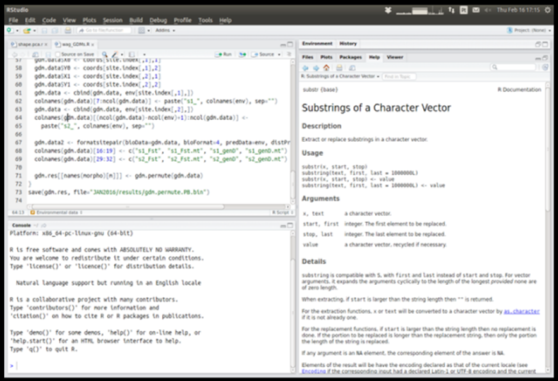
ADVANCED COURSE

INTRODUCTION TO R

FUNCTIONS AND PACKAGES



```
gdn.data$Y0 <- coords[site.index[,1],2]
gdn.data$X1 <- coords[site.index[,2],1]
gdn.data$Y1 <- coords[site.index[,2],2]
gdn.data <- cbind(gdn.data, env[site.index[,1],])
colnames(gdn.data)[7:ncol(gdn.data)] <- paste("s1_", colnames(env), sep="")
gdn.data <- cbind(gdn.data, env[site.index[,2],])
colnames(gdn.data)[(ncol(gdn.data)-ncol(env)+1):ncol(gdn.data)] <-
  paste("s2_", colnames(env), sep="")

gdn.data2 <- formatsitepair(bioData=gdn.data, bioFormat=4, predData=env, distPr
colnames(gdn.data)[16:19] <- c("s1_Fst", "s1_Fst.mt", "s1_genD", "s1_genD.mt")
colnames(gdn.data)[29:32] <- c("s2_Fst", "s2_Fst.mt", "s2_genD", "s2_genD.mt")

gdn.res[[names(morpho)[m]]] <- gdn.permute(gdn.data)
}
save(gdn.res, file="JAN2016/results/gdn.permute.PB.bin")
```

```
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

substr {base} R Documentation

Substrings of a Character Vector

Description

Extract or replace substrings in a character vector.

Usage

```
substr(x, start, stop)
substring(text, first, last = 1000000L)
substr(x, start, stop) <- value
substring(text, first, last = 1000000L) <- value
```

Arguments

x, text	a character vector.
start, first	integer. The first element to be replaced.
stop, last	integer. The last element to be replaced.
value	a character vector, recycled if necessary.

Details

substring is compatible with S, with first and last instead of start and stop. For vector arguments, it expands the arguments cyclically to the length of the longest *provided* none are of zero length.

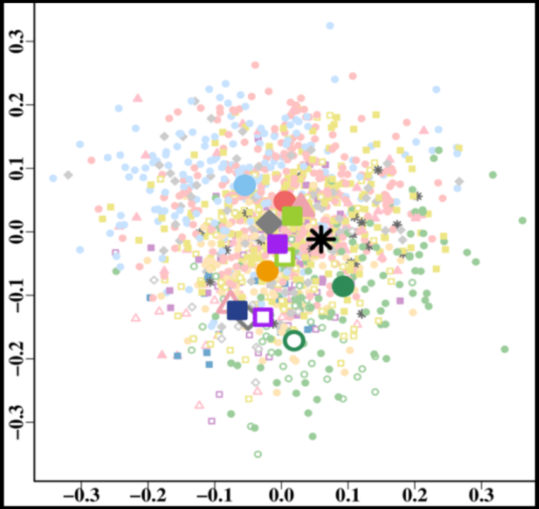
When extracting, if start is larger than the string length then "" is returned.

For the extraction functions, x or text will be converted to a character vector by as.character if it is not already one.

For the replacement functions, if start is larger than the string length then no replacement is done. If the portion to be replaced is longer than the replacement string, then only the portion the length of the string is replaced.

If any argument is an NA element, the corresponding element of the answer is NA.

Elements of the result will be have the encoding declared as that of the current locale (see Encoding if the corresponding input had a declared Latin-1 or UTF-8 encoding and the current



Antigoni Kaliontzopoulou

4 - 8 November 2019| CIBIO-InBIO, Vairão, Portugal

Everything that exists is an object
Everything that happens is a function call

- John Chambers

- Everything you do in R is organized into functions
- Functions are pieces of code, intended to fulfill... a function!
- Objective: code abstraction and shortening. When using a function, you do not need to see all of its code (but you may want to see the related documentation, see further on)

- They can be thought of as recipes for operations you repeat a lot

  e.g. Imagine you always want to add three numbers and divide by three

```
> x <- 15
> y <- 23
> z <- 36
> sum.all <- x + y + z
> sum.div.3 <- sum.all/3
> sum.div.3
[1] 24.66667
```

  This would be much faster to repeat if we turned it into a function

```
> sum.div.3 <- function(x, y, z) {
+ sum.all <- x + y + z
+ sum.div.3 <- sum.all/3
+ return(sum.div.3)
+ }

sum.div.3(15, 23, 36)
```

- We can write our own functions (see end of the week)

- Mostly we use built-in R functions
- These are downloaded from CRAN (see packages further on)

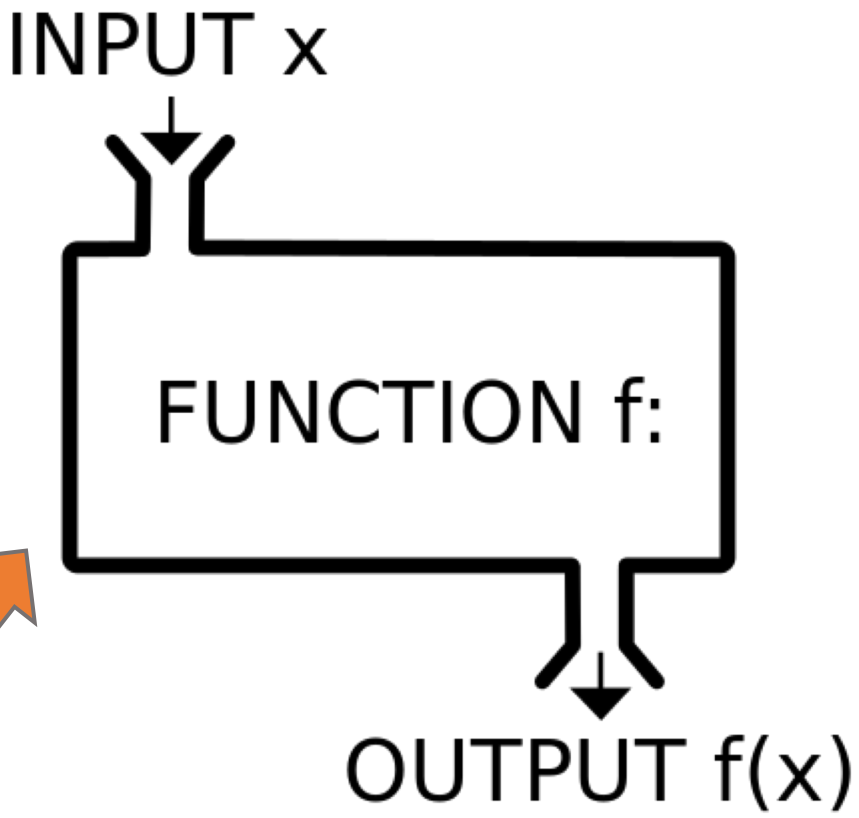
- Functions are objects on their own right
- But functions and objects are different things

- When calling a function as an object, we see its content. In some cases we can see the associated code, in others the code is “hidden”

- IMPORTANT NOTICE: using built-in function names to name objects can cause you ***important trouble***!!!

The object(s) that we want to “manipulate” (and maybe some conditions on how to do it)

ARGUMENT(S)



The “recipe” of what we want to do
The code of the function

FUNCTION OBJECT

The result of implementing the function f on object x, with the defined arguments

- Various types of functions

  → In terms of origin (i.e. built-in, user-defined, from packages)

  → Arithmetic, statistical, plotting etc

  → Informative about objects

  → Informative about our workspace or system

- **Primitive** functions, are functions that directly call C code, and they have no underlying R code associated

- Once the function is applied, the result is shown in our screen, but it is not stored
- You need to store function results into a new object

- Functions can be nested one inside the other
- In long series of nested functions, always read “inside-out”

- **Function arguments**
  - Can be defined by position (in order)

    by complete name (e.g. method = A)

    by partial name (e.g. meth = A)

```
> mean(1:100, 0.025, T)
[1] 50.5
> mean(x = 1:100, trim = 0.025, na.rm = T)
[1] 50.5
> mean(x = 1:100, t = 0.025, n= T)
[1] 50.5
```

  - Some arguments have default values
  - Be **VERY** careful with defaults:

    when you **do not** set an argument, you will be using the default

  - **…** is a special argument that allows us to pass arguments not passed to the function used, but used inside it through other functions (see e.g. arguments from par() passed to plot(), Wednesday)

- Mostly (at least at the beginning) you will use built-in functions
- Functions are organized in **PACKAGES**
- There is a huge series of packages, for the full list see:

  https://cran.r-project.org/web/packages/

- To use a package, you need to:
  1. Install it (only once, unless you update R)

     ```
     > install.packages(“vegan”)
     ```

  2. Load it (in each session that you need it)

     ```
     > library(“vegan”)
     ```

- A useful command: installed.packages()

- Getting help:
  - ?function_name (e.g. ?mean)
  - ?package_name (e.g. ?vegan)
  - Search in the "help" tab of Rstudio

    Exact match

  - ??some_useful_term

    → Fuzzy matching, more flexible, depends on installed packages

- Function help pages may contain
  - **Description:** verbiage on what the function does
  - **Usage:** how the function is used (argument order and defaults, see next)
  - **Arguments:** description of the arguments
  - **Details:** more relevant info on argument options
  - **Value:** description of the returned object (class, content etc)
  - **References**
  - **See also:** other relevant functions (similar, related, complementary etc.)
  - **Examples**

mean {base} ← **Package in which the function belongs**

R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`
: An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.

`trim`
: the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

`na.rm`
: a logical value indicating whether `NA` values should be stripped before the computation proceeds.

`...`
: further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`weighted.mean`, `mean.POSIXct`, `colMeans` for row and column means.

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

Why should WE care?

- ✓ R includes endless packages
- ✓ Basically, any biological data analysis you might think about can be done in R

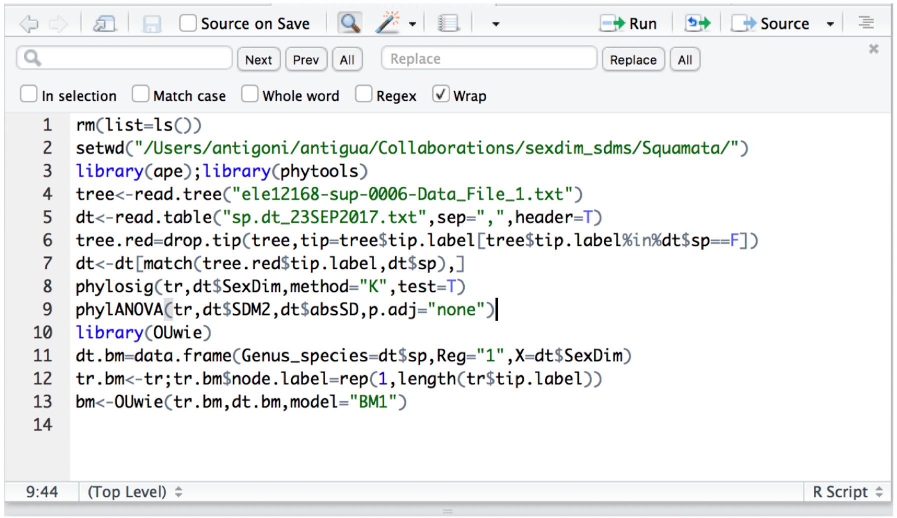
  https://cran.r-project.org/web/packages/available_packages_by_name.html

- ✓ Many basic, general purpose analyses are implemented in packages base and stats, which are available by default with any R installation

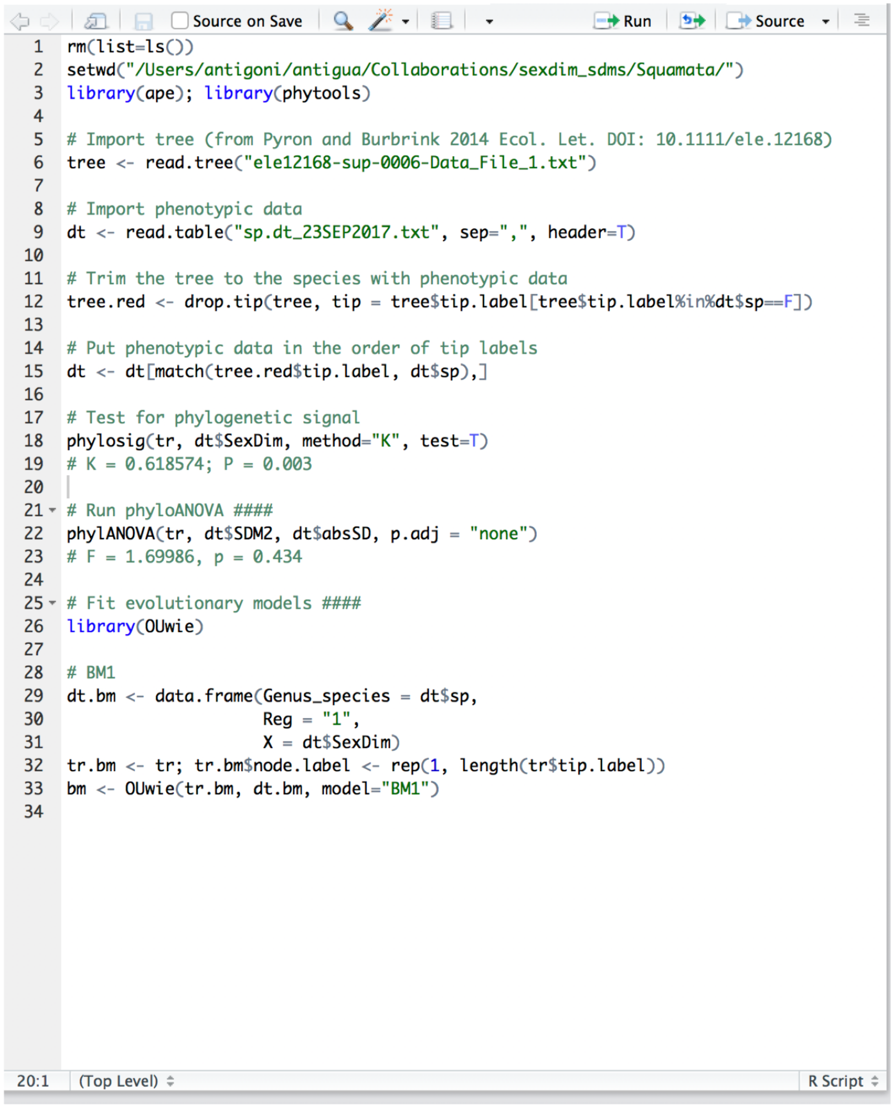
  base: https://stat.ethz.ch/R-manual/R-devel/library/base/html/00Index.html

  stats: https://stat.ethz.ch/R-manual/R-devel/library/stats/html/00Index.html

- ✓ Typically, we use a series of functions to perform everyday operations with our data
- ✓ It is VERY useful to adopt good code-writing practices from the beggining



```r
rm(list=ls())
setwd("/Users/antigoni/antigua/Collaborations/sexdim_sdms/Squamata/")
library(ape);library(phytools)
tree<-read.tree("ele12168-sup-0006-Data_File_1.txt")
dt<-read.table("sp.dt_23SEP2017.txt",sep=",",header=T)
tree.red=drop.tip(tree,tip=tree$tip.label[tree$tip.label%in%dt$sp==F])
dt<-dt[match(tree.red$tip.label,dt$sp),]
phylosig(tr,dt$SexDim,method="K",test=T)
phylANOVA(tr,dt$SDM2,dt$absSD,p.adj="none")
library(OUwie)
dt.bm=data.frame(Genus_species=dt$sp,Reg="1",X=dt$SexDim)
tr.bm<-tr;tr.bm$node.label=rep(1,length(tr$tip.label))
bm<-OUwie(tr.bm,dt.bm,model="BM1")

```



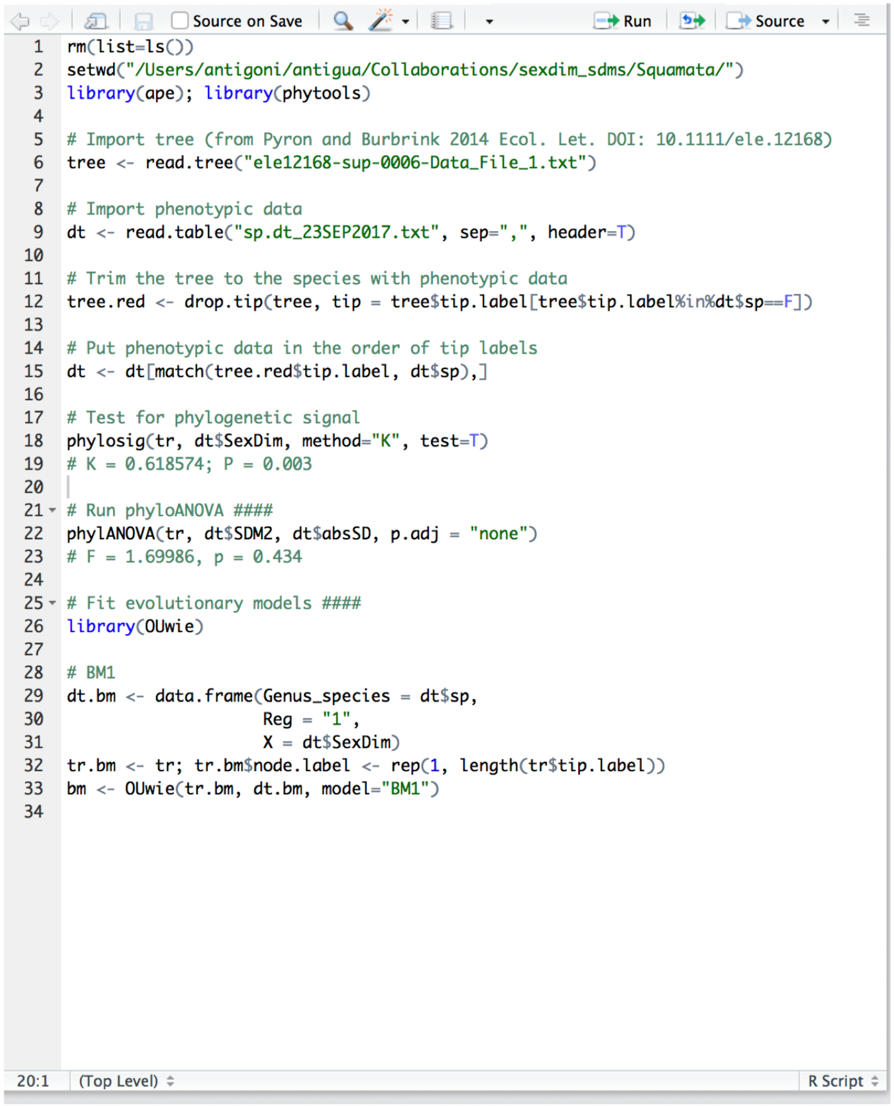
```r
rm(list=ls())
setwd("/Users/antigoni/antigua/Collaborations/sexdim_sdms/Squamata/")
library(ape); library(phytools)

# Import tree (from Pyron and Burbrink 2014 Ecol. Let. DOI: 10.1111/ele.12168)
tree <- read.tree("ele12168-sup-0006-Data_File_1.txt")

# Import phenotypic data
dt <- read.table("sp.dt_23SEP2017.txt", sep=",", header=T)

# Trim the tree to the species with phenotypic data
tree.red <- drop.tip(tree, tip = tree$tip.label[tree$tip.label%in%dt$sp==F])

# Put phenotypic data in the order of tip labels
dt <- dt[match(tree.red$tip.label, dt$sp),]

# Test for phylogenetic signal
phylosig(tr, dt$SexDim, method="K", test=T)
# K = 0.618574; P = 0.003

# Run phyloANOVA ####
phylANOVA(tr, dt$SDM2, dt$absSD, p.adj = "none")
# F = 1.69986, p = 0.434

# Fit evolutionary models ####
library(OUwie)

# BM1
dt.bm <- data.frame(Genus_species = dt$sp,
                    Reg = "1",
                    X = dt$SexDim)
tr.bm <- tr; tr.bm$node.label <- rep(1, length(tr$tip.label))
bm <- OUwie(tr.bm, dt.bm, model="BM1")

```

- ✓ Typically, we use a series of functions to perform everyday operations with our data
- ✓ It is VERY useful to adopt good code-writing practices from the beggining

- ✓ For you:
  - easier to remember what you did (and possibly why)
  - lighter on your brain
  - easier to debug
  - easier to modify
  - easier to translate into the "Statistical analyses" section of your paper

- ✓ For collaborators: good code style makes it easier to share and interchange ideas

- ✓ For reviewers: nowadays many top-rank journals require the R-code for review during article submission



```r
rm(list=ls())
setwd("/Users/antigoni/antigua/Collaborations/sexdim_sdms/Squamata/")
library(ape); library(phytools)

# Import tree (from Pyron and Burbrink 2014 Ecol. Let. DOI: 10.1111/ele.12168)
tree <- read.tree("ele12168-sup-0006-Data_File_1.txt")

# Import phenotypic data
dt <- read.table("sp.dt_23SEP2017.txt", sep=",", header=T)

# Trim the tree to the species with phenotypic data
tree.red <- drop.tip(tree, tip = tree$tip.label[tree$tip.label%in%dt$sp==F])

# Put phenotypic data in the order of tip labels
dt <- dt[match(tree.red$tip.label, dt$sp),]

# Test for phylogenetic signal
phylosig(tr, dt$SexDim, method="K", test=T)
# K = 0.618574; P = 0.003

# Run phyloANOVA ####
phylANOVA(tr, dt$SDM2, dt$absSD, p.adj = "none")
# F = 1.69986, p = 0.434

# Fit evolutionary models ####
library(OUwie)

# BM1
dt.bm <- data.frame(Genus_species = dt$sp,
                    Reg = "1",
                    X = dt$SexDim)
tr.bm <- tr; tr.bm$node.label <- rep(1, length(tr$tip.label))
bm <- OUwie(tr.bm, dt.bm, model="BM1")

```

Google's R Style Guide - https://google.github.io/styleguide/Rguide.xml

The goal of the R Programming Style Guide is to make our R code easier to **read**, **share**, and **verify**.

File Names

File names should end in .R and, of course, be meaningful.

GOOD: predict_ad_revenue.R

BAD: foo.R

Identifiers

- ✓ Don't use underscores (_) or hyphens (-)
- ✓ The preferred form for **variable names** is all lower case letters and words separated with dots (variable.name), but variableName is also accepted

  GOOD: avg.clicks

  OK: avgClicks

  BAD: avg_Clicks

- ✓ **function names** have initial capital letters and no dots (FunctionName)

  GOOD: CalculateAvgClicks

  BAD: calculate_avg_clicks , calculateAvgClicks

- ✓ **constants** are named like functions but with an initial k

  kConstantName

Syntax

- ✓ Keep maximum **line length** to 80 characters (avoid very long lines)
- ✓ When **indenting** your code, use two spaces. Never use tabs or mix tabs and spaces
- ✓ Place **spaces** around all binary operators (=, +, -, <-, etc.)
- ✓ Do not place a space before a **comma**, but always place one after a comma
- ✓ Extra spacing (i.e., more than one space in a row) is okay if it improves alignment of equals signs or arrows (<-)

Assignment

- ✓ **use ”<-”**, not ”=“, for assignment

  GOOD:

  ```
  x <- 5
  ```

  BAD:

  ```
  x = 5
  ```

  EVEN WORSE:

  ```
  x=5
  ```

Braces

- ✓ an **opening curly brace** should never go on its own line
- ✓ a **closing curly brace** should always go on its own line
- ✓ you may omit curly braces when a block consists of a single statement; however, you must *consistently* either use or not use curly braces for single statement blocks
- ✓ always begin the body of a block on a new line.

GOOD:

```
if (is.null(ylim)) {
    ylim <- c(0, 0.06)
}
```

or (but not both)

```
if (is.null(ylim))
    ylim <- c(0, 0.06)
```

BAD:

```
if (is.null(ylim)) ylim <- c(0, 0.06)
if (is.null(ylim)) {ylim <- c(0, 0.06)}
```

Braces (continued)

- ✓ surround **else** with braces
- ✓ an else statement should always be surrounded on the same line by curly braces

GOOD:
```
if (condition) {
    one or more lines
} else {
    one or more lines
}
```

BAD:
```
if (condition) {
    one or more lines
}
else {
    one or more lines
}
```

BAD:
```
if (condition)
    one line
else
    one line
```

General Layout and Ordering

If everyone uses the same general ordering, we'll be able to read and understand each other's scripts faster and more easily.

1. Copyright statement comment
2. Author comment
3. File description comment, including purpose of program, inputs, and outputs
4. source() and library() statements
5. Function definitions
6. Executed statements, if applicable (e.g., print, plot)

Commenting Guidelines

- ✓ Comment your code
- ✓ Comment your code!
- ✓ Comment your code!!
- ✓ Comment your code!!!
- ✓ **Comment your code!!!!!!!!!!!!!!!**
- ✓ Entire commented lines should begin with # and one space
- ✓ Short comments can be placed after code preceded by two spaces, #, and then one space

…and more ...

Google's R Style Guide - https://google.github.io/styleguide/Rguide.xml