Common questions (and answers) about *R*

David M. Rosenberg

Committee on Neurobiology
University of Chicago

October 20, 2009

Outline

Admin issues
Things to ask for

- **Email addresses:**
  If you use a non-University of Chicago email account, let me know either via an email from your `uchicago.edu` email account or a written note.

Admin issues
Things to ask for

- **Email addresses:**
  If you use a non-University of Chicago email account, let me know either via an email from your `uchicago.edu` email account or a written note.

- **Use of lab time:**
  What do you think would be the best use of your time in lab.
  - Status quo. Labs are principally time for you to try to work on the lab.
  - *Short* lectures. Presentations like this (but shorter; 10-20 minutes) at the start / end / middle of labs seems helpful.
  - Review of the previous week's assignment and / or group discussion of the different ways you solved the problems.
  - Discussion / question-and-answer time about the topics and concepts presented in lecture *not* directly pertaining to computational programming.

Admin issues
Things to ask for

- **Email addresses:**
  If you use a non-University of Chicago email account, let me know either via an email from your `uchicago.edu` email account or a written note.

- **Use of lab time:**
  What do you think would be the best use of your time in lab.
  - Status quo. Labs are principally time for you to try to work on the lab.
  - *Short* lectures. Presentations like this (but shorter; 10-20 minutes) at the start / end / middle of labs seems helpful.
  - Review of the previous week's assignment and / or group discussion of the different ways you solved the problems.
  - Discussion / question-and-answer time about the topics and concepts presented in lecture *not* directly pertaining to computational programming.

Admin issues
Things to ask for

- **Email addresses:**
  If you use a non-University of Chicago email account, let me know either via an email from your `uchicago.edu` email account or a written note.

- **Use of lab time:**
  What do you think would be the best use of your time in lab.
  - Status quo. Labs are principally time for you to try to work on the lab.
  - *Short* lectures. Presentations like this (but shorter; 10-20 minutes) at the start / end / middle of labs seems helpful.
  - Review of the previous week's assignment and / or group discussion of the different ways you solved the problems.
  - Discussion / question-and-answer time about the topics and concepts presented in lecture *not* directly pertaining to computational programming.

Admin issues
Things to ask for

- **Email addresses:**
  If you use a non-University of Chicago email account, let me know either via an email from your `uchicago.edu` email account or a written note.

- **Use of lab time:**
  What do you think would be the best use of your time in lab.
  - Status quo. Labs are principally time for you to try to work on the lab.
  - *Short* lectures. Presentations like this (but shorter; 10-20 minutes) at the start / end / middle of labs seems helpful.
  - Review of the previous week's assignment and / or group discussion of the different ways you solved the problems.
  - Discussion / question-and-answer time about the topics and concepts presented in lecture *not* directly pertaining to computational programming.

Admin issues
Things to ask for

- **Email addresses:**
  If you use a non-University of Chicago email account, let me know either via an email from your `uchicago.edu` email account or a written note.

- **Use of lab time:**
  What do you think would be the best use of your time in lab.
  - Status quo. Labs are principally time for you to try to work on the lab.
  - *Short* lectures. Presentations like this (but shorter; 10-20 minutes) at the start / end / middle of labs seems helpful.
  - Review of the previous week's assignment and / or group discussion of the different ways you solved the problems.
  - Discussion / question-and-answer time about the topics and concepts presented in lecture *not* directly pertaining to computational programming.

Admin issues
Things to provide

- **Homework examples & template:**
  Files on chalk.
- **Grades & comments:**
  By next week you will have exercises 0 and 1 returned to you.
- **Tools & utilities:**
  Source code "cleaner," problem uploader
- **Cheat sheets:**
  Math "cheat sheets" (i.e. trigonometric identities, tables of common integrals and derivatives, etc.) as well as "cheat sheets" for *R* are in progress.

Admin issues
Things to provide

- **Homework examples & template:**
  Files on chalk.
- **Grades & comments:**
  By next week you will have exercises 0 and 1 returned to you.
- **Tools & utilities:**
  Source code "cleaner," problem uploader
- **Cheat sheets:**
  Math "cheat sheets" (i.e. trigonometric identities, tables of common integrals and derivatives, etc.) as well as "cheat sheets" for *R* are in progress.

Admin issues
Things to provide

- **Homework examples & template:**
  Files on chalk.
- **Grades & comments:**
  By next week you will have exercises 0 and 1 returned to you.
- **Tools & utilities:**
  Source code “cleaner,” problem uploader
- **Cheat sheets:**
  Math “cheat sheets” (i.e. trigonometric identities, tables of common integrals and derivatives, etc.) as well as “cheat sheets” for *R* are in progress.

Admin issues
Things to provide

- **Homework examples & template:**
  Files on chalk.
- **Grades & comments:**
  By next week you will have exercises 0 and 1 returned to you.
- **Tools & utilities:**
  Source code “cleaner,” problem uploader
- **Cheat sheets:**
  Math “cheat sheets” (i.e. trigonometric identities, tables of common integrals and derivatives, etc.) as well as “cheat sheets” for *R* are in progress.

Outline

Manipulating Data - Vectors

Basic usage

- Rules
  - Single type
  - Accessed by index
  - Can be named or unnamed

```
>c('hello', TRUE); c(1, TRUE)
[1] "hello" "TRUE"
[1] 1 1
>class(c('hello', TRUE)); class(c(1, TRUE));
[1] "character"
[1] "numeric"
>c('Hello', 14);
[1] "Hello" "14"
>class(c(TRUE, TRUE));
[1] "logical"
```

```
>my_name <- c('D', 'a', 'v', 'i',
              'd', ' ', 'R', '.');
>length(my_name);
[1] 8
>names(my_name);
NULL
>names(my_name) <-
   c(  paste('first', as.character(1:5),
         sep="_"),
       'space',
       paste('last', as.character(1:2),
         sep="_")
    );
>my_name;
first_1 first_2 first_3 first_4 first_5
    "D"     "a"     "v"     "i"     "d"
  space  last_1  last_2
    " "     "R"     "."
```

Manipulating Data - Vectors

Basic usage

Altering vectors

Use `<-` to alter / overwrite an existing vector

- **Growing vectors**
  Use `c(vector, othervector, value, ...)` to add to an existing vector
- **Removing elements**
  Use `vec <- vec[-(element_index)]` to *remove* the element at element_index from `vec`
- **Single-element alterations**
  Use `vec[index] <- newvalue` to alter a single element
  Beware of type mismatches here!

```
>temp_vec <- 1:5;
>temp_vec
[1] 1 2 3 4 5
>temp_vec <- c(temp_vec, 6);
>temp_vec;
[1] 1 2 3 4 5 6
>temp_vec2 <- c(temp_vec, 'Cat');
>temp_vec2;
[1] "1"   "2"   "3"   "4"   "5"   "6"
[7] "Cat"
>temp_vec[-2];
[1] 1 3 4 5 6
>temp_vec[3] <- 100;
>temp_vec;
[1]   1   2 100   4   5   6
>temp_vec1 <- 1:5; temp_vec2 <- 7:10;
>c(temp_vec1, 6, temp_vec2);
 [1]  1  2  3  4  5  6  7  8  9 10
```

Manipulating Data - Vectors
Indexing

Numerical / nominative

- Ranges
- By names
- Finding indexes

```
>my_name[1:3];
first_1 first_2 first_3
    "D"     "a"     "v"
>my_name[(1:3) * 2];
first_2 first_4   space
    "a"     "i"     " "
>my_name[c(1, 5, length(my_name))];
first_1 first_5  last_2
    "D"     "d"     "."
>my_name['first_3'];
first_3
    "v"
>my_name[c('first_1', 'last_1')]
first_1  last_1
    "D"     "R"
```

Logical

- Review

```
>temp_vec <- 1:10;
>temp_vec <= 5;
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
 [7] FALSE FALSE FALSE FALSE
>temp_vec[temp_vec <= 5];
[1] 1 2 3 4 5
>my_name == 'D'
first_1 first_2 first_3 first_4 first_5
   TRUE   FALSE   FALSE   FALSE   FALSE
  space  last_1  last_2
  FALSE   FALSE   FALSE
>my_name[my_name == 'D'];
first_1
    "D"
>my_name[my_name == ' ' | my_name == '.'];
 space last_2
   " "     "."
>my_name[!(my_name %in% c(' ', '.'))];
first_1 first_2 first_3 first_4 first_5
    "D"     "a"     "v"     "i"     "d"
 last_1
    "R"
```

Outline

Complex types
Overview

Lists

Lists are like vectors, except that they can hold *different* types in different slots.

```
>list_ex <- list(my_name, 1:5, function(x) { return
      (1/x) });
>list_ex;
[[1]]
first_1 first_2 first_3 first_4 first_5
    "D"     "a"     "v"     "i"     "d"
  space  last_1  last_2
    " "     "R"     "."

[[2]]
[1] 1 2 3 4 5

[[3]]
function(x) { return(1/x) }
>list_ex2 <- list(name_ta=my_name, one_to_five=1:5,
       myfun = function(x) { return(1/x) });
>list_ex2;
$name_ta
first_1 first_2 first_3 first_4 first_5
    "D"     "a"     "v"     "i"     "d"
  space  last_1  last_2
    " "     "R"     "."

$one_to_five
[1] 1 2 3 4 5

$myfun
function(x) { return(1/x) }
>class(list_ex);
[1] "list"
>class(list_ex2[[1]]);
[1] "character"
```

Complex types
Overview

Data frames

Data frames are kind of like matrices and kind of like lists. Different members can be of different types (like lists). Each column (member) of a data frame, however, must be of equal length (like columns in a matrix).[a]

[a] In fact, data frames *are* lists.

```
>cost_per=c(0.10, 0.25, 0.50);
>items=c('spam', 'egg', 'foobar');
>on_hand=c(123, 153, 55);
>df <- data.frame(items=items, on_hand=on_hand,
       cost_per=cost_per);
>df
   items on_hand cost_per
1   spam     123     0.10
2    egg     153     0.25
3 foobar      55     0.50
>total_value <- df$on_hand * df$cost_per
```

```
>df <- cbind(df, total_value=total_value)
>df
   items on_hand cost_per total_value
1   spam     123     0.10       12.30
2    egg     153     0.25       38.25
3 foobar      55     0.50       27.50
>df2 <- cbind(df, weight_per=c(0.5, 0.1, 3))
>df2
   items on_hand cost_per total_value
1   spam     123     0.10       12.30
2    egg     153     0.25       38.25
3 foobar      55     0.50       27.50
  weight_per
1        0.5
2        0.1
3        3.0
```

Complex types
Overview

```
>df2 <- cbind(
        df2,
        total_weight=df2$on_hand * df2$weight_per,
        value_density=df2$total_value/(df2$on_hand * df2$weight_per)
              )
>df2
   items on_hand cost_per total_value
1   spam     123     0.10       12.30
2    egg     153     0.25       38.25
3 foobar      55     0.50       27.50
  weight_per total_weight value_density
1        0.5         61.5     0.2000000
2        0.1         15.3     2.5000000
3        3.0        165.0     0.1666667
>class(df2);
[1] "data.frame"
>is.list(df2);
[1] TRUE
```

Complex types
Slicing and extracting

Slicing

Using brackets ([`indices`]-notation) to create "subsets" of a vector, list, or data frame is called *slicing*.

Slice type preservation

Slicing an object always returns an object of the same type.

- A slice of a character vector is a character vector; a slice of a numeric vector is a numeric vector, etc.
- A slice of a list / data frame is always a list / data frame (even if it has only 1 member).
- A SLICE OF A LIST / DATA FRAME IS ALWAYS A LIST / DATA FRAME (EVEN IF IT HAS ONLY 1 MEMBER).

Complex types
Slicing and extracting

Extracting

Using double-brackets ([[`index`]]) or dollar-sign notation (`mylist$member_name`) to examine single members of a list or data frame is called extraction.

Extract types

The result of an extraction operation is of type defined by the returned member.

- Extraction *only* works with lists / data frames.
- Extraction can only return a single member.

Complex types
Syntax

Slice / extraction

1. Slicing can only be performed using single brackets.
2. Extraction can only be performed using double-brackets or dollar-sign - name notation.

Other complex operations

- `cbind()` joins matrices / data frames columnwise
- `rbind()` joins matrices / data frames columnwise
- `names()` is used for getting and setting element names
- `attr()` is used for getting and setting other object attributes.

```
>df <- data.frame(items=items, on_hand=on_hand,
       cost_per=cost_per);
>list1 <- as.list(df);
>list1;
$items
[1] spam   egg    foobar
Levels: egg foobar spam

$on_hand
[1] 123 153  55

$cost_per
[1] 0.10 0.25 0.50
>names(df);
[1] "items"    "on_hand"  "cost_per"
>names(df)[1] <- c('My items');
>df
  My items on_hand cost_per
1     spam      123     0.10
2      egg      153     0.25
3   foobar       55     0.50
```

Complex types
Syntax

Slice / extraction

1. Slicing can only be performed using single brackets.
2. Extraction can only be performed using double-brackets or dollar-sign - name notation.

Other complex operations

- `cbind()` joins matrices / data frames columnwise
- `rbind()` joins matrices / data frames columnwise
- `names()` is used for getting and setting element names
- `attr()` is used for getting and setting other object attributes.

```
>df <- data.frame(items=items, on_hand=on_hand,
       cost_per=cost_per);
>list1 <- as.list(df);
>list1;
$items
[1] spam   egg    foobar
Levels: egg foobar spam

$on_hand
[1] 123 153  55

$cost_per
[1] 0.10 0.25 0.50
>names(df);
[1] "items"    "on_hand"  "cost_per"
>names(df)[1] <- c('My items');
>df
  My items on_hand cost_per
1     spam      123     0.10
2      egg      153     0.25
3   foobar       55     0.50
```

Complex types
Syntax

Slice / extraction

1. Slicing can only be performed using single brackets.
2. Extraction can only be performed using double-brackets or dollar-sign - name notation.

Other complex operations

- `cbind()` joins matrices / data frames columnwise
- `rbind()` joins matrices / data frames columnwise
- `names()` is used for getting and setting element names
- `attr()` is used for getting and setting other object attributes.

```
>df <- data.frame(items=items, on_hand=on_hand,
       cost_per=cost_per);
>list1 <- as.list(df);
>list1;
$items
[1] spam   egg    foobar
Levels: egg foobar spam

$on_hand
[1] 123 153  55

$cost_per
[1] 0.10 0.25 0.50
>names(df);
[1] "items"    "on_hand"  "cost_per"
>names(df)[1] <- c('My items');
>df
  My items on_hand cost_per
1     spam      123     0.10
2      egg      153     0.25
3   foobar       55     0.50
```

Complex types
Syntax

Slice / extraction

1. Slicing can only be performed using single brackets.
2. Extraction can only be performed using double-brackets or dollar-sign - name notation.

Other complex operations

- `cbind()` joins matrices / data frames columnwise
- `rbind()` joins matrices / data frames columnwise
- `names()` is used for getting and setting element names
- `attr()` is used for getting and setting other object attributes.

```
>df <- data.frame(items=items, on_hand=on_hand,
       cost_per=cost_per);
>list1 <- as.list(df);
>list1;
$items
[1] spam   egg    foobar
Levels: egg foobar spam

$on_hand
[1] 123 153  55

$cost_per
[1] 0.10 0.25 0.50
>names(df);
[1] "items"    "on_hand"  "cost_per"
>names(df)[1] <- c('My items');
>df
  My items on_hand cost_per
1     spam      123     0.10
2      egg      153     0.25
3   foobar       55     0.50
```

Complex types
Syntax

Slice / extraction

1. Slicing can only be performed using single brackets.
2. Extraction can only be performed using double-brackets or dollar-sign - name notation.

Other complex operations

- `cbind()` joins matrices / data frames columnwise
- `rbind()` joins matrices / data frames columnwise
- `names()` is used for getting and setting element names
- `attr()` is used for getting and setting other object attributes.

```
>df <- data.frame(items=items, on_hand=on_hand,
       cost_per=cost_per);
>list1 <- as.list(df);
>list1;
$items
[1] spam   egg    foobar
Levels: egg foobar spam

$on_hand
[1] 123 153  55

$cost_per
[1] 0.10 0.25 0.50
>names(df);
[1] "items"    "on_hand"  "cost_per"
>names(df)[1] <- c('My items');
>df
  My items on_hand cost_per
1     spam      123     0.10
2      egg      153     0.25
3   foobar       55     0.50
```

Complex types
Syntax

Slice / extraction

1. Slicing can only be performed using single brackets.
2. Extraction can only be performed using double-brackets or dollar-sign - name notation.

Other complex operations

- `cbind()` joins matrices / data frames columnwise
- `rbind()` joins matrices / data frames columnwise
- `names()` is used for getting and setting element names
- `attr()` is used for getting and setting other object attributes.

```
>df <- data.frame(items=items, on_hand=on_hand,
       cost_per=cost_per);
>list1 <- as.list(df);
>list1;
$items
[1] spam   egg    foobar
Levels: egg foobar spam

$on_hand
[1] 123 153  55

$cost_per
[1] 0.10 0.25 0.50
>names(df);
[1] "items"    "on_hand"  "cost_per"
>names(df)[1] <- c('My items');
>df
  My items on_hand cost_per
1     spam      123     0.10
2      egg      153     0.25
3   foobar       55     0.50
```

Outline

Functions
Unexplained functions

Reading other people's functions

If you *need* to know how the functions I gave you work, here's an overview of what I've skipped over and why.

1. **random data** The functions `runif()` and `rnorm()` are used to generate arbitrary random numeric vectors.
2. **string manipulation** Manipulation of strings uses *another* language called *regular expressions* or *regex* that is embedded in *R* (and nearly all other languages). Everything i've given you that manipulates text (e.g. `parsePolynomial()`) uses a number of *regex* functions including `gsub()`, `strsplit()` and `paste()`. String processing is a great thing to learn, but not the focus of this course.

Functions
Unexplained functions

Reading other people's functions

If you *need* to know how the functions I gave you work, here's an overview of what I've skipped over and why.

1. **other** Most of the remaining functions that you have seen but not learned about are either
   - Peculiarities of the *R* language and not useful in terms of general numerical computing. (e.g. `options(expressions=500000);`)
   - Operating system interaction. (e.g. `system.time(qSort(inVec));`)
   - Artifacts of my formatting. (e.g. the first "block" in the source I posted for appendix 1.)
   - Abstract / challenging / confusing in nature or implementation. (The hacks used in the `maxima_utilites.R` file)

Functions
Reading the documentation

Documentation

Consult the online documentation frequently. This includes the examples.

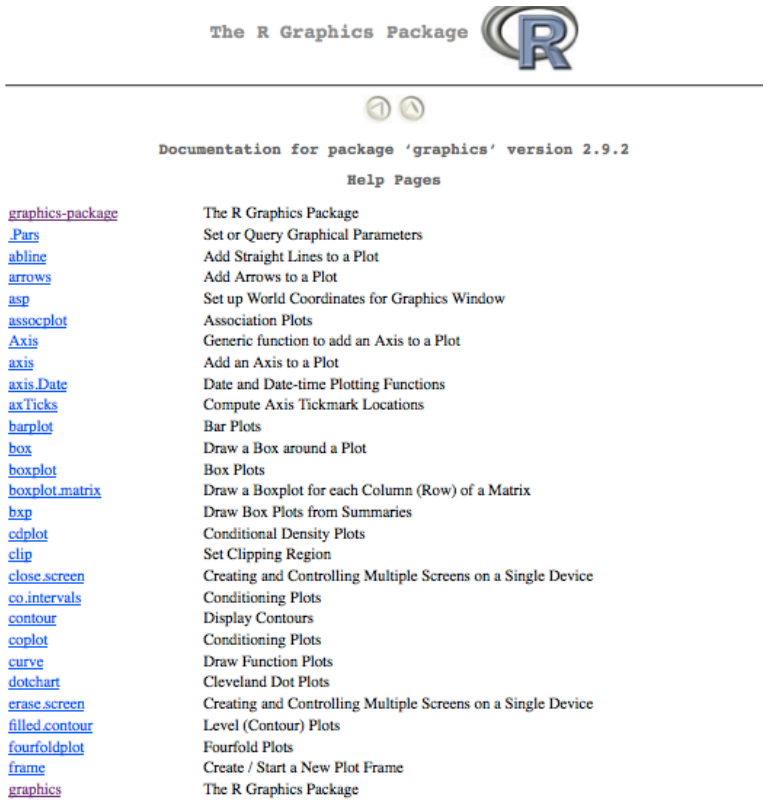
Invariably, you will encounter both functions that you have forgotten how to use and "problems" for which you would *expect* a function to have already addressed.

> *How do you expect them to graph the functions for the cobweb plots? Is there an R function for plotting a given algebraic expression? This also applies for plotting the exponential functions they generate in the glucose model question.*
>
> *Look at the code on the top left of page 3. I think this should answer that question. There is another function, curve() that I haven't talked about that can be used as well. Try the following:*
>
> ```
> curve(1-exp(-x), from=0.01, to=10)
> ```

.

Functions
Reading the documentation

Given the assumption that generation a plot from function might be in the "graphics" package . . .



The R Graphics Package

Documentation for package 'graphics' version 2.9.2

Help Pages

graphics-package	The R Graphics Package
.Pars	Set or Query Graphical Parameters
abline	Add Straight Lines to a Plot
arrows	Add Arrows to a Plot
asp	Set up World Coordinates for Graphics Window
assocplot	Association Plots
Axis	Generic function to add an Axis to a Plot
axis	Add an Axis to a Plot
axis.Date	Date and Date-time Plotting Functions
axTicks	Compute Axis Tickmark Locations
barplot	Bar Plots
box	Draw a Box around a Plot
boxplot	Box Plots
boxplot.matrix	Draw a Boxplot for each Column (Row) of a Matrix
bxp	Draw Box Plots from Summaries
cdplot	Conditional Density Plots
clip	Set Clipping Region
close.screen	Creating and Controlling Multiple Screens on a Single Device
co.intervals	Conditioning Plots
contour	Display Contours
coplot	Conditioning Plots
curve	Draw Function Plots
dotchart	Cleveland Dot Plots
erase.screen	Creating and Controlling Multiple Screens on a Single Device
filled.contour	Level (Contour) Plots
fourfoldplot	Fourfold Plots
frame	Create / Start a New Plot Frame
graphics	The R Graphics Package

Functions
Reading the documentation

Commands

- `help()` (`?`) - Function documentation
- `help(package=`*`package name`*`)`
- `help.search()` (`??`)
- `help.start()`
- `example()`
- `function name` (without parenthesis)

```
>help(ls);
>?ls
>?`+`  # use backquotes when R doesn't
```

Functions
Reading the documentation

Commands

- `help()` (`?`)
- `help(package=`*`package name`*`)` - Package documentation listing
- `help.search()` (`??`)
- `help.start()`
- `example()`
- `function name` (without parenthesis)

```
># help(package=RInterval);
># truncated for space
>cat(paste(help(package=RInterval)$info[[1]], collapse="\n"),
      '\n');
Package:        RInterval
Type:           Package
Title:          Package for efficiently dealing with
                genomic intervals
Version:        0.14
Date:           2009-09-09
Author:         David M. Rosenberg
Maintainer:     Who to complain to
                <rosenbergdm@uchicago.edu>
Description:    Classes and methods for efficiently
                analyzing single dimension genomic
                intervals, with an emphasis on copy
                number variation.
License:        LGPL
LazyLoad:       yes
Packaged:       2009-09-10 17:19:11 UTC; root
Built:          R 2.9.2; ; 2009-10-15 17:33:27 UTC;
                unix
```

Functions
Reading the documentation

Commands

- `help()` (`?`)
- `help(package=`*`package name`*`)`
- `help.search()` (`??`) - Search *all* documentation for occurances of the given term.
- `help.start()`
- `example()`
- `function name` (without parenthesis)

```
>help.search('digest');
```

Functions
Reading the documentation

Commands

- `help()` (`?`)
- `help(package=`*`package name`*`)`
- `help.search()` (`??`)
- `help.start()` - Open *HTML* help browser in the default web browser.
- `example()`
- `function name` (without parenthesis)

Functions
Reading the documentation

Commands

- `help()` (`?`)
- `help(package=`*`package name`*`)`
- `help.search()` (`??`)
- `help.start()`
- `example()` - Execute any example code in a function's documentation. These examples should *never* fail.
- `function name` (without parenthesis)

```
>example(is.integer);
is.ntg>  ## as.integer() truncates:
is.ntg>  x <- pi * c(-1:1,10)

is.ntg>  as.integer(x)
[1] -3  0  3 31
```

Functions
Reading the documentation

Commands

- `help()` (`?`)
- `help(package=`*`package name`*`)`
- `help.search()` (`??`)
- `help.start()`
- `example()`
- `function name` (without parenthesis) - Show function body.

```
>isTRUE;
function (x)
identical(TRUE, x)
<environment: namespace:base>
>lsf.str;
function (pos = -1, envir, ...)
{
    if (missing(envir))
        envir <- as.environment(pos)
    ls.str(pos = pos, envir = envir, mode = "function", ...)
}
<environment: namespace:utils>
```

Outline

Style
Submissions

Example and template

An example and a template for how I'd like you to *try* to submit your homework to me will be available by tonight.

The "submission" style guidelines are primarily to speed my grading of homework.

- Each assignment should be submitted as a *single* `.R` file. Additional (non-`.R`) files / hard copies may be necessary, including
  - Plots
  - Analytical problems
  - Data tables
- Anything in the assignment which is not *R* code should be commented out (preceded by pound signs).

Style
Submissions

File header

The following header should start your `.R` file (the first three lines may be omitted). Substitute the relevant information for each all-caps slot.

```
#!/usr/bin/env r
# encoding: utf-8
#
# name: STUDENT_NAME
# assignment: ASSIGNMENT_NUMBER
# date: DATE_OF_SUBMISSION
# filename: NAME_OF_FILE
```

Question header

The following block should be inserted before each answer.

```
#/-############################################################################
# name: STUDENT_NAME
# assignment: ASSIGNMENT_NUMBER
# date: DATE_OF_SUBMISSION
# question: QUESTION_NUMBER
# subquestion: SUBQUESTION_LETTER_IF_PRESENT_OTHERWISE_EMPTY
# other files: NAMES_OF_OTHER_RELATED_FILES
##############################################################################/-#
```

Style
Submissions

File header

The following header should start your `.R` file (the first three lines may be omitted). Substitute the relevant information for each all-caps slot.

Question header

The following block should be inserted before each answer.

```
#/-#####################################################################
# name: STUDENT_NAME
# assignment: ASSIGNMENT_NUMBER
# date: DATE_OF_SUBMISSION
# question: QUESTION_NUMBER
# subquestion: SUBQUESTION_LETTER_IF_PRESENT_OTHERWISE_EMPTY
# other files: NAMES_OF_OTHER_RELATED_FILES
#####################################################################/-
```

Style
Rules

Category	Description
variable names	Variable names should describe their function and follow a consistent capitalization scheme
indentation	*code blocks* (generally delimited by brackets) should be indented (recommended 2 spaces) relative to surrounding code
blank lines	use blank lines to separate independent "chunks" or concepts
spaces	blank spaces should surround symbols and parentheses
comments	comments should be used liberally to explain code and concepts
line length	Individual lines should be no more than 80 characters long. Continuation lines (if necessary) should be indented relative to surrounding code

Style
The golden rule

The Golden Rule

Whatever your personal stylistic preferences / conventions are, **be consistent**.