) # y = x^2 + px + q
                {
                w <- sqrt( p^2 / 4 - q )
                c(-p/2-w, -p/2+w)
                } # End of function


pq(3, -12)
```

```
## [1] -5.274917  2.274917
```

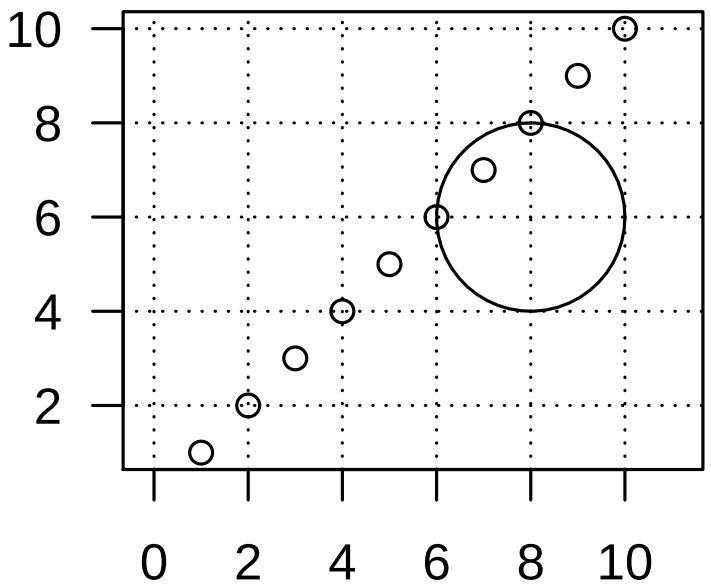
Functions: example II

```
x <- seq(-10, 10, len=100)
plot(x,  x^2 +3*x -12, type="l", col=2)
abline(h=0, v=0)
```



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 3: Writing functions

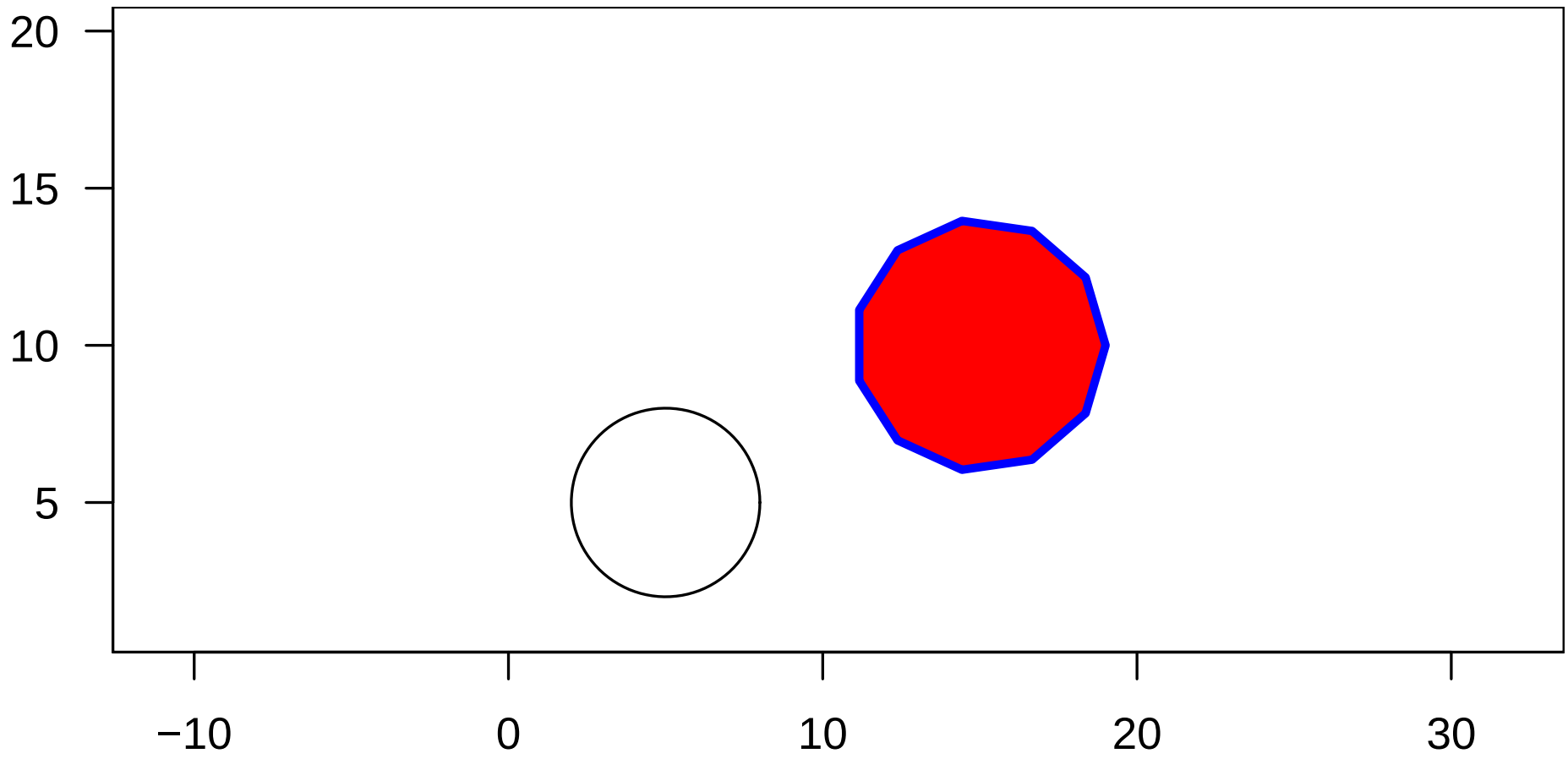
Write a function that

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?

```r
# Small helper function drawing circles into existing graphics
# Berry Boessenkool, berry-b@gmx.de, 2012
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]
   }
 # input checking - is every value numeric?
 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 3 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)
```

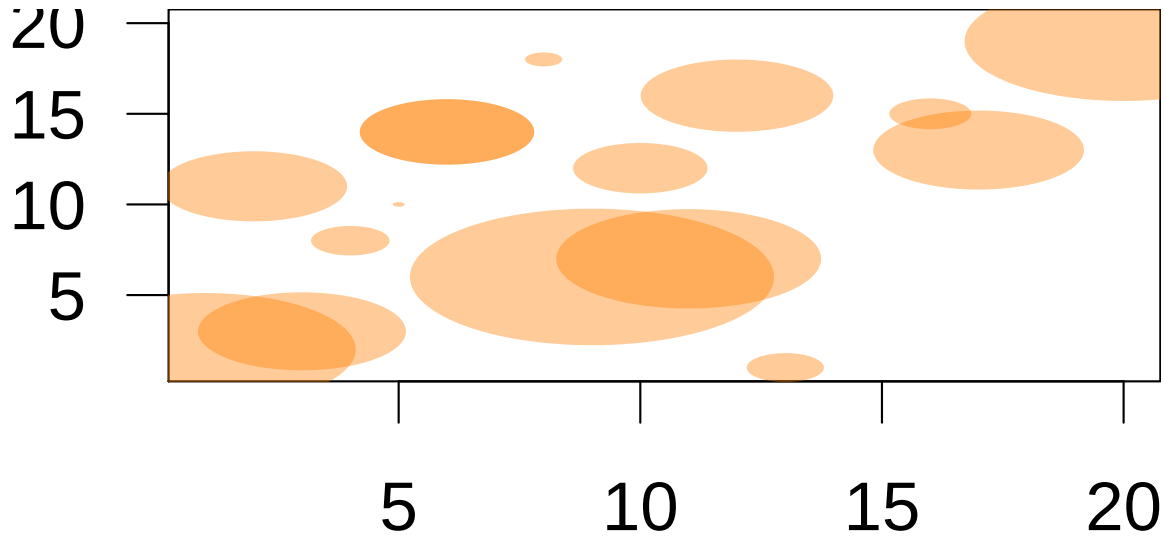


Solution for exercise 3 III: functions

```
# 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. You want to go line by line in one specific function. You set `debug(thatFunction)`.

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. You want to go line by line in one specific function. You set `debug(thatFunction)`.
- You want to learn about lexical scoping (Where does R find variables?). http://trestletech.com/2013/04/package-wide-variablescache-in-r-package/ http://adv-r.had.co.nz/Environments.html

Debugging: useful functions

Debugging: useful functions

source("projectFuns.R") execute complete file

Debugging: useful functions

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

Debugging: useful functions

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

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

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

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

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

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

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

Debugging: useful functions

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

```
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

Debugging: useful functions

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

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

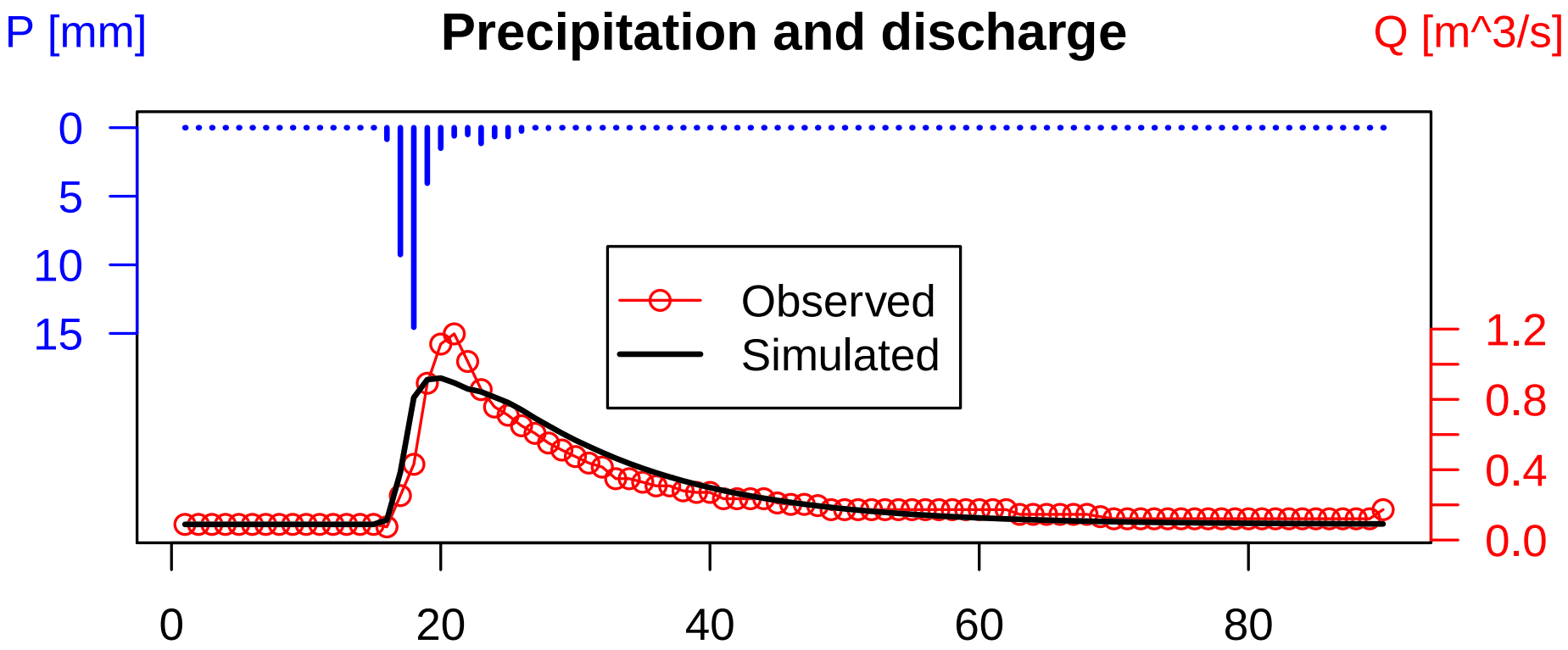
`stop`: Interrupts function execution and gives error
`warning`: continues but gives warning
`message`: to inform instead of worry the user

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 4: 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 4: 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` - debug/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

Outline

Feedback

Please fill out the feedback form at

bit.ly/feedbackR

(it only takes a few minutes and helps to improve the course)

Thanks!