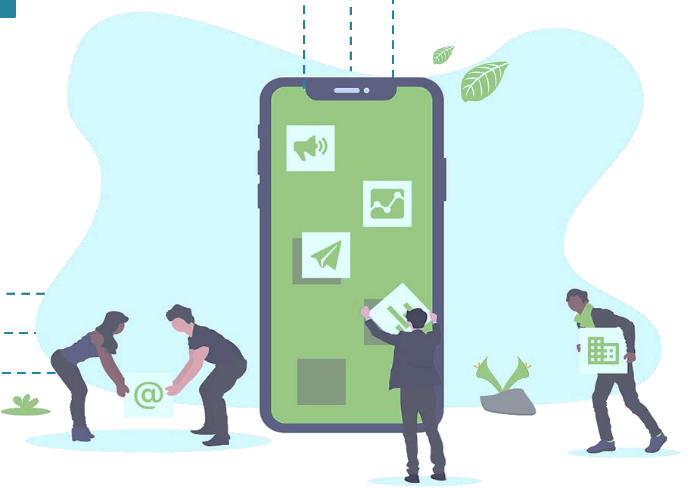
The Data Science Track



Prepared By: R. Daynolo

1

1

13. Functions

2

2

Functions

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {
        ## Do something interesting
}
```

Functions

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

Functions have ***named arguments*** which potentially have ***default values***.

- The *formal arguments* are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be *missing* or might have default values

Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to `sd` are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, it is not recommend to mess around with the order of the arguments too much, since it can lead to some confusion.

Argument Matching

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
```

The following two calls are equivalent.

```
> lm(data = mydata, y~x, model = FALSE, 1:100)
> lm(y~x, mydata, 1:100, model = FALSE)
```

Argument Matching

- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
- Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Argument Matching

Function arguments can also be *partially* matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Prepared By: R. Daynolo

9

9

Defining a Function

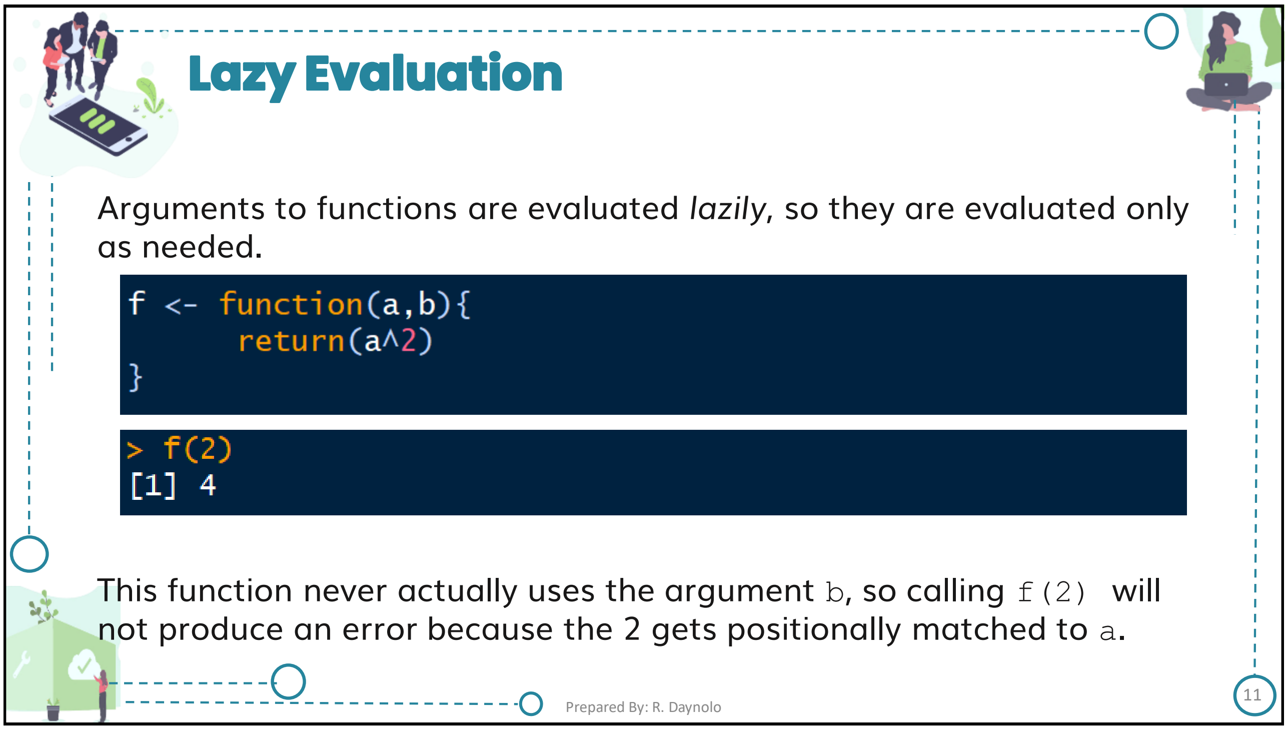
```
f <- function (a, b = 1, c = 2, d = NULL) {
      }
```

In addition to not specifying a default value, you can also set an argument value to `NULL`.

Prepared By: R. Daynolo

10

10



Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed.

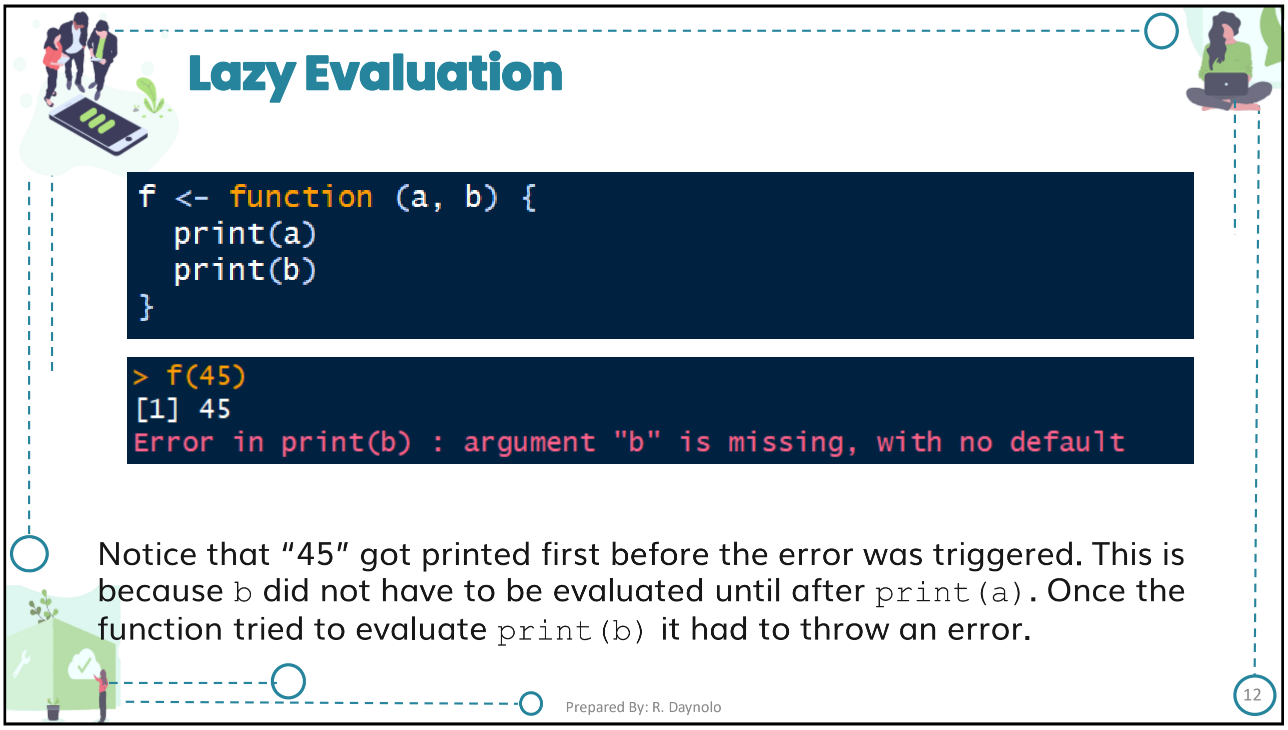
```
f <- function(a,b){
       return(a^2)
}
```

```
> f(2)
[1] 4
```

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the 2 gets positionally matched to `a`.

Prepared By: R. Daynolo

11



Lazy Evaluation

```
f <- function (a, b) {
  print(a)
  print(b)
}
```

```
> f(45)
[1] 45
Error in print(b) : argument "b" is missing, with no default
```

Notice that "45" got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.

Prepared By: R. Daynolo

12

The "…" Argument

The … argument indicate a variable number of arguments that are usually passed on to other functions.

- … is often used when extending another function and you don't want to copy the entire argument list of the original function

```
myplot <- function(x, y, type = "l", ...){
  plot(x,y, type = type, ...)
}
```

- *Generic functions* use … so that extra arguments can be passed to methods (more on this later)

```
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x000000000391e720>
<environment: namespace:base>
```

The "…" Argument

The … argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL
> args(cat)
function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
NULL
```

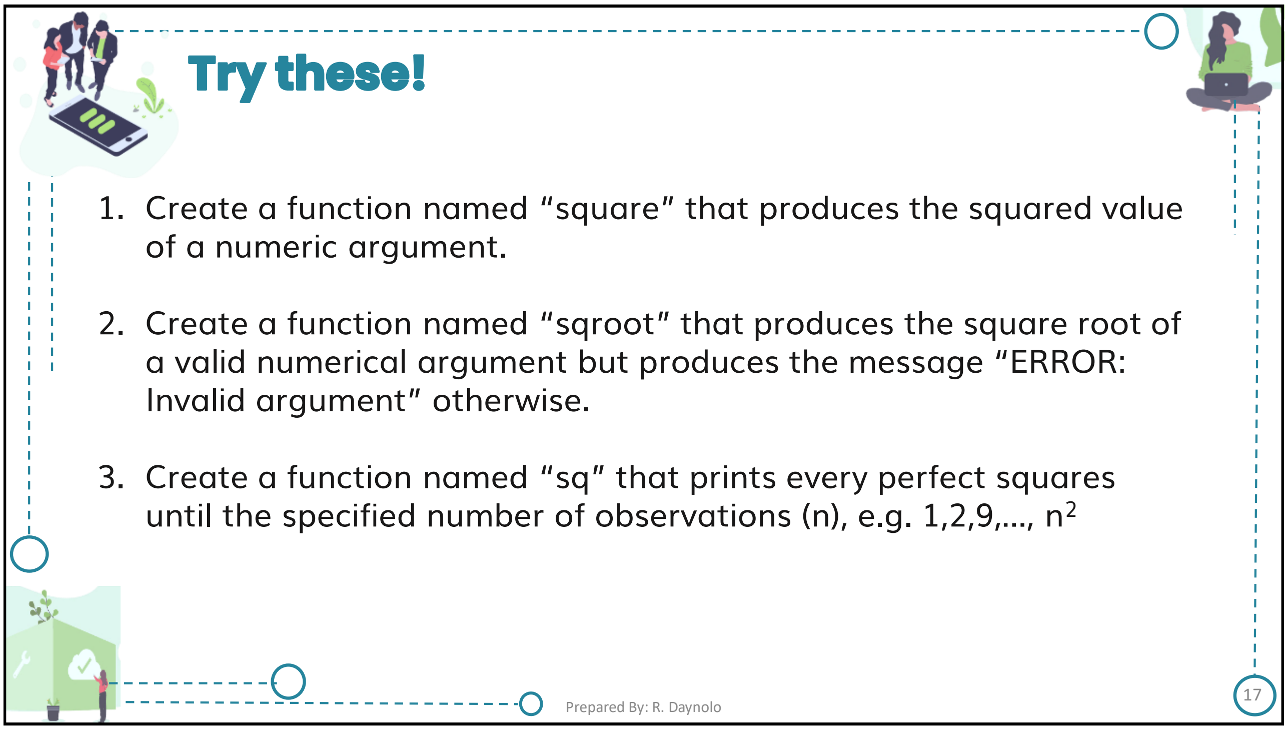
Arguments Coming After the "..." Argument

One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL
> paste("a", "b", sep = ":")
[1] "a:b"
> paste("a", "b", se = ":")
[1] "a b :"
```

Important Notes in Creating Functions

- Make sure all arguments are properly assigned
- The body should contain all necessary steps to be done by R
- `return([object])` function is used to provide a returning value for the function



Try these!

1. Create a function named "square" that produces the squared value of a numeric argument.

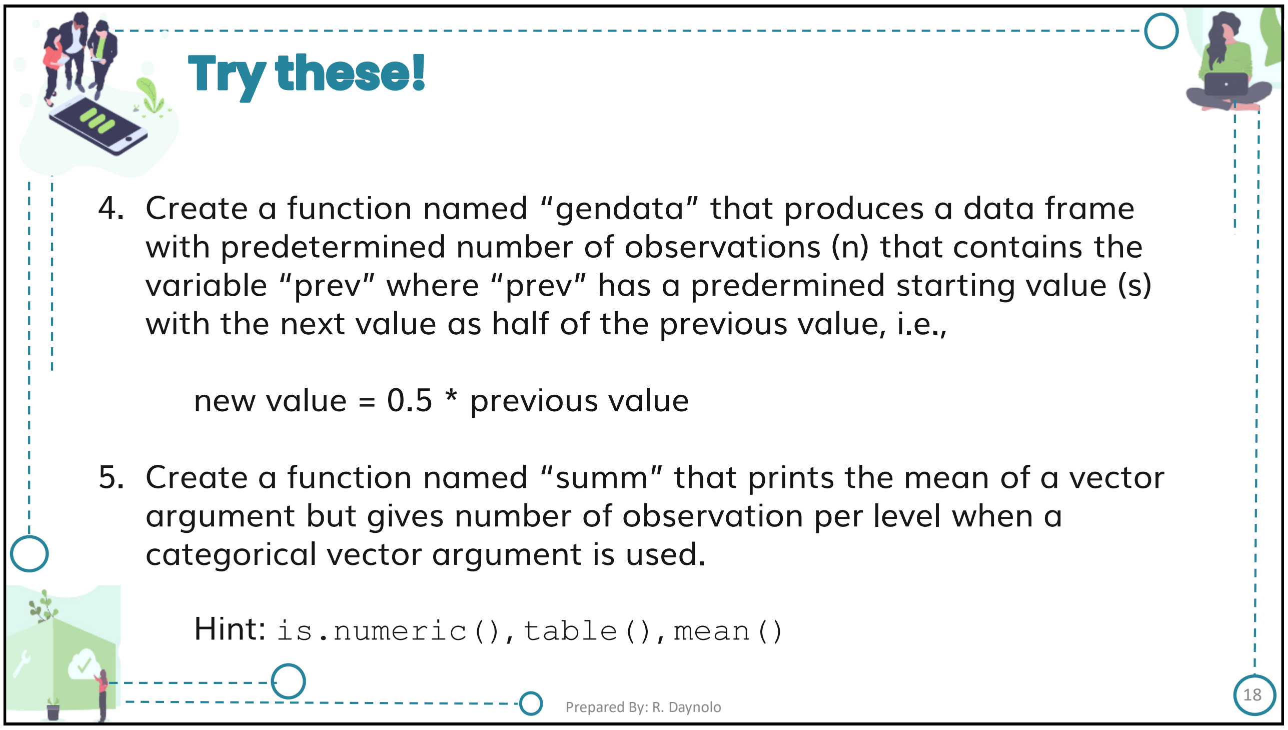
2. Create a function named "sqroot" that produces the square root of a valid numerical argument but produces the message "ERROR: Invalid argument" otherwise.

3. Create a function named "sq" that prints every perfect squares until the specified number of observations (n), e.g. 1,2,9,..., n^2

Prepared By: R. Daynolo

17

17



Try these!

4. Create a function named "gendata" that produces a data frame with predetermined number of observations (n) that contains the variable "prev" where "prev" has a predermined starting value (s) with the next value as half of the previous value, i.e.,

   new value = 0.5 * previous value

5. Create a function named "summ" that prints the mean of a vector argument but gives number of observation per level when a categorical vector argument is used.

   Hint: `is.numeric(), table(), mean()`

Prepared By: R. Daynolo

18

18

14. Scoping Rules

19

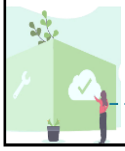
19

A Diversion on Binding Values to Symbol

How does R know which value to assign to which symbol? When I type,

```
> lm <- function(x){x*x}
> lm
function(x){x*x}
```

how does R know what value to assign to the symbol `lm`? Why doesn't it give it the value of `lm` that is in the ***stats*** package?



Prepared By: R. Daynolo

20

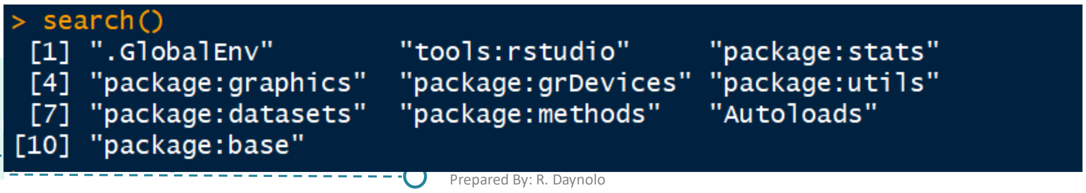
20

A Diversion on Binding Values to Symbol

When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly

1. Search the global environment for a symbol name matching the one requested.
2. Search the namespaces of each of the packages on the search list

The search list can be found by using the `search` function.



```
> search()
 [1] ".GlobalEnv"         "tools:rstudio"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

Binding Values to Symbol

- The *global environment* or the user's workspace is always the first element of the search list and the base package is always the last.
- The order of the packages on the search list matters!
- User's can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available.
- When a user loads a package with `library` the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions so it's possible to have an object named c and a function named c.

Scoping Rules

The scoping rules for R are the main feature that make it different from the original S language.

- The scoping rules determine how a value is associated with a **free variable** in a function
- R uses **lexical scoping** or **static scoping.** A common alternative is **dynamic scoping.**
- Related to the scoping rules is how R uses the search list to bind a value to a symbol
- Lexical scoping turns out to be particularly useful for simplifying statistical computations

Lexical Scoping

Consider the following function.

```
f <- function(x,y) {
  x^2 + y / z
}
```

This function has 2 formal arguments `x` and `y`. In the body of the function there is another symbol `z`. In this case `z` is called a ***free variable***. The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned inside the function body).

Lexical Scoping

Lexical Scoping in R means that

the values of free variables are searched for in the environment in which the function was defined.

What is an environment?

- An environment is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value.
- Every environment has a parent environment; it is possible for an environment to have multiple "children"
- the only environment without a parent is the empty environment
- A function + an environment = a *closure* or *function closure*.

Lexical Scoping

Searching for the value for a free variable:

- If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment.
- The search continues down the sequence of parent environments until we hit the top-level environment; this usually the global environment (workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the empty environment. If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

Lexical Scoping

Why does all this matter?

- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace
- This behavior is logical for most people and is usually the "right thing" to do
- However, in R you can have functions defined *inside other functions*
  - Languages like C don't let you do this
- Now things get interesting — In this case the environment in which a function is defined is the body of another function!

Lexical Scoping

```
make.power <- function(n) {
  pow <- function(x) {
    x^n
  }
  pow
}
```

This function returns another function as its value

```
> cube <- make.power(3)
> square <- make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Exploring a Function Closure

What's in a function's environment?

```
> ls(environment(cube))
[1] "n"   "pow"
> get("n", environment(cube))
[1] 3
> ls(environment(square))
[1] "n"   "pow"
> get("n", environment(square))
[1] 2
```

Prepared By: R. Daynolo

29

29

Lexical vs Dynamic Scoping

```
y <- 10

f <- function(x){
  y <- 2
  y^2 + g(x)
}

g <- function(x){
  x*y
}
```

What is the value of

```
> f(3)
```

Prepared By: R. Daynolo

30

30

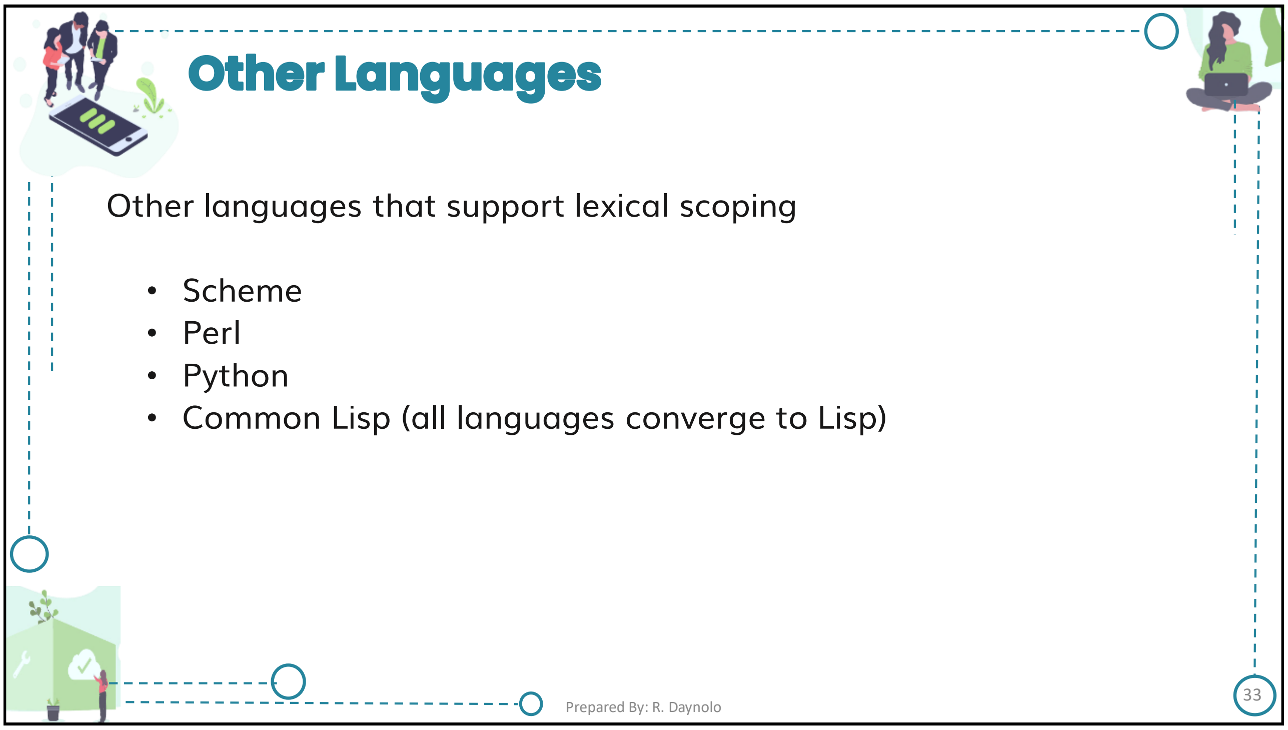
Lexical vs Dynamic Scoping

- With lexical scoping the value of `y` in the function `g` is looked up in the environment in which the function was defined, in this case the global environment, so the value of `y` is `10`.
- With dynamic scoping, the value of `y` is looked up in the environment from which the function was called (sometimes referred to as the ***calling environment***).
  - In R the calling environment is known as the ***parent frame***
- So the value of `y` would be `2`.

Lexical vs Dynamic Scoping

When a function is *defined* in the global environment and is subsequently *called* from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping.

```
> g <- function(x) {
+   a <- 3
+   x + a + y
+ }
> g(2)
Error in g(2) : object 'y' not found
> y <- 3
> g(2)
[1] 8
```

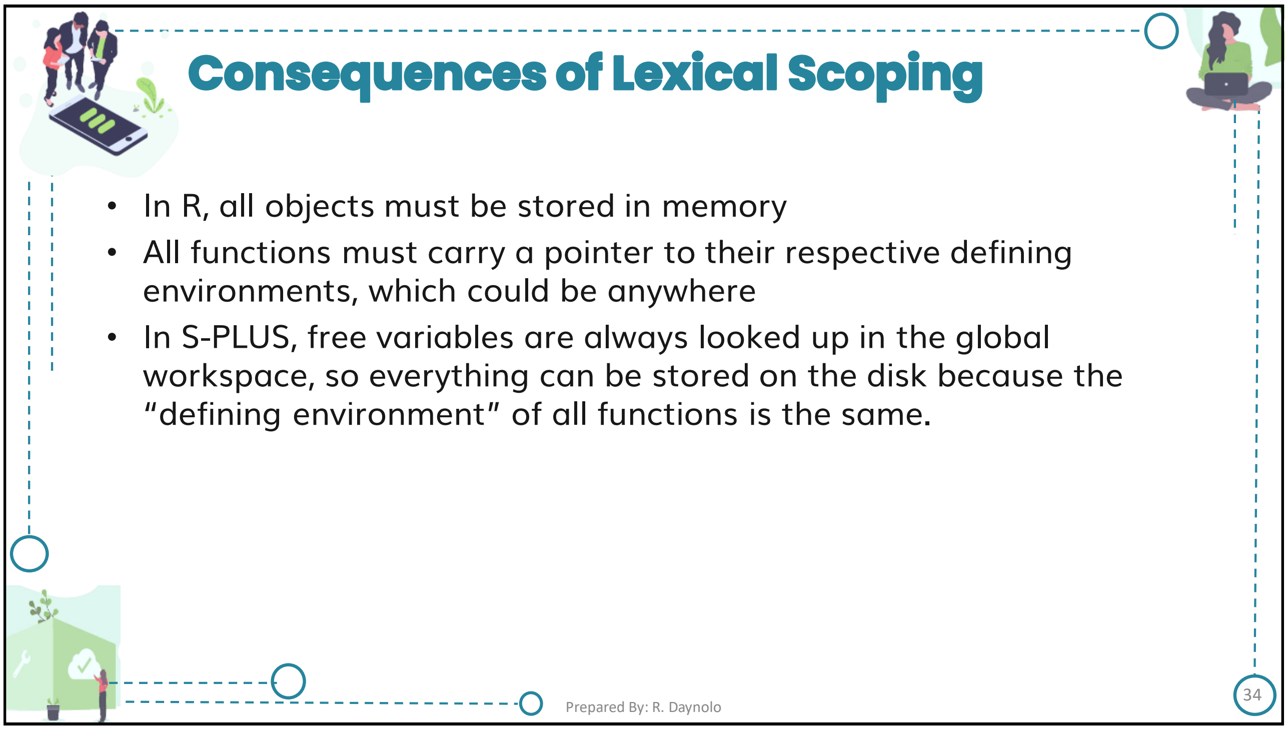


Other Languages

Other languages that support lexical scoping

- Scheme
- Perl
- Python
- Common Lisp (all languages converge to Lisp)

33



Consequences of Lexical Scoping

- In R, all objects must be stored in memory
- All functions must carry a pointer to their respective defining environments, which could be anywhere
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment" of all functions is the same.

34

Application: Optimization

Why is any of this information useful?

- Optimization routines in R like `optim`, `nlm`, and `optimize` require you to pass a function whose argument is a vector of parameters (e.g. a log-likelihood)
- However, an object function might depend on a host of other things besides its parameters (like *data*)
- When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed

Prepared By: R. Daynolo

35

35

Maximizing a Normal Likelihood

Write a "constructor" function

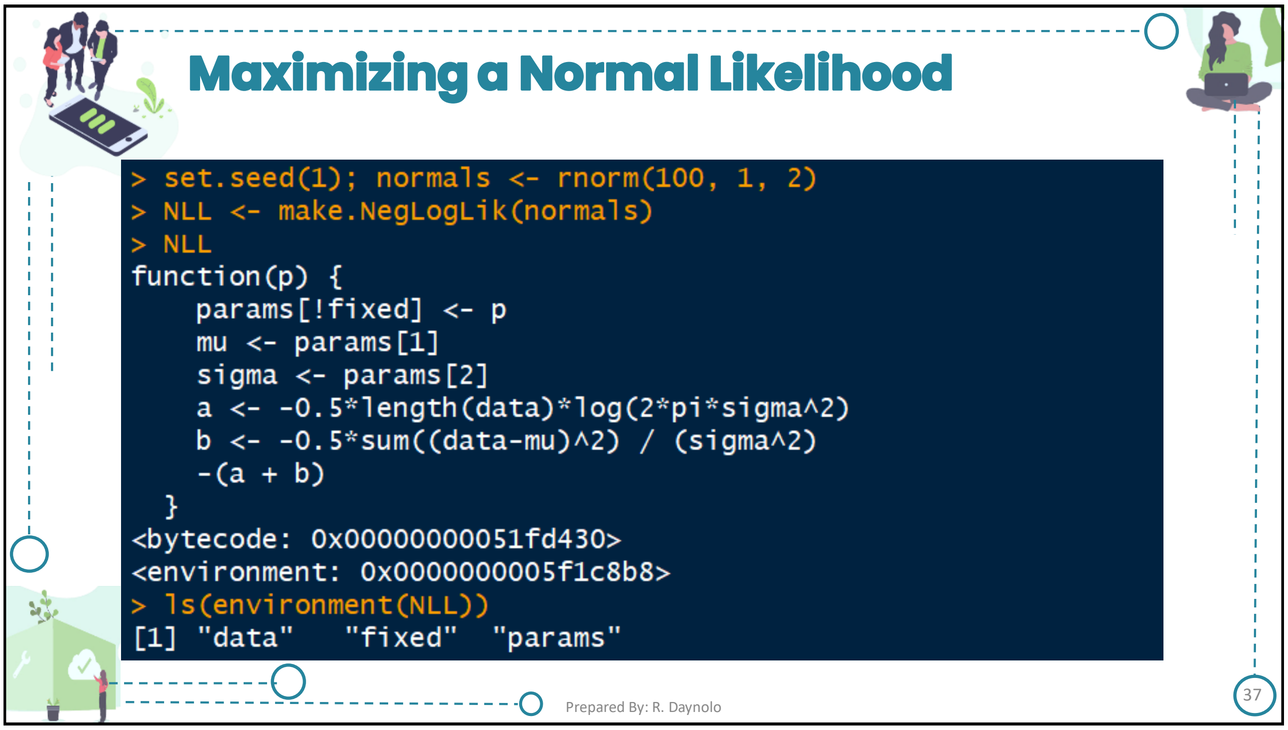
```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
  params <- fixed
  function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + b)
  }
}
```

Note: Optimization functions in R minimize functions, so you need to use the negative log-likelihood.

Prepared By: R. Daynolo

36

36



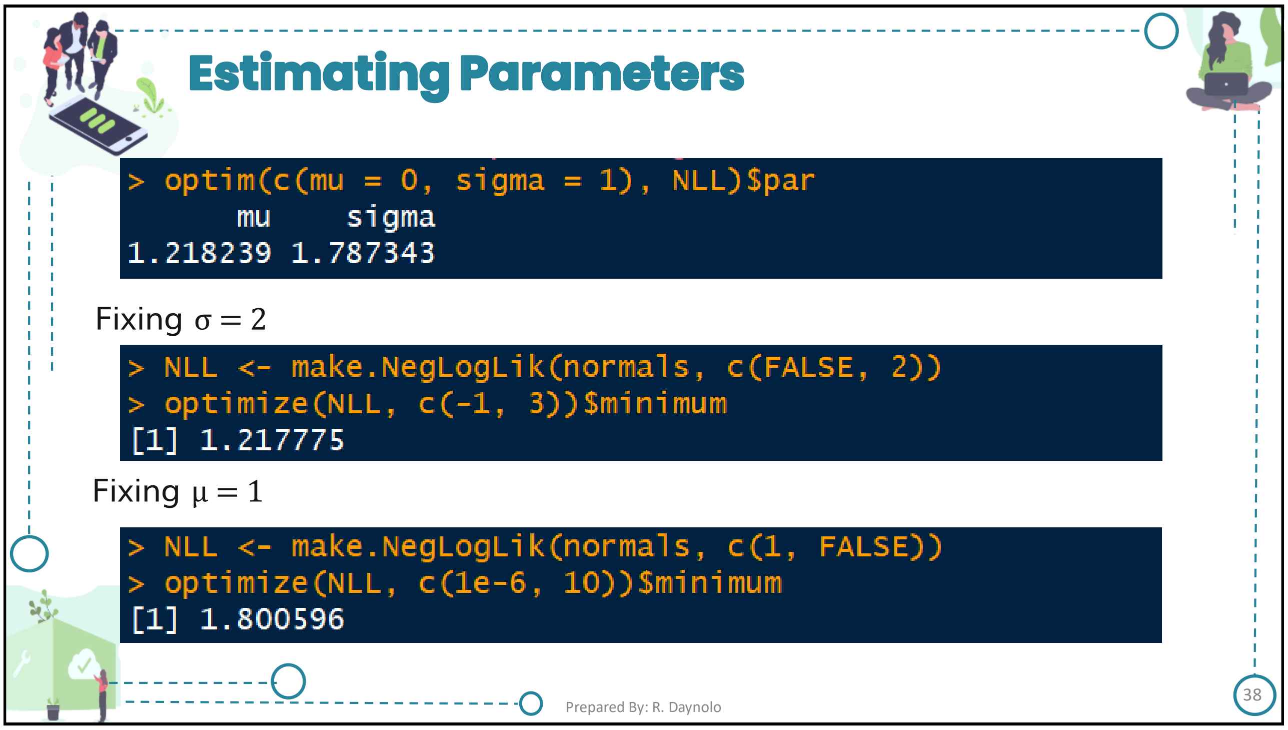
Maximizing a Normal Likelihood

```
> set.seed(1); normals <- rnorm(100, 1, 2)
> NLL <- make.NegLogLik(normals)
> NLL
function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + b)
  }
<bytecode: 0x00000000051fd430>
<environment: 0x0000000005f1c8b8>
> ls(environment(NLL))
[1] "data"   "fixed"  "params"
```

Prepared By: R. Daynolo

37

37



Estimating Parameters

```
> optim(c(mu = 0, sigma = 1), NLL)$par
      mu    sigma
1.218239 1.787343
```

Fixing $\sigma = 2$

```
> NLL <- make.NegLogLik(normals, c(FALSE, 2))
> optimize(NLL, c(-1, 3))$minimum
[1] 1.217775
```

Fixing $\mu = 1$

```
> NLL <- make.NegLogLik(normals, c(1, FALSE))
> optimize(NLL, c(1e-6, 10))$minimum
[1] 1.800596
```

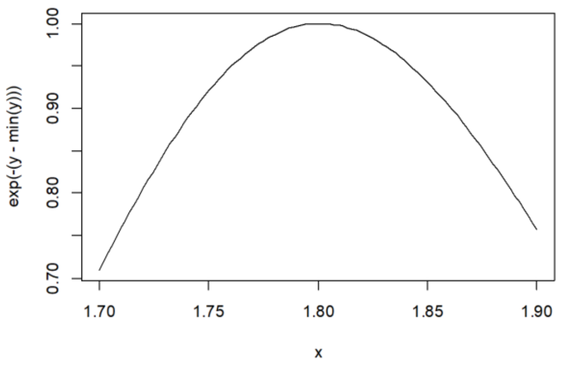
Prepared By: R. Daynolo

38

38

Plotting the Likelihood

```
NLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, NLL)
plot(x, exp(-(y - min(y))), type = "l")
```



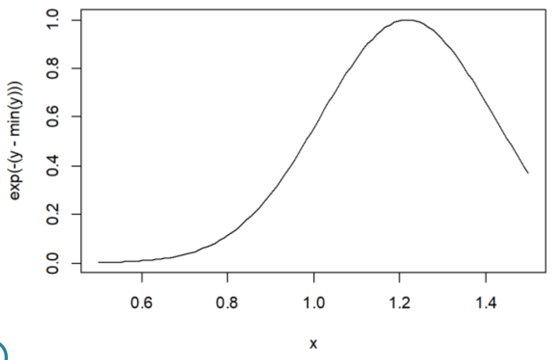
Prepared By: R. Daynolo

39

39

Plotting the Likelihood

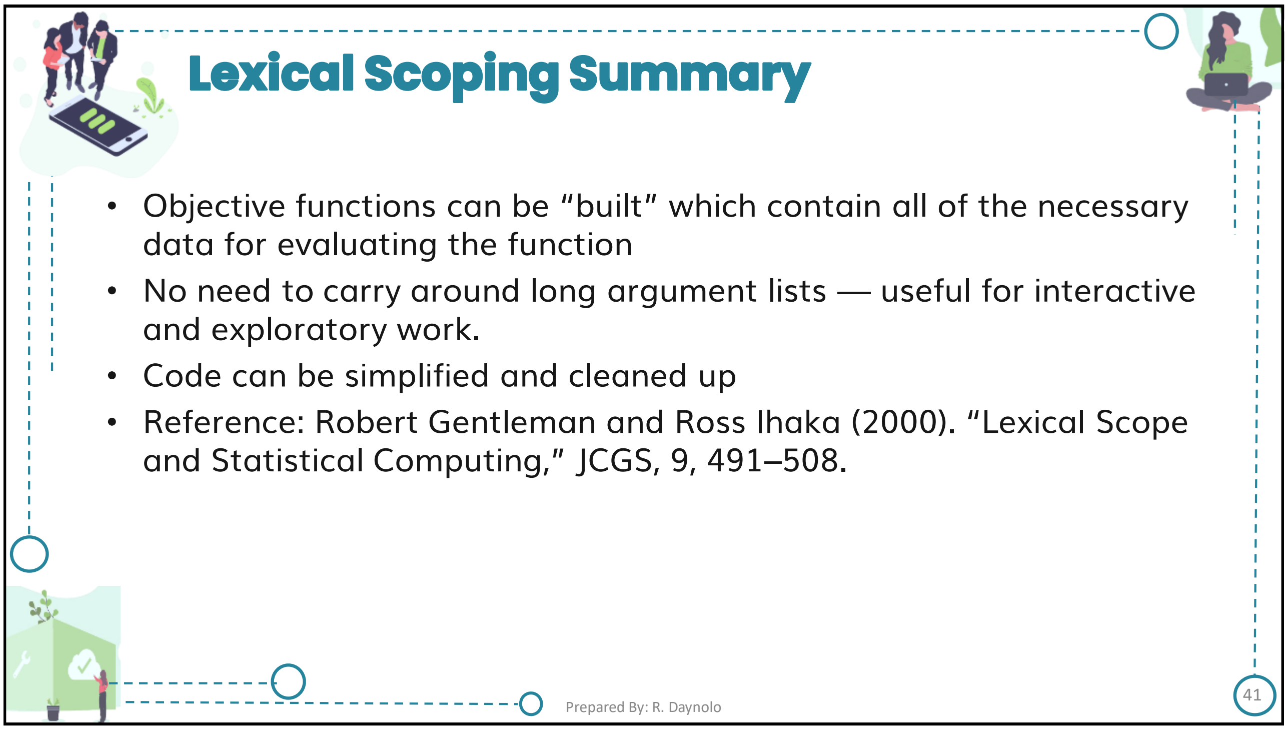
```
NLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, NLL)
plot(x, exp(-(y - min(y))), type = "l")
```



Prepared By: R. Daynolo

40

40



Lexical Scoping Summary

- Objective functions can be "built" which contain all of the necessary data for evaluating the function
- No need to carry around long argument lists — useful for interactive and exploratory work.
- Code can be simplified and cleaned up
- Reference: Robert Gentleman and Ross Ihaka (2000). "Lexical Scope and Statistical Computing," JCGS, 9, 491–508.

Prepared By: R. Daynolo

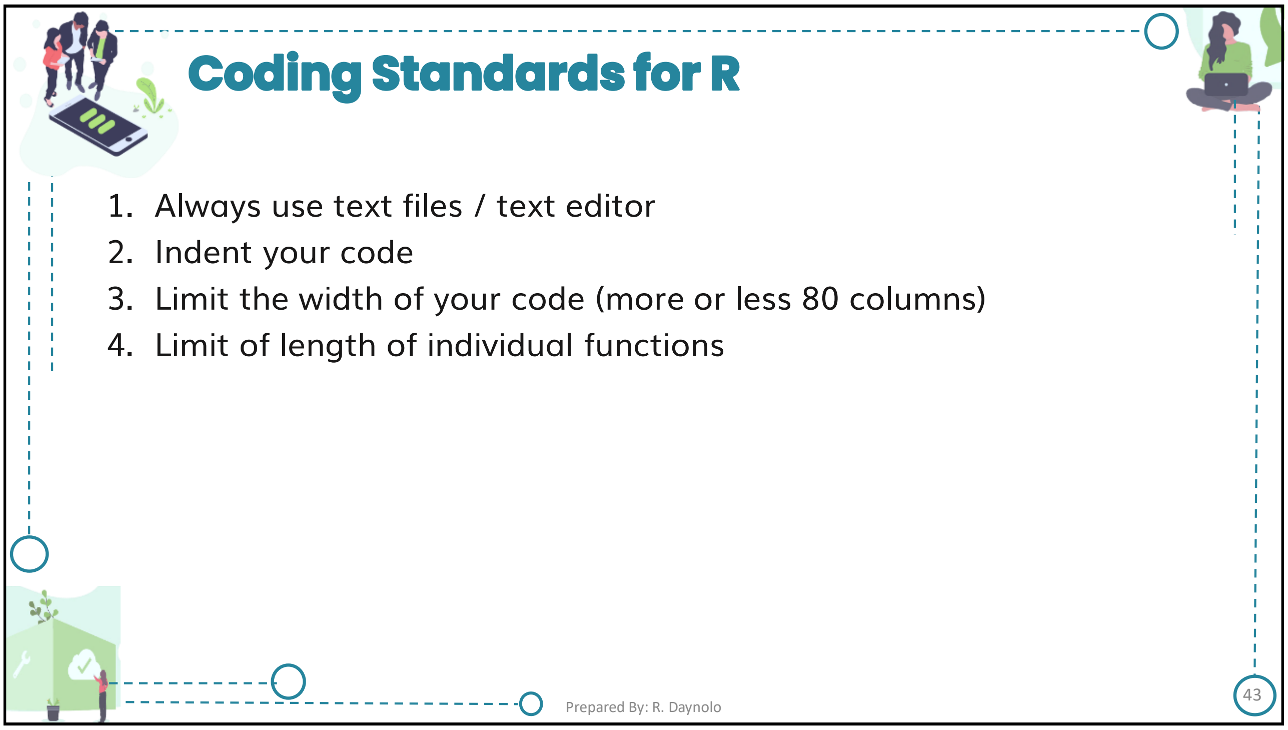
41

41

Coding Standards for R

42

42



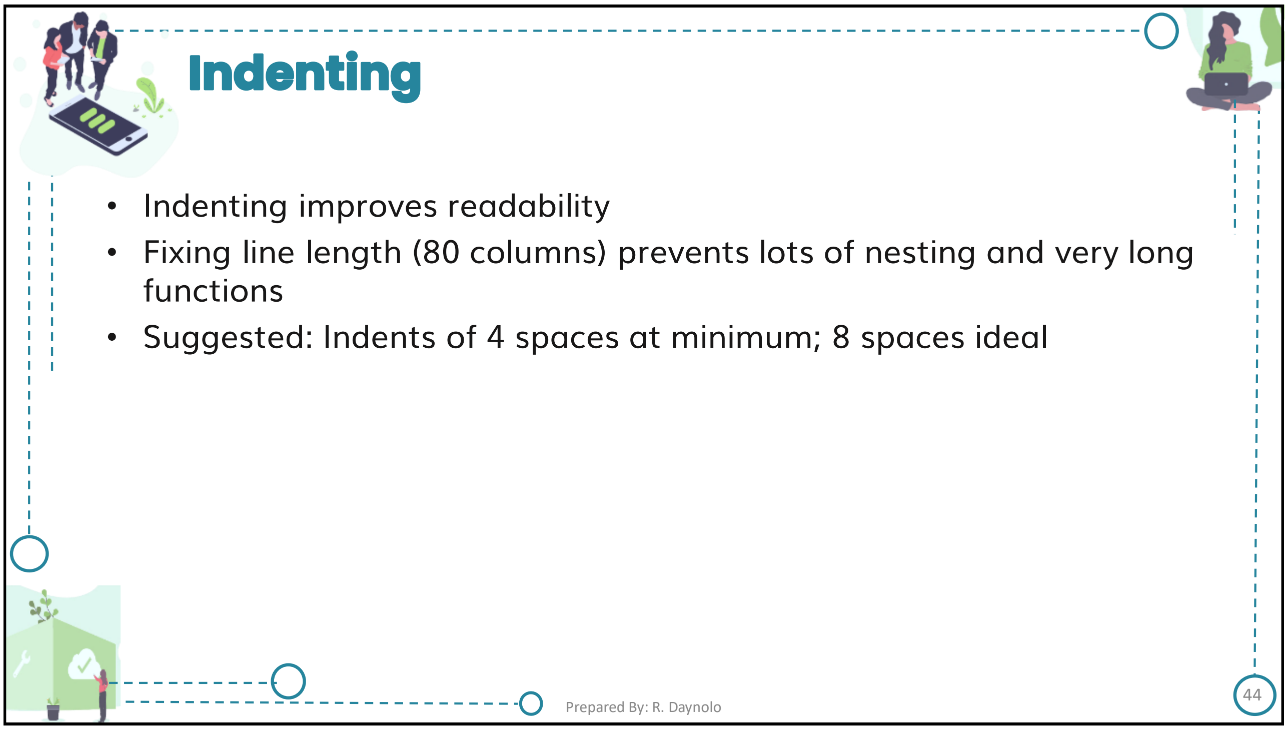
Coding Standards for R

1. Always use text files / text editor
2. Indent your code
3. Limit the width of your code (more or less 80 columns)
4. Limit of length of individual functions

Prepared By: R. Daynolo

43

43



Indenting

- Indenting improves readability
- Fixing line length (80 columns) prevents lots of nesting and very long functions
- Suggested: Indents of 4 spaces at minimum; 8 spaces ideal

Prepared By: R. Daynolo

44

44

Dates and Times in R

Dates and Times in R

R has developed a special representation of dates and times

- Dates are represented by the `Date` class
- Times are represented by the `POSIXct` or the `POSIXlt` class
- Dates are stored internally as the number of days since 1970-01-01
- Times are stored internally as the number of seconds since 1970-01-01

Dates in R

Dates are represented by the `Date` class and can be coerced from a character string using the `as.Date()` function.

```
> x <- as.Date("1970-01-01")
> x
[1] "1970-01-01"
> unclass(x)
[1] 0
> unclass(as.Date("1970-01-02"))
[1] 1
```

Times in R

Times are represented using the `POSIXct` or the `POSIXlt` class

- `POSIXct` is just a very large integer under the hood; it use a useful class when you want to store times in something like a data frame
- `POSIXlt` is a list underneath and it stores other useful information like the day of the week, day of the year, month, day of the month

Times in R

There are a number of generic functions that work on dates and times

- `weekdays`: give the day of the week
- `months`: give the month name
- `quarters`: give the quarter number (“Q1”, “Q2”, “Q3”, or “Q4”)

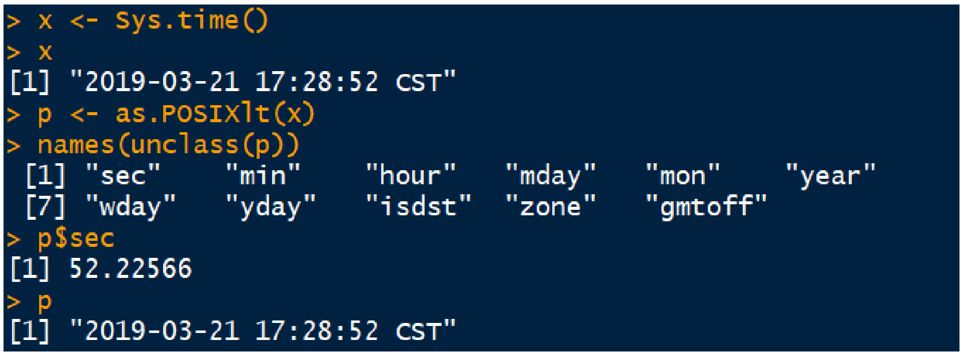
Prepared By: R. Daynolo

49

49

Times in R

Times can be coerced from a character string using the `as.POSIXlt` or `as.POSIXct` function.



```
> x <- Sys.time()
> x
[1] "2019-03-21 17:28:52 CST"
> p <- as.POSIXlt(x)
> names(unclass(p))
 [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"
 [7] "wday"   "yday"   "isdst"  "zone"   "gmtoff"
> p$sec
[1] 52.22566
> p
[1] "2019-03-21 17:28:52 CST"
```

Prepared By: R. Daynolo

50

50

Times in R

You can also use `POSIXct` format.

```
> x
[1] "2019-03-21 17:28:52 CST"
> unclass(x)
[1] 1553160532
> x$sec
Error in x$sec : $ operator is invalid for atomic vectors
> p <- as.POSIXlt(x)
> p$sec
[1] 52.22566
```

Times in R

Finally, there is the `strptime` function in case your dates are written in a different format

```
> datestring <- c("March 1, 2018 10:30", "April 24, 2019 9:10")
> y <- strptime(datestring, "%B %d, %Y %H:%M")
> y
[1] "2018-03-01 10:30:00 CST"
[2] "2019-04-24 09:10:00 CST"
> class(y)
[1] "POSIXlt" "POSIXt"
```

I can never remember the formatting strings. Check `?strptime` for details.

Operations on Dates and Times

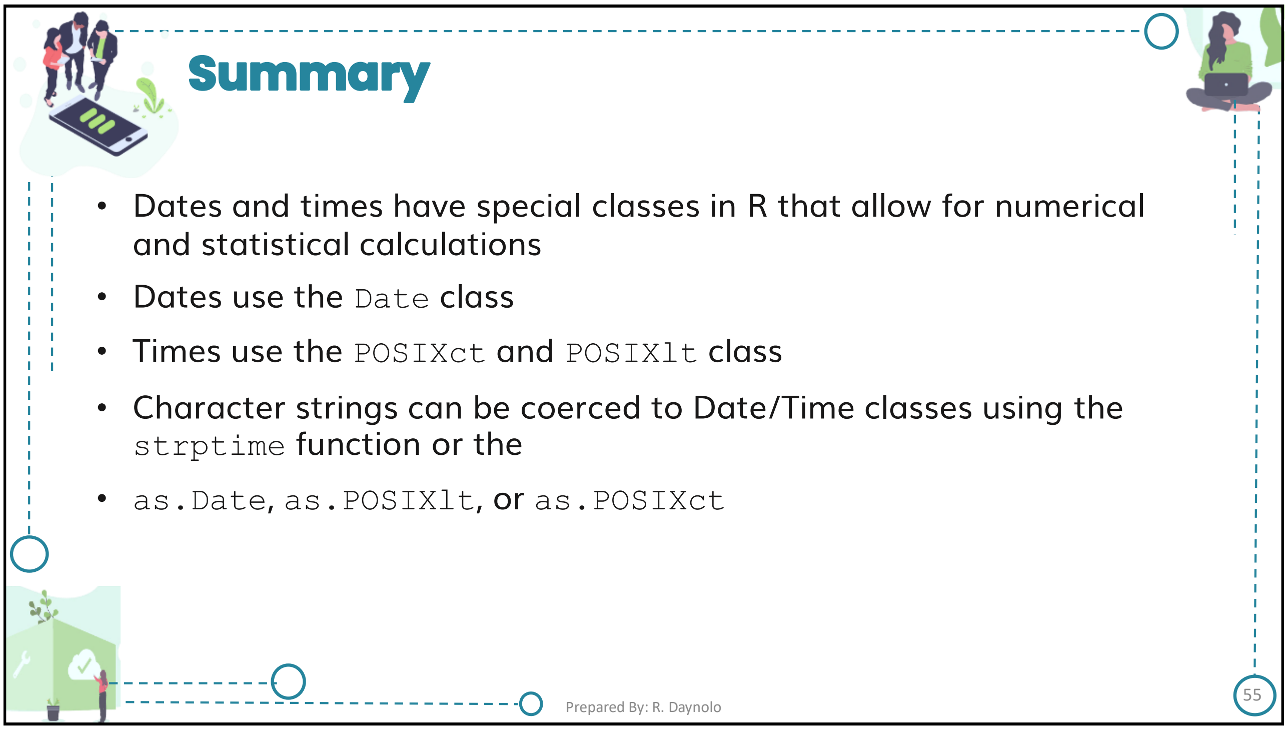
You can use mathematical operations on dates and times (well, just + and -). You can do comparisons too (==, <=)

```
> a <- as.Date("2012-01-01")
> b <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
> a-b
Error in a - b : non-numeric argument to binary operator
In addition: Warning message:
Incompatible methods ("-.Date", "-.POSIXt") for "-"
> a <- as.POSIXlt(a)
> a-b
Time difference of 356.8511 days
```

Operations on Dates and Times

Even keeps track of leap years, leap seconds, daylight savings, and time zones

```
> q <- as.Date("2012-03-01")
> w <- as.Date("2012-02-28")
> q-w
Time difference of 2 days
> q <- as.POSIXct("2012-10-25 01:00:00")
> w <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
> w-q
Time difference of 13 hours
```



Summary

- Dates and times have special classes in R that allow for numerical and statistical calculations
- Dates use the `Date` class
- Times use the `POSIXct` and `POSIXlt` class
- Character strings can be coerced to Date/Time classes using the `strptime` function or the
- `as.Date`, `as.POSIXlt`, or `as.POSIXct`

Prepared By: R. Daynolo

55

55