Introduction to R

Xing Qiu

Department of Biostatistics and Computational Biology, University of Rochester

Summer School on Computational Immunology
6/1/2015, Rochester, New York

Outline

Introduction

- R is an open source implementation of the S language, you may call it a free "clone" of commercial statistical computing system S-plus.
- R is a high level scripting language (like MatLab and Python but unlike C and FORTRAN) which is easy to use. It is best suited for small (thousands of data points) to medium (under a million data points) sized data analysis.
- R is the *de facto* programming language among statisticians for developing statistical software. In recent years it has been gaining user base in other fields such as bioinformatics (via the BioConductor project), mathematical finance, medical imaging, and social sciences.

Introduction

- R is an open source implementation of the S language, you may call it a free “clone” of commercial statistical computing system S-plus.
- R is a high level scripting language (like MatLab and Python but unlike C and FORTRAN) which is easy to use. It is best suited for small (thousands of data points) to medium (under a million data points) sized data analysis.
- R is the *de facto* programming language among statisticians for developing statistical software. In recent years it has been gaining user base in other fields such as bioinformatics (via the BioConductor project), mathematical finance, medical imaging, and social sciences.

Introduction

- R is an open source implementation of the S language, you may call it a free “clone” of commercial statistical computing system S-plus.
- R is a high level scripting language (like MatLab and Python but unlike C and FORTRAN) which is easy to use. It is best suited for small (thousands of data points) to medium (under a million data points) sized data analysis.
- R is the *de facto* programming language among statisticians for developing statistical software. In recent years it has been gaining user base in other fields such as bioinformatics (via the BioConductor project), mathematical finance, medical imaging, and social sciences.

R and R libraries

- R installation. Windows, Mac, Linux. My lectures will be based on R 3.2 for Windows. If you are not sure about using the 64-bit or 32-bit version, use the 32-bit one.
- R is a functional language. Most commonly used functions are provided by the R core, such as `lm()`.
- Many useful functions are provided as external R libraries that you can install by `install.packages("package-name")`.
- The idea of a *repository* (Android/iPhone Apps).

R and R libraries

- R installation. Windows, Mac, Linux. My lectures will be based on R 3.2 for Windows. If you are not sure about using the 64-bit or 32-bit version, use the 32-bit one.
- R is a functional language. Most commonly used functions are provided by the R core, such as `lm()`.
- Many useful functions are provided as external R libraries that you can install by `install.packages("package-name")`.
- The idea of a *repository* (Android/iPhone Apps).

R and R libraries

- R installation. Windows, Mac, Linux. My lectures will be based on R 3.2 for Windows. If you are not sure about using the 64-bit or 32-bit version, use the 32-bit one.
- R is a functional language. Most commonly used functions are provided by the R core, such as `lm()`.
- Many useful functions are provided as external R libraries that you can install by `install.packages("package-name")`.
- The idea of a *repository* (Android/iPhone Apps).

R and R libraries

- R installation. Windows, Mac, Linux. My lectures will be based on R 3.2 for Windows. If you are not sure about using the 64-bit or 32-bit version, use the 32-bit one.
- R is a functional language. Most commonly used functions are provided by the R core, such as `lm()`.
- Many useful functions are provided as external R libraries that you can install by `install.packages("package-name")`.
- The idea of a *repository* (Android/iPhone Apps).

R and files

- R can read/write from/to text-based files directly.
- Example: `read.table()`; `read.csv()`.
- Packages `xlsx` and `foreign` provide functionalities to read files saved by Excel, Matlab, SAS, etc.
- Caution: To be on the safe side, always use Excel/Matlab/SAS etc to produce text-based data sheets (such as `.csv` files) first.
- Search: "R data import/export" for a free manual on general strategy of read/write data.
- R is a *scripting* language. R scripts are simple text files.
- In principle, every text editor can be used to write R programs. (RStudio, TextMate Tinn-R, Emacs/ESS)
- Another type of R file: image files. Will discuss them later.

R and files

- R can read/write from/to text-based files directly.
- Example: `read.table()`; `read.csv()`.
- Packages `xlsx` and `foreign` provide functionalities to read files saved by Excel, Matlab, SAS, etc.
- Caution: To be on the safe side, always use Excel/Matlab/SAS etc to produce text-based data sheets (such as `.csv` files) first.
- Search: "R data import/export" for a free manual on general strategy of read/write data.
- R is a *scripting* language. R scripts are simple text files.
- In principle, every text editor can be used to write R programs. (RStudio, TextMate Tinn-R, Emacs/ESS)
- Another type of R file: image files. Will discuss them later.

R and files

- R can read/write from/to text-based files directly.
- Example: `read.table()`; `read.csv()`.
- Packages `xlsx` and `foreign` provide functionalities to read files saved by Excel, Matlab, SAS, etc.
- Caution: To be on the safe side, always use Excel/Matlab/SAS etc to produce text-based data sheets (such as `.csv` files) first.
- Search: "R data import/export" for a free manual on general strategy of read/write data.
- R is a *scripting* language. R scripts are simple text files.
- In principle, every text editor can be used to write R programs. (RStudio, TextMate Tinn-R, Emacs/ESS)
- Another type of R file: image files. Will discuss them later.

R and files

- R can read/write from/to text-based files directly.
- Example: `read.table()`; `read.csv()`.
- Packages `xlsx` and `foreign` provide functionalities to read files saved by Excel, Matlab, SAS, etc.
- Caution: To be on the safe side, always use Excel/Matlab/SAS etc to produce text-based data sheets (such as `.csv` files) first.
- Search: “R data import/export” for a free manual on general strategy of read/write data.
- R is a *scripting* language. R scripts are simple text files.
- In principle, every text editor can be used to write R programs. (RStudio, TextMate Tinn-R, Emacs/ESS)
- Another type of R file: image files. Will discuss them later.

R and files

- R can read/write from/to text-based files directly.
- Example: `read.table()`; `read.csv()`.
- Packages `xlsx` and `foreign` provide functionalities to read files saved by Excel, Matlab, SAS, etc.
- Caution: To be on the safe side, always use Excel/Matlab/SAS etc to produce text-based data sheets (such as `.csv` files) first.
- Search: "R data import/export" for a free manual on general strategy of read/write data.
- R is a *scripting* language. R scripts are simple text files.
- In principle, every text editor can be used to write R programs. (RStudio, TextMate Tinn-R, Emacs/ESS)
- Another type of R file: image files. Will discuss them later.

R and files

- R can read/write from/to text-based files directly.
- Example: `read.table()`; `read.csv()`.
- Packages `xlsx` and `foreign` provide functionalities to read files saved by Excel, Matlab, SAS, etc.
- Caution: To be on the safe side, always use Excel/Matlab/SAS etc to produce text-based data sheets (such as `.csv` files) first.
- Search: “R data import/export” for a free manual on general strategy of read/write data.
- R is a *scripting* language. R scripts are simple text files.
- In principle, every text editor can be used to write R programs. (RStudio, TextMate Tinn-R, Emacs/ESS)
- Another type of R file: image files. Will discuss them later.

R and files

- R can read/write from/to text-based files directly.
- Example: `read.table()`; `read.csv()`.
- Packages `xlsx` and `foreign` provide functionalities to read files saved by Excel, Matlab, SAS, etc.
- Caution: To be on the safe side, always use Excel/Matlab/SAS etc to produce text-based data sheets (such as `.csv` files) first.
- Search: "R data import/export" for a free manual on general strategy of read/write data.
- R is a *scripting* language. R scripts are simple text files.
- In principle, every text editor can be used to write R programs. (RStudio, TextMate Tinn-R, Emacs/ESS)
- Another type of R file: image files. Will discuss them later.

R and files

- R can read/write from/to text-based files directly.
- Example: `read.table()`; `read.csv()`.
- Packages `xlsx` and `foreign` provide functionalities to read files saved by Excel, Matlab, SAS, etc.
- Caution: To be on the safe side, always use Excel/Matlab/SAS etc to produce text-based data sheets (such as `.csv` files) first.
- Search: "R data import/export" for a free manual on general strategy of read/write data.
- R is a *scripting* language. R scripts are simple text files.
- In principle, every text editor can be used to write R programs. (RStudio, TextMate Tinn-R, Emacs/ESS)
- Another type of R file: image files. Will discuss them later.

Running R interactively

- Start R. Try some file related operations. `getwd()`, `setwd()`, `dir()`.
- Remark: `ls()` has a different meaning in R.
- Some simple calculator-like operations. `^`, `log()`.

Running R interactively

- Start R. Try some file related operations. `getwd()`, `setwd()`, `dir()`.
- Remark: `ls()` has a different meaning in R.
- Some simple calculator-like operations. `^`, `log()`.

Running R interactively

- Start R. Try some file related operations. `getwd()`, `setwd()`, `dir()`.
- Remark: `ls()` has a different meaning in R.
- Some simple calculator-like operations. `^`, `log()`.

Objects

- The most basic types of objects: numeric (integer, real), complex, character, logical.
- Combinations of the above building blocks: vector, matrix, array, list, data.frame; string, factor, etc.
- Other common objects: expression, function, formula, ordered.
- Example: assign `x` a value by `<-` and by `assign()`. The latter is more suitable in a function/loop. Note that "=" can also be used but its usage is discouraged because "=" has a different syntactic meaning in function construction.
- Use `ls()` to list all objects, `rm()` to remove a particular object.
- Use `str()` to find out the *types* of an object.

Objects

- The most basic types of objects: numeric (integer, real), complex, character, logical.
- Combinations of the above building blocks: vector, matrix, array, list, data.frame; string, factor, etc.
- Other common objects: expression, function, formula, ordered.
- Example: assign `x` a value by `<-` and by `assign()`. The latter is more suitable in a function/loop. Note that "=" can also be used but its usage is discouraged because "=" has a different syntactic meaning in function construction.
- Use `ls()` to list all objects, `rm()` to remove a particular object.
- Use `str()` to find out the *types* of an object.

Objects

- The most basic types of objects: numeric (integer, real), complex, character, logical.
- Combinations of the above building blocks: vector, matrix, array, list, data.frame; string, factor, etc.
- Other common objects: expression, function, formula, ordered.
- Example: assign `x` a value by `<-` and by `assign()`. The latter is more suitable in a function/loop. Note that "=" can also be used but its usage is discouraged because "=" has a different syntactic meaning in function construction.
- Use `ls()` to list all objects, `rm()` to remove a particular object.
- Use `str()` to find out the *types* of an object.

Objects

- The most basic types of objects: numeric (integer, real), complex, character, logical.
- Combinations of the above building blocks: vector, matrix, array, list, data.frame; string, factor, etc.
- Other common objects: expression, function, formula, ordered.
- Example: assign `x` a value by `<-` and by `assign()`. The latter is more suitable in a function/loop. Note that “=” can also be used but its usage is discouraged because “=” has a different syntactic meaning in function construction.
- Use `ls()` to list all objects, `rm()` to remove a particular object.
- Use `str()` to find out the *types* of an object.

Objects

- The most basic types of objects: numeric (integer, real), complex, character, logical.
- Combinations of the above building blocks: vector, matrix, array, list, data.frame; string, factor, etc.
- Other common objects: expression, function, formula, ordered.
- Example: assign `x` a value by `<-` and by `assign()`. The latter is more suitable in a function/loop. Note that "=" can also be used but its usage is discouraged because "=" has a different syntactic meaning in function construction.
- Use `ls()` to list all objects, `rm()` to remove a particular object.
- Use `str()` to find out the *types* of an object.

Objects

- The most basic types of objects: numeric (integer, real), complex, character, logical.
- Combinations of the above building blocks: vector, matrix, array, list, data.frame; string, factor, etc.
- Other common objects: expression, function, formula, ordered.
- Example: assign `x` a value by `<-` and by `assign()`. The latter is more suitable in a function/loop. Note that "=" can also be used but its usage is discouraged because "=" has a different syntactic meaning in function construction.
- Use `ls()` to list all objects, `rm()` to remove a particular object.
- Use `str()` to find out the *types* of an object.

Converting an object to a different type

- You can convert (cast, coerce) one type of objects into another, provided that this conversion is reasonable.
- Examples: `x <- c(rep(0, 3), rep(1, 4)); str(x); as.complex(x); as.logical(x); as.character(x)`
- Counter example: `as.numeric("foo")`.

Converting an object to a different type

- You can convert (cast, coerce) one type of objects into another, provided that this conversion is reasonable.
- Examples: `x <- c(rep(0, 3), rep(1, 4)); str(x); as.complex(x); as.logical(x); as.character(x)`
- Counter example: `as.numeric("foo").`

Converting an object to a different type

- You can convert (cast, coerce) one type of objects into another, provided that this conversion is reasonable.
- Examples: `x <- c(rep(0, 3), rep(1, 4)); str(x); as.complex(x); as.logical(x); as.character(x)`
- Counter example: `as.numeric("foo")`.

String functions

- `paste(a, b, sep="foo").`
- `strsplit("foo-bar", "-").`
- Show that math functions are invalid for strings.

String functions

- `paste(a, b, sep="foo")`.
- `strsplit("foo-bar", "-")`.
- Show that math functions are invalid for strings.

String functions

- `paste(a, b, sep="foo")`.
- `strsplit("foo-bar", "-")`.
- Show that math functions are invalid for strings.

Getting help

- The RStudio way.
- `help("mean")` or a shorter version `?"mean"`
- Remark: the use of quotes indicates that "mean" is a *string object*. Unquoted `mean` is a *function object*. In this example, R is smart enough to figure out you want to find help on a topic called mean even if you don't quote this keyword, but it is a good habit to quote a string because this subtle difference can be crucial under other circumstances.
- Try `help(mean)`, `help("+")`, and `help(+)`.
- Last but not the least: Being an open source software, all source code are open. Just type the function name will give you the source code!
- A few words about the generic functions such as `summary()` and `summary.lm()`.

Getting help

- The RStudio way.
- `help("mean")` or a shorter version `?"mean"`
- Remark: the use of quotes indicates that “mean” is a *string object*. Unquoted `mean` is a *function object*. In this example, R is smart enough to figure out you want to find help on a topic called mean even if you don't quote this keyword, but it is a good habit to quote a string because this subtle difference can be crucial under other circumstances.
- Try `help(mean)`, `help("+")`, and `help(+)`.
- Last but not the least: Being an open source software, all source code are open. Just type the function name will give you the source code!
- A few words about the generic functions such as `summary()` and `summary.lm()`.

Getting help

- The RStudio way.
- `help("mean")` or a shorter version `?"mean"`
- Remark: the use of quotes indicates that "mean" is a *string object*. Unquoted `mean` is a *function object*. In this example, R is smart enough to figure out you want to find help on a topic called mean even if you don't quote this keyword, but it is a good habit to quote a string because this subtle difference can be crucial under other circumstances.
- Try `help(mean)`, `help("+")`, and `help(+)`.
- Last but not the least: Being an open source software, all source code are open. Just type the function name will give you the source code!
- A few words about the generic functions such as `summary()` and `summary.lm()`.

Getting help

- The RStudio way.
- `help("mean")` or a shorter version `?"mean"`
- Remark: the use of quotes indicates that "mean" is a *string object*. Unquoted `mean` is a *function object*. In this example, R is smart enough to figure out you want to find help on a topic called mean even if you don't quote this keyword, but it is a good habit to quote a string because this subtle difference can be crucial under other circumstances.
- Try `help(mean)`, `help("+")`, and `help(+)`.
- Last but not the least: Being an open source software, all source code are open. Just type the function name will give you the source code!
- A few words about the generic functions such as `summary()` and `summary.lm()`.

Getting help

- The RStudio way.
- `help("mean")` or a shorter version `?"mean"`
- Remark: the use of quotes indicates that "mean" is a *string object*. Unquoted `mean` is a *function object*. In this example, R is smart enough to figure out you want to find help on a topic called mean even if you don't quote this keyword, but it is a good habit to quote a string because this subtle difference can be crucial under other circumstances.
- Try `help(mean)`, `help("+")`, and `help(+)`.
- Last but not the least: Being an open source software, all source code are open. Just type the function name will give you the source code!
- A few words about the generic functions such as `summary()` and `summary.lm()`.

Getting help

- The RStudio way.
- `help("mean")` or a shorter version `?"mean"`
- Remark: the use of quotes indicates that “mean” is a *string object*. Unquoted `mean` is a *function object*. In this example, R is smart enough to figure out you want to find help on a topic called mean even if you don’t quote this keyword, but it is a good habit to quote a string because this subtle difference can be crucial under other circumstances.
- Try `help(mean)`, `help("+")`, and `help(+)`.
- Last but not the least: Being an open source software, all source code are open. Just type the function name will give you the source code!
- A few words about the generic functions such as `summary()` and `summary.lm()`.

R commands, case sensitivity, etc

- R commands are case sensitive, so “Mike” and “mike” are two different commands/objects.
- Object names must start with letters or a ’.’. By UNIX convention, objects start with ’.’ is meant to be *invisible*, so a simple `ls()` won’t list it.
- You can put several mini-commands in one line by separating them with ’;’.
- First and secondary prompt.

R commands, case sensitivity, etc

- R commands are case sensitive, so “Mike” and “mike” are two different commands/objects.
- Object names must start with letters or a ’.’. By UNIX convention, objects start with ’.’ is meant to be *invisible*, so a simple `ls()` won’t list it.
- You can put several mini-commands in one line by separating them with ’;’.
- First and secondary prompt.

R commands, case sensitivity, etc

- R commands are case sensitive, so “Mike” and “mike” are two different commands/objects.
- Object names must start with letters or a ’.’. By UNIX convention, objects start with ’.’ is meant to be *invisible*, so a simple `ls()` won’t list it.
- You can put several mini-commands in one line by separating them with ’;’.
- First and secondary prompt.

R commands, case sensitivity, etc

- R commands are case sensitive, so “Mike” and “mike” are two different commands/objects.
- Object names must start with letters or a ’.’. By UNIX convention, objects start with ’.’ is meant to be *invisible*, so a simple `ls()` won’t list it.
- You can put several mini-commands in one line by separating them with ’;’.
- First and secondary prompt.

Saving objects, etc

- When you quit R (by using command `q()`), R will ask you to save image. If you say yes (one single character 'y' is OK), R will save all the objects (called an *image of the current workspace*) in a file `.RData` under the current working directory (I'll call it `pwd` henceforth).
- Next time when you start R in the same directory, R will pick these objects up (unless you tell it **not** to do so manually).
- The command history is saved in a file `.Rhistory` under `pwd`.

Saving objects, etc

- When you quit R (by using command `q()`), R will ask you to save image. If you say yes (one single character 'y' is OK), R will save all the objects (called an *image of the current workspace*) in a file `.RData` under the current working directory (I'll call it `pwd` henceforth).
- Next time when you start R in the same directory, R will pick these objects up (unless you tell it **not** to do so manually).
- The command history is saved in a file `.Rhistory` under `pwd`.

Saving objects, etc

- When you quit R (by using command `q()`), R will ask you to save image. If you say yes (one single character 'y' is OK), R will save all the objects (called an *image of the current workspace*) in a file `.RData` under the current working directory (I'll call it `pwd` henceforth).
- Next time when you start R in the same directory, R will pick these objects up (unless you tell it **not** to do so manually).
- The command history is saved in a file `.Rhistory` under `pwd`.

Manually save/load a workspace

- In real life, you almost always want to analyze a set of data in different ways. So you need to save the workspace in different files and manually load them later.
- `save.image("foo.RData")`, `save(obj1, obj2, file="foo.RData")`. The latter gives you finer control over which object(s) to save.
- When quitting R and R asks you to save workspace, answer "no".
- `load("foo.RData")` reloads the saved workspace.

Manually save/load a workspace

- In real life, you almost always want to analyze a set of data in different ways. So you need to save the workspace in different files and manually load them later.
- `save.image("foo.RData")`, `save(obj1, obj2, file="foo.RData")`. The latter gives you finer control over which object(s) to save.
- When quitting R and R asks you to save workspace, answer "no".
- `load("foo.RData")` reloads the saved workspace.

Manually save/load a workspace

- In real life, you almost always want to analyze a set of data in different ways. So you need to save the workspace in different files and manually load them later.
- `save.image("foo.RData")`, `save(obj1, obj2, file="foo.RData")`. The latter gives you finer control over which object(s) to save.
- When quitting R and R asks you to save workspace, answer "no".
- `load("foo.RData")` reloads the saved workspace.

Manually save/load a workspace

- In real life, you almost always want to analyze a set of data in different ways. So you need to save the workspace in different files and manually load them later.
- `save.image("foo.RData")`, `save(obj1, obj2, file="foo.RData")`. The latter gives you finer control over which object(s) to save.
- When quitting R and R asks you to save workspace, answer "no".
- `load("foo.RData")` reloads the saved workspace.

A longer interactive session

- Assign a vector by function `c()`.
- Join two vectors by `c()`.
- vector arithmetic (`(1:3)^2`, `x + 2*y`, etc). It's usually much faster than a for loop (will be introduced later).
- Vector only arithmetic. `sum()`, `prod`, `max()`, `min()` `sort()`, `order()`, `rank()`.
- Advanced: `cumsum()`, `cumprod()`, `cummax()`, `cummin()`; `pmax(x,y)`, `pmin(x,y)`.

A longer interactive session

- Assign a vector by function `c()`.
- Join two vectors by `c()`.
- vector arithmetic (`(1:3)^2`, `x + 2*y`, etc). It's usually much faster than a for loop (will be introduced later).
- Vector only arithmetic. `sum()`, `prod`, `max()`, `min()` `sort()`, `order()`, `rank()`.
- Advanced: `cumsum()`, `cumprod()`, `cummax()`, `cummin()`; `pmax(x,y)`, `pmin(x,y)`.

A longer interactive session

- Assign a vector by function `c()`.
- Join two vectors by `c()`.
- vector arithmetic (`(1:3)^2`, `x + 2*y`, etc). It's usually much faster than a for loop (will be introduced later).
- Vector only arithmetic. `sum()`, `prod`, `max()`, `min()` `sort()`, `order()`, `rank()`.
- Advanced: `cumsum()`, `cumprod()`, `cummax()`, `cummin()`; `pmax(x,y)`, `pmin(x,y)`.

A longer interactive session

- Assign a vector by function `c()`.
- Join two vectors by `c()`.
- vector arithmetic (`(1:3)^2`, `x + 2*y`, etc). It's usually much faster than a for loop (will be introduced later).
- Vector only arithmetic. `sum()`, `prod`, `max()`, `min()` `sort()`, `order()`, `rank()`.
- Advanced: `cumsum()`, `cumprod()`, `cummax()`, `cummin()`; `pmax(x,y)`, `pmin(x,y)`.

A longer interactive session

- Assign a vector by function `c()`.
- Join two vectors by `c()`.
- vector arithmetic (`(1:3)^2`, `x + 2*y`, etc). It's usually much faster than a for loop (will be introduced later).
- Vector only arithmetic. `sum()`, `prod`, `max()`, `min()` `sort()`, `order()`, `rank()`.
- Advanced: `cumsum()`, `cumprod()`, `cummax()`, `cummin()`; `pmax(x,y)`, `pmin(x,y)`.

Generating vectors

- `-2:3`, `seq(0, 10, 2)`, `seq(10, 0, 2)` (wrong), `seq(10, 0, -2)`.
- `rep(TRUE, 3)`, `rep(c(TRUE, FALSE), 3)`, `rep(c(TRUE, FALSE), each=3)`.
- `x <- rnorm(5)`. I will get back to random number generation later.
- Generating a logical vector. `y <- x <= 0`.
- Logic vector function `all()`, `any()`.
- Generate a long vector from two (or more) short vectors `c(1:3, 8:12)`.

Generating vectors

- `-2:3`, `seq(0, 10, 2)`, `seq(10, 0, 2)` (wrong), `seq(10, 0, -2)`.
- `rep(TRUE, 3)`, `rep(c(TRUE, FALSE), 3)`, `rep(c(TRUE, FALSE), each=3)`.
- `x <- rnorm(5)`. I will get back to random number generation later.
- Generating a logical vector. `y <- x <= 0`.
- Logic vector function `all()`, `any()`.
- Generate a long vector from two (or more) short vectors `c(1:3, 8:12)`.

Generating vectors

- `-2:3`, `seq(0, 10, 2)`, `seq(10, 0, 2)` (wrong), `seq(10, 0, -2)`.
- `rep(TRUE, 3)`, `rep(c(TRUE, FALSE), 3)`, `rep(c(TRUE, FALSE), each=3)`.
- `x <- rnorm(5)`. I will get back to random number generation later.
- Generating a logical vector. `y <- x <= 0`.
- Logic vector function `all()`, `any()`.
- Generate a long vector from two (or more) short vectors `c(1:3, 8:12)`.

Generating vectors

- `-2:3`, `seq(0, 10, 2)`, `seq(10, 0, 2)` (wrong), `seq(10, 0, -2)`.
- `rep(TRUE, 3)`, `rep(c(TRUE, FALSE), 3)`, `rep(c(TRUE, FALSE), each=3)`.
- `x <- rnorm(5)`. I will get back to random number generation later.
- Generating a logical vector. `y <- x <= 0`.
- Logic vector function `all()`, `any()`.
- Generate a long vector from two (or more) short vectors `c(1:3, 8:12)`.

Generating vectors

- `-2:3`, `seq(0, 10, 2)`, `seq(10, 0, 2)` (wrong), `seq(10, 0, -2)`.
- `rep(TRUE, 3)`, `rep(c(TRUE, FALSE), 3)`, `rep(c(TRUE, FALSE), each=3)`.
- `x <- rnorm(5)`. I will get back to random number generation later.
- Generating a logical vector. `y <- x <= 0`.
- Logic vector function `all()`, `any()`.
- Generate a long vector from two (or more) short vectors `c(1:3, 8:12)`.

Generating vectors

- `-2:3`, `seq(0, 10, 2)`, `seq(10, 0, 2)` (wrong), `seq(10, 0, -2)`.
- `rep(TRUE, 3)`, `rep(c(TRUE, FALSE), 3)`, `rep(c(TRUE, FALSE), each=3)`.
- `x <- rnorm(5)`. I will get back to random number generation later.
- Generating a logical vector. `y <- x <= 0`.
- Logic vector function `all()`, `any()`.
- Generate a long vector from two (or more) short vectors `c(1:3, 8:12)`.

Slicing (subsetting) a vector

- `x <- rnorm(r), length(x).`
- `x[2], x[2:4], x[c(3,1,2)].`
- Negative indices removes elements `x[c(-2, -4)].`
- Logical vector. `y <- c(TRUE, TRUE, FALSE, FALSE, TRUE); x[y], x[x<0].`
- Change a slice of a vector: `y[y < 0] <- -y[y < 0]` is equivalent to `y <- abs(y).`

Slicing (subsetting) a vector

- `x <- rnorm(r), length(x).`
- `x[2], x[2:4], x[c(3,1,2)].`
- Negative indices removes elements `x[c(-2, -4)].`
- Logical vector. `y <- c(TRUE, TRUE, FALSE, FALSE, TRUE); x[y], x[x<0].`
- Change a slice of a vector: `y[y < 0] <- -y[y < 0]` is equivalent to `y <- abs(y).`

Slicing (subsetting) a vector

- `x <- rnorm(r),length(x).`
- `x[2],x[2:4],x[c(3,1,2)].`
- Negative indices removes elements `x[c(-2, -4)].`
- Logical vector. `y <- c(TRUE, TRUE, FALSE, FALSE, TRUE); x[y],x[x<0].`
- Change a slice of a vector: `y[y < 0] <- -y[y < 0]` is equivalent to `y <- abs(y).`

Slicing (subsetting) a vector

- `x <- rnorm(r), length(x)`.
- `x[2], x[2:4], x[c(3,1,2)]`.
- Negative indices removes elements `x[c(-2, -4)]`.
- Logical vector. `y <- c(TRUE, TRUE, FALSE, FALSE, TRUE); x[y], x[x<0]`.
- Change a slice of a vector: `y[y < 0] <- -y[y < 0]` is equivalent to `y <- abs(y)`.

Slicing (subsetting) a vector

- `x <- rnorm(r), length(x).`
- `x[2], x[2:4], x[c(3,1,2)].`
- Negative indices removes elements `x[c(-2, -4)].`
- Logical vector. `y <- c(TRUE, TRUE, FALSE, FALSE, TRUE); x[y], x[x<0].`
- Change a slice of a vector: `y[y < 0] <- -y[y < 0]` is equivalent to `y <- abs(y).`

Named vectors

- You can assign names to each element of a vector.
- `fruit <- c(5, 10, 1, 20); names(fruit) <- c("orange", "banana", "apple", "peach")`
- Check the names of `fruit`: `names(fruit)`.
- Now you can slice the `fruit` vector by names: `lunch <- fruit[c("apple","orange")]`
- A named vector is sometimes called a hash table or a dictionary in other computer languages.

Named vectors

- You can assign names to each element of a vector.
- `fruit <- c(5, 10, 1, 20); names(fruit) <- c("orange", "banana", "apple", "peach")`
- Check the names of `fruit`: `names(fruit)`.
- Now you can slice the `fruit` vector by names: `lunch <- fruit[c("apple","orange")]`
- A named vector is sometimes called a hash table or a dictionary in other computer languages.

Named vectors

- You can assign names to each element of a vector.
- `fruit <- c(5, 10, 1, 20); names(fruit) <- c("orange", "banana", "apple", "peach")`
- Check the names of `fruit`: `names(fruit)`.
- Now you can slice the `fruit` vector by names: `lunch <- fruit[c("apple","orange")]`
- A named vector is sometimes called a hash table or a dictionary in other computer languages.

Named vectors

- You can assign names to each element of a vector.
- `fruit <- c(5, 10, 1, 20); names(fruit) <- c("orange", "banana", "apple", "peach")`
- Check the names of `fruit`: `names(fruit)`.
- Now you can slice the `fruit` vector by names: `lunch <- fruit[c("apple","orange")]`
- A named vector is sometimes called a hash table or a dictionary in other computer languages.

Named vectors

- You can assign names to each element of a vector.
- `fruit <- c(5, 10, 1, 20); names(fruit) <- c("orange", "banana", "apple", "peach")`
- Check the names of `fruit`: `names(fruit)`.
- Now you can slice the `fruit` vector by names: `lunch <- fruit[c("apple","orange")]`
- A named vector is sometimes called a hash table or a dictionary in other computer languages.

A fairly complex named vector

- Use Welch *t*-test as an example.
- `x <- rnorm(5); y <- rnorm(5); w <- t.test(x, y); w`
- Use `names(w)` or `attributes(w)` to see the structure of `w`.
- Use `w[["p.value"]]` or `w$p.value` to extract a specific element.

A fairly complex named vector

- Use Welch *t*-test as an example.
- `x <- rnorm(5); y <- rnorm(5); w <- t.test(x, y); w`
- Use `names(w)` or `attributes(w)` to see the structure of `w`.
- Use `w[["p.value"]]` or `w$p.value` to extract a specific element.

A fairly complex named vector

- Use Welch *t*-test as an example.
- `x <- rnorm(5); y <- rnorm(5); w <- t.test(x, y); w`
- Use `names(w)` or `attributes(w)` to see the structure of `w`.
- Use `w[["p.value"]]` or `w$p.value` to extract a specific element.

A fairly complex named vector

- Use Welch *t*-test as an example.
- `x <- rnorm(5); y <- rnorm(5); w <- t.test(x, y); w`
- Use `names(w)` or `attributes(w)` to see the structure of `w`.
- Use `w[["p.value"]]` or `w$p.value` to extract a specific element.

Arrays and matrices

- Arrays are high dimensional vectors. A 2d-array is called a *matrix* and is one of the most used object in R.
- Internally, all arrays are stored as (1d) vectors, the differences are that
  - there is a `dim` attribute of an array;
  - we can easily extract a sub-array without thinking about the internal index;
  - there are many matrix arithmetic tools available, and they are much faster than doing loops over individual elements of a matrix.

Arrays and matrices

- Arrays are high dimensional vectors. A 2d-array is called a *matrix* and is one of the most used object in R.
- Internally, all arrays are stored as (1d) vectors, the differences are that
  - there is a `dim` attribute of an array;
  - we can easily extract a sub-array without thinking about the internal index;
  - there are many matrix arithmetic tools available, and they are much faster than doing loops over individual elements of a matrix.

Arrays and matrices

- Arrays are high dimensional vectors. A 2d-array is called a *matrix* and is one of the most used object in R.
- Internally, all arrays are stored as (1d) vectors, the differences are that
  - there is a `dim` attribute of an array;
  - we can easily extract a sub-array without thinking about the internal index;
  - there are many matrix arithmetic tools available, and they are much faster than doing loops over individual elements of a matrix.

Arrays and matrices

- Arrays are high dimensional vectors. A 2d-array is called a *matrix* and is one of the most used object in R.
- Internally, all arrays are stored as (1d) vectors, the differences are that
  - there is a `dim` attribute of an array;
  - we can easily extract a sub-array without thinking about the internal index;
  - there are many matrix arithmetic tools available, and they are much faster than doing loops over individual elements of a matrix.

Arrays and matrices

- Arrays are high dimensional vectors. A 2d-array is called a *matrix* and is one of the most used object in R.
- Internally, all arrays are stored as (1d) vectors, the differences are that
  - there is a `dim` attribute of an array;
  - we can easily extract a sub-array without thinking about the internal index;
  - there are many matrix arithmetic tools available, and they are much faster than doing loops over individual elements of a matrix.

Array/matrix manipulation

- Generate a 2d array (matrix): `x <- 1:12; mat1 <- array(x, dim = c(3, 4))`
- The last command is equivalent to `mat2 <- matrix(x, nrow = 3, ncol = 4)`
- But not equivalent to `mat3 <- matrix(x, nrow = 3, ncol = 4, byrow = TRUE)`.
- Constructing a high-dimensional array `A3 <- array(x, dim = c(3, 2, 2))`
- Concatenate two matrices: `cbind()` and `rbind()`.
- Merge two matrices (data frames) by a column (such as patient's unique ID): `merge(data1, data2, by="ID")`
- Or by a few columns: `merge(data1, data2, by=c("FirstName", "LastName", "DOB"))`
- Always pay attention to data merging! Computers can not deal with typos, lower/upper case, white space, nicknames, duplicates, etc. very well.

Array/matrix manipulation

- Generate a 2d array (matrix): `x <- 1:12; mat1 <- array(x, dim = c(3, 4))`
- The last command is equivalent to `mat2 <- matrix(x, nrow = 3, ncol = 4)`
- But not equivalent to `mat3 <- matrix(x, nrow = 3, ncol = 4, byrow = TRUE)`.
- Constructing a high-dimensional array `A3 <- array(x, dim = c(3, 2, 2))`
- Concatenate two matrices: `cbind()` and `rbind()`.
- Merge two matrices (data frames) by a column (such as patient's unique ID): `merge(data1, data2, by="ID")`
- Or by a few columns: `merge(data1, data2, by=c("FirstName", "LastName", "DOB"))`
- Always pay attention to data merging! Computers can not deal with typos, lower/upper case, white space, nicknames, duplicates, etc. very well.

Array/matrix manipulation

- Generate a 2d array (matrix): `x <- 1:12; mat1 <- array(x, dim = c(3, 4))`
- The last command is equivalent to `mat2 <- matrix(x, nrow = 3, ncol = 4)`
- But not equivalent to `mat3 <- matrix(x, nrow = 3, ncol = 4, byrow = TRUE)`.
- Constructing a high-dimensional array `A3 <- array(x, dim = c(3, 2, 2))`
- Concatenate two matrices: `cbind()` and `rbind()`.
- Merge two matrices (data frames) by a column (such as patient's unique ID): `merge(data1, data2, by="ID")`
- Or by a few columns: `merge(data1, data2, by=c("FirstName", "LastName", "DOB"))`
- Always pay attention to data merging! Computers can not deal with typos, lower/upper case, white space, nicknames, duplicates, etc. very well.

Array/matrix manipulation

- Generate a 2d array (matrix): `x <- 1:12; mat1 <- array(x, dim = c(3, 4))`
- The last command is equivalent to `mat2 <- matrix(x, nrow = 3, ncol = 4)`
- But not equivalent to `mat3 <- matrix(x, nrow = 3, ncol = 4, byrow = TRUE)`.
- Constructing a high-dimensional array `A3 <- array(x, dim = c(3, 2, 2))`
- Concatenate two matrices: `cbind()` and `rbind()`.
- Merge two matrices (data frames) by a column (such as patient's unique ID): `merge(data1, data2, by="ID")`
- Or by a few columns: `merge(data1, data2, by=c("FirstName", "LastName", "DOB"))`
- Always pay attention to data merging! Computers can not deal with typos, lower/upper case, white space, nicknames, duplicates, etc. very well.

Array/matrix manipulation

- Generate a 2d array (matrix): `x <- 1:12; mat1 <- array(x, dim = c(3, 4))`
- The last command is equivalent to `mat2 <- matrix(x, nrow = 3, ncol = 4)`
- But not equivalent to `mat3 <- matrix(x, nrow = 3, ncol = 4, byrow = TRUE)`.
- Constructing a high-dimensional array `A3 <- array(x, dim = c(3, 2, 2))`
- Concatenate two matrices: `cbind()` and `rbind()`.
- Merge two matrices (data frames) by a column (such as patient's unique ID): `merge(data1, data2, by="ID")`
- Or by a few columns: `merge(data1, data2, by=c("FirstName", "LastName", "DOB"))`
- Always pay attention to data merging! Computers can not deal with typos, lower/upper case, white space, nicknames, duplicates, etc. very well.

Array/matrix manipulation

- Generate a 2d array (matrix): `x <- 1:12; mat1 <- array(x, dim = c(3, 4))`
- The last command is equivalent to `mat2 <- matrix(x, nrow = 3, ncol = 4)`
- But not equivalent to `mat3 <- matrix(x, nrow = 3, ncol = 4, byrow = TRUE)`.
- Constructing a high-dimensional array `A3 <- array(x, dim = c(3, 2, 2))`
- Concatenate two matrices: `cbind()` and `rbind()`.
- Merge two matrices (data frames) by a column (such as patient's unique ID): `merge(data1, data2, by="ID")`
- Or by a few columns: `merge(data1, data2, by=c("FirstName", "LastName", "DOB"))`
- Always pay attention to data merging! Computers can not deal with typos, lower/upper case, white space, nicknames, duplicates, etc. very well.

Array/matrix manipulation

- Generate a 2d array (matrix): `x <- 1:12; mat1 <- array(x, dim = c(3, 4))`
- The last command is equivalent to `mat2 <- matrix(x, nrow = 3, ncol = 4)`
- But not equivalent to `mat3 <- matrix(x, nrow = 3, ncol = 4, byrow = TRUE)`.
- Constructing a high-dimensional array `A3 <- array(x, dim = c(3, 2, 2))`
- Concatenate two matrices: `cbind()` and `rbind()`.
- Merge two matrices (data frames) by a column (such as patient's unique ID): `merge(data1, data2, by="ID")`
- Or by a few columns: `merge(data1, data2, by=c("FirstName", "LastName", "DOB"))`
- Always pay attention to data merging! Computers can not deal with typos, lower/upper case, white space, nicknames, duplicates, etc. very well.

Array/matrix manipulation

- Generate a 2d array (matrix): `x <- 1:12; mat1 <- array(x, dim = c(3, 4))`
- The last command is equivalent to `mat2 <- matrix(x, nrow = 3, ncol = 4)`
- But not equivalent to `mat3 <- matrix(x, nrow = 3, ncol = 4, byrow = TRUE)`.
- Constructing a high-dimensional array `A3 <- array(x, dim = c(3, 2, 2))`
- Concatenate two matrices: `cbind()` and `rbind()`.
- Merge two matrices (data frames) by a column (such as patient's unique ID): `merge(data1, data2, by="ID")`
- Or by a few columns: `merge(data1, data2, by=c("FirstName", "LastName", "DOB"))`
- Always pay attention to data merging! Computers can not deal with typos, lower/upper case, white space, nicknames, duplicates, etc. very well.

Submatrix

- You can use two one-dim slicing to get a submatrix (or an element) from a matrix/array. `mat1[1:2, 2:3]`
- Like named array/lists, we can assign row and column names to matrices and use names to select submatrices.
- `rownames(mat1) <- c("a", "b", "c"); colnames(mat1) <- c("foo", "bar", "ham", "egg")`
- `mat1[c("a", "c"), c("egg", "foo")]`

Submatrix

- You can use two one-dim slicing to get a submatrix (or an element) from a matrix/array. `mat1[1:2, 2:3]`
- Like named array/lists, we can assign row and column names to matrices and use names to select submatrices.
- `rownames(mat1) <- c("a", "b", "c"); colnames(mat1) <- c("foo", "bar", "ham", "egg")`
- `mat1[c("a", "c"), c("egg", "foo")]`

Submatrix

- You can use two one-dim slicing to get a submatrix (or an element) from a matrix/array. `mat1[1:2, 2:3]`
- Like named array/lists, we can assign row and column names to matrices and use names to select submatrices.
- `rownames(mat1) <- c("a", "b", "c"); colnames(mat1) <- c("foo", "bar", "ham", "egg")`
- `mat1[c("a", "c"), c("egg", "foo")]`

Submatrix

- You can use two one-dim slicing to get a submatrix (or an element) from a matrix/array. `mat1[1:2, 2:3]`
- Like named array/lists, we can assign row and column names to matrices and use names to select submatrices.
- `rownames(mat1) <- c("a", "b", "c"); colnames(mat1) <- c("foo", "bar", "ham", "egg")`
- `mat1[c("a", "c"), c("egg", "foo")]`

Matrix calculations

- `mat1 %*% mat2;mat1 * mat2;eigen(mat1); solve(mat1[1:3, 1:3]);t(mat1).`
- Row/column operations: `rowSums(mat1); colMeans(mat2). apply(mat1, 1, sd).`
- Google is your best guide for finding a specific matrix operation in R, such as SVD, QR decomp, tensor product, etc.

Matrix calculations

- `mat1 %*% mat2`; `mat1 * mat2`; `eigen(mat1)`; `solve(mat1[1:3, 1:3])`; `t(mat1)`.
- Row/column operations: `rowSums(mat1)`; `colMeans(mat2)`. `apply(mat1, 1, sd)`.
- Google is your best guide for finding a specific matrix operation in R, such as SVD, QR decomp, tensor product, etc.

Matrix calculations

- `mat1 %*% mat2`; `mat1 * mat2`; `eigen(mat1)`; `solve(mat1[1:3, 1:3])`; `t(mat1)`.
- Row/column operations: `rowSums(mat1)`; `colMeans(mat2)`. `apply(mat1, 1, sd)`.
- Google is your best guide for finding a specific matrix operation in R, such as SVD, QR decomp, tensor product, etc.

Data frames

- Remember the main difference between vector and list: elements in one vector must have the same type; list can contain different objects.
- A data frame is a matrix that mixes different types of data (numbers, factors, strings, logical, etc).
- Convenient but not efficient for computation.
- You can convert a matrix into a data frame; the opposite is trickier. You will get a string matrix if the data frame contains non-numerical elements.

Data frames

- Remember the main difference between vector and list: elements in one vector must have the same type; list can contain different objects.
- A data frame is a matrix that mixes different types of data (numbers, factors, strings, logical, etc).
- Convenient but not efficient for computation.
- You can convert a matrix into a data frame; the opposite is trickier. You will get a string matrix if the data frame contains non-numerical elements.

Data frames

- Remember the main difference between vector and list: elements in one vector must have the same type; list can contain different objects.
- A data frame is a matrix that mixes different types of data (numbers, factors, strings, logical, etc).
- Convenient but not efficient for computation.
- You can convert a matrix into a data frame; the opposite is trickier. You will get a string matrix if the data frame contains non-numerical elements.

Data frames

- Remember the main difference between vector and list: elements in one vector must have the same type; list can contain different objects.
- A data frame is a matrix that mixes different types of data (numbers, factors, strings, logical, etc).
- Convenient but not efficient for computation.
- You can convert a matrix into a data frame; the opposite is trickier. You will get a string matrix if the data frame contains non-numerical elements.

Dealing with missing values

- Two types of missing values: `NA` and `NaN`.
- `max()`, `mean()` do not work as expected!
- Per-function solution: `na.rm=TRUE`. This switch is not always available.
- To remove missing values, you need to identity them. But the usual test, `x == NA` or `x == NaN`, does not work.
- Workaround: `is.na(x)`, `is.nan(x)`.
- `x <- x[!is.na(x)]` removes (destructively) all the NAs from `x`.
- For data frame, use `na.omit()` as a shortcut.
- Finally, there are ways to *impute* these missing values from the rest of your data based on statistical models.

Dealing with missing values

- Two types of missing values: `NA` and `NaN`.
- `max()`, `mean()` do not work as expected!
- Per-function solution: `na.rm=TRUE`. This switch is not always available.
- To remove missing values, you need to identity them. But the usual test, `x == NA` or `x == NaN`, does not work.
- Workaround: `is.na(x)`, `is.nan(x)`.
- `x <- x[!is.na(x)]` removes (destructively) all the NAs from `x`.
- For data frame, use `na.omit()` as a shortcut.
- Finally, there are ways to *impute* these missing values from the rest of your data based on statistical models.

Dealing with missing values

- Two types of missing values: `NA` and `NaN`.
- `max()`, `mean()` do not work as expected!
- Per-function solution: `na.rm=TRUE`. This switch is not always available.
- To remove missing values, you need to identity them. But the usual test, `x == NA` or `x == NaN`, does not work.
- Workaround: `is.na(x)`, `is.nan(x)`.
- `x <- x[!is.na(x)]` removes (destructively) all the NAs from `x`.
- For data frame, use `na.omit()` as a shortcut.
- Finally, there are ways to *impute* these missing values from the rest of your data based on statistical models.

Dealing with missing values

- Two types of missing values: `NA` and `NaN`.
- `max()`, `mean()` do not work as expected!
- Per-function solution: `na.rm=TRUE`. This switch is not always available.
- To remove missing values, you need to identity them. But the usual test, `x == NA` or `x == NaN`, does not work.
- Workaround: `is.na(x)`, `is.nan(x)`.
- `x <- x[!is.na(x)]` removes (destructively) all the NAs from `x`.
- For data frame, use `na.omit()` as a shortcut.
- Finally, there are ways to *impute* these missing values from the rest of your data based on statistical models.

Dealing with missing values

- Two types of missing values: `NA` and `NaN`.
- `max()`, `mean()` do not work as expected!
- Per-function solution: `na.rm=TRUE`. This switch is not always available.
- To remove missing values, you need to identity them. But the usual test, `x == NA` or `x == NaN`, does not work.
- Workaround: `is.na(x)`, `is.nan(x)`.
- `x <- x[!is.na(x)]` removes (destructively) all the NAs from `x`.
- For data frame, use `na.omit()` as a shortcut.
- Finally, there are ways to *impute* these missing values from the rest of your data based on statistical models.

Dealing with missing values

- Two types of missing values: `NA` and `NaN`.
- `max()`, `mean()` do not work as expected!
- Per-function solution: `na.rm=TRUE`. This switch is not always available.
- To remove missing values, you need to identity them. But the usual test, `x == NA` or `x == NaN`, does not work.
- Workaround: `is.na(x)`, `is.nan(x)`.
- `x <- x[!is.na(x)]` removes (destructively) all the NAs from `x`.
- For data frame, use `na.omit()` as a shortcut.
- Finally, there are ways to *impute* these missing values from the rest of your data based on statistical models.

Dealing with missing values

- Two types of missing values: `NA` and `NaN`.
- `max()`, `mean()` do not work as expected!
- Per-function solution: `na.rm=TRUE`. This switch is not always available.
- To remove missing values, you need to identity them. But the usual test, `x == NA` or `x == NaN`, does not work.
- Workaround: `is.na(x)`, `is.nan(x)`.
- `x <- x[!is.na(x)]` removes (destructively) all the NAs from `x`.
- For data frame, use `na.omit()` as a shortcut.
- Finally, there are ways to *impute* these missing values from the rest of your data based on statistical models.

Dealing with missing values

- Two types of missing values: `NA` and `NaN`.
- `max()`, `mean()` do not work as expected!
- Per-function solution: `na.rm=TRUE`. This switch is not always available.
- To remove missing values, you need to identity them. But the usual test, `x == NA` or `x == NaN`, does not work.
- Workaround: `is.na(x)`, `is.nan(x)`.
- `x <- x[!is.na(x)]` removes (destructively) all the NAs from `x`.
- For data frame, use `na.omit()` as a shortcut.
- Finally, there are ways to *impute* these missing values from the rest of your data based on statistical models.

Branching

- if-else structure.

```
    if (TEST) {
      ...
    } else {
      ...
    }
}
```

- A shortcut: `ifelse(TEST, outcome1, outcome2)`
- `switch(x, var1=outcome1, var2=outcome2, ..., other.outcome)`
- group commands by '{cmd1; cmd2; ...; cmdm}', the difference between one-liner and full grouped structure.

Loop

- The `for` loop.

```
for (i in counter.vector){
  ...
}
```

- Double/triple loops.
- `repeat`, `while` loop and the use of `break/next`.
- The `stop(MESSAGE)` function.
- The `foreach` loop.
- About efficiency. Matrix multiplication example.

Example of loop/branch

```
Y <- sample(c(1:10, 1:10))
Y.coded <- rep("low", 20)
for (i in 1:length(Y)){
  if (Y[i] <= 3) {
    Y.coded[i] <- "low"
  } else if (Y[i] <= 6) {
    Y.coded[i] <- "med"
  } else {
    Y.coded[i] <- "high"
  }
}
Y.coded <- ordered(Y.coded, levels=c("low",
        "med", "high"))
```

Logic functions

- Logical operators: $<$, $<=$, $>$, $>=$.
- Testing equality: $==$. This is another reason why $=$ should not be used as the assignment operator.
- Testing inequality: `!=`.
- Boolean operation: `!`, `&`, `|`.
- `all(vec)`, `any(vec)`; '&&', '||'.

Logic functions

- Logical operators: $<$, $<=$, $>$, $>=$.
- Testing equality: $==$. This is another reason why $=$ should not be used as the assignment operator.
- Testing inequality: `!=`.
- Boolean operation: `!`, `&`, `|`.
- `all(vec)`, `any(vec)`; '&&', '||'.

Logic functions

- Logical operators: $<$, $<=$, $>$, $>=$.
- Testing equality: $==$. This is another reason why $=$ should not be used as the assignment operator.
- Testing inequality: `!=`.
- Boolean operation: `!`, `&`, `|`.
- `all(vec)`, `any(vec)`; '&&', '||'.

Logic functions

- Logical operators: $<$, $<=$, $>$, $>=$.
- Testing equality: $==$. This is another reason why $=$ should not be used as the assignment operator.
- Testing inequality: `!=`.
- Boolean operation: `!`, `&`, `|`.
- `all(vec)`, `any(vec)`; '&&', '||'.

Logic functions

- Logical operators: $<$, $<=$, $>$, $>=$.
- Testing equality: $==$. This is another reason why $=$ should not be used as the assignment operator.
- Testing inequality: `!=`.
- Boolean operation: `!`, `&`, `|`.
- `all(vec)`, `any(vec)`; '&&', '||'.

Writing your own functions

- Basic usage: `myfunc <- function(arg1, arg2, ...) expression`
- The good habit: use `return()` statement.
- binary operators.
- Named arguments and defaults, calling conventions.
- The '...' argument.
- Assignments within functions, super-assignment operator, `assign("name", var)` and `get("name")`.
- Recursive functions. Not a good idea though.

Writing your own functions

- Basic usage: `myfunc <- function(arg1, arg2, ...) expression`
- The good habit: use `return()` statement.
- binary operators.
- Named arguments and defaults, calling conventions.
- The '...' argument.
- Assignments within functions, super-assignment operator, `assign("name", var)` and `get("name")`.
- Recursive functions. Not a good idea though.

Writing your own functions

- Basic usage: `myfunc <- function(arg1, arg2, ...)  expression`
- The good habit: use `return()` statement.
- binary operators.
- Named arguments and defaults, calling conventions.
- The '...' argument.
- Assignments within functions, super-assignment operator, `assign("name", var)` and `get("name")`.
- Recursive functions. Not a good idea though.

Writing your own functions

- Basic usage: `myfunc <- function(arg1, arg2, ...) expression`
- The good habit: use `return()` statement.
- binary operators.
- Named arguments and defaults, calling conventions.
- The '...' argument.
- Assignments within functions, super-assignment operator, `assign("name", var)` and `get("name")`.
- Recursive functions. Not a good idea though.

Writing your own functions

- Basic usage: `myfunc <- function(arg1, arg2, ...)  expression`
- The good habit: use `return()` statement.
- binary operators.
- Named arguments and defaults, calling conventions.
- The '...' argument.
- Assignments within functions, super-assignment operator, `assign("name", var)` and `get("name")`.
- Recursive functions. Not a good idea though.

Writing your own functions

- Basic usage: `myfunc <- function(arg1, arg2, ...)  expression`
- The good habit: use `return()` statement.
- binary operators.
- Named arguments and defaults, calling conventions.
- The '...' argument.
- Assignments within functions, super-assignment operator, `assign("name", var)` and `get("name")`.
- Recursive functions. Not a good idea though.

Writing your own functions

- Basic usage: `myfunc <- function(arg1, arg2, ...) expression`
- The good habit: use `return()` statement.
- binary operators.
- Named arguments and defaults, calling conventions.
- The '...' argument.
- Assignments within functions, super-assignment operator, `assign("name", var)` and `get("name")`.
- Recursive functions. Not a good idea though.

Function examples

```
twosam <- function(y1, y2, trim=0) {
  n1 <- length(y1); n2 <- length(y2)
  yb1 <- mean(y1, trim=trim)
  yb2 <- mean(y2, trim=trim)
  s1 <- var(y1); s2 <- var(y2)
  s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
  tst <- (yb1-yb2)/sqrt(s*(1/n1 + 1/n2))
  ## note the use of <<-
  N1 <<- n1
  return(tst)
}

x <- rnorm(8); y <- rnorm(10)
twosam(x, y, 0.2)
```

Graphical devices

- Interactive: `x11()`, `windows()`, and `quartz()`.
- Non-interactive: `postscript()`, `pdf()`, `png()`, `tiff()`, `jpeg`, etc.

Graphical devices

- Interactive: `x11()`, `windows()`, and `quartz()`.
- Non-interactive: `postscript()`, `pdf()`, `png()`, `tiff()`, `jpeg`, etc.

Plotting commands

- High level commands, such as the *generic* function `plot()`, which creates a new plot on the graphics devices (so you don't have to create a device in advance).
- Low level commands, such as `points()` and `lines()`. They do not create new plot, but add simple features to an existing one.

Plotting commands

- High level commands, such as the *generic* function `plot()`, which creates a new plot on the graphics devices (so you don't have to create a device in advance).
- Low level commands, such as `points()` and `lines()`. They do not create new plot, but add simple features to an existing one.

High level plotting commands

```
x <- 1:12; y <- 1:12 + rnorm(12)
plot(x,y)

A <- factor(rep(c("low", "med", "high"), each=4),
 levels=c("low", "med", "high"))

plot(A)
plot(A,y)

y2 <- 2*y + rnorm(12)
plot(y2 ~ x + A)
```

Low level plotting commands

- `points()`
- `lines()`
- `abline()` and it variants.
- `text()`
- `title()`
- `axis()`
- `legend()`

Low level plotting commands

- `points()`
- `lines()`
- `abline()` and it variants.
- `text()`
- `title()`
- `axis()`
- `legend()`

Low level plotting commands

- `points()`
- `lines()`
- `abline()` and it variants.
- `text()`
- `title()`
- `axis()`
- `legend()`

Low level plotting commands

- `points()`
- `lines()`
- `abline()` and it variants.
- `text()`
- `title()`
- `axis()`
- `legend()`

Low level plotting commands

- `points()`
- `lines()`
- `abline()` and it variants.
- `text()`
- `title()`
- `axis()`
- `legend()`

Low level plotting commands

- `points()`
- `lines()`
- `abline()` and it variants.
- `text()`
- `title()`
- `axis()`
- `legend()`

Low level plotting commands

- `points()`
- `lines()`
- `abline()` and it variants.
- `text()`
- `title()`
- `axis()`
- `legend()`

Low-level plotting commands

```
plot(x,y)
points(rnorm(40)+5, rnorm(40)+4, col=2, pch=2)
abline(v=8.0)
abline(h=5.0, col=3, lty=2, lwd=2)
title("An example")
```

Remarks on graphics

- You can put several subfigures in one figure (`par(mfrow=c(2,3))`).
- There are many more built-in high/low graphical procedures that you can use.
- Packages such as `lattice`, `ggplot2` provides even more graphical procedures. You can also make 3D interactive graphics, movies, etc.
- If we still have time ... `demo(graphics)`
- If we still have plenty of time ... `library(rgl); demo(rgl)`

Remarks on graphics

- You can put several subfigures in one figure (`par(mfrow=c(2,3))`).
- There are many more built-in high/low graphical procedures that you can use.
- Packages such as `lattice`, `ggplot2` provides even more graphical procedures. You can also make 3D interactive graphics, movies, etc.
- If we still have time ... `demo(graphics)`
- If we still have plenty of time ... `library(rgl); demo(rgl)`

Remarks on graphics

- You can put several subfigures in one figure (`par(mfrow=c(2,3))`).
- There are many more built-in high/low graphical procedures that you can use.
- Packages such as `lattice`, `ggplot2` provides even more graphical procedures. You can also make 3D interactive graphics, movies, etc.
- If we still have time ... `demo(graphics)`
- If we still have plenty of time ... `library(rgl); demo(rgl)`

Remarks on graphics

- You can put several subfigures in one figure (`par(mfrow=c(2,3))`).
- There are many more built-in high/low graphical procedures that you can use.
- Packages such as `lattice`, `ggplot2` provides even more graphical procedures. You can also make 3D interactive graphics, movies, etc.
- If we still have time ... `demo(graphics)`
- If we still have plenty of time ... `library(rgl); demo(rgl)`

Remarks on graphics

- You can put several subfigures in one figure (`par(mfrow=c(2,3))`).
- There are many more built-in high/low graphical procedures that you can use.
- Packages such as `lattice`, `ggplot2` provides even more graphical procedures. You can also make 3D interactive graphics, movies, etc.
- If we still have time ... `demo(graphics)`
- If we still have plenty of time ... `library(rgl); demo(rgl)`

Useful advanced topics not covered in this short course

- Object-oriented programming and functional programming. R is a strong object-oriented programming language in which *every* data type is an object. It also has nice support of functional programming. Google "Object-Oriented Programming, Functional Programming and R" by John M. Chambers.
- High-performance computing. R has many ways (packages) to utilize modern computing architecture, such as multi-core computer (with or without shared memory), cluster-computers with message passing interface (MPI), and GPU computing. Search "R, HighPerformanceComputing".

Useful advanced topics not covered in this short course

- Object-oriented programming and functional programming. R is a strong object-oriented programming language in which *every* data type is an object. It also has nice support of functional programming. Google “Object-Oriented Programming, Functional Programming and R” by John M. Chambers.
- High-performance computing. R has many ways (packages) to utilize modern computing architecture, such as multi-core computer (with or without shared memory), cluster-computers with message passing interface (MPI), and GPU computing. Search “R, HighPerformanceComputing”.

Useful advanced topics not covered in this short course (II)

- R can call C/Fortran in an R function to boost performance. It can also call Python functions (via `rPython` package) or other scripting language via `system()`. It can also be called from many other languages such as Perl or Python.
- R can talk to SQL databases. (`RODBC` package)
- R can even be used to develop GUI applications. `fgui`, `RGtk2`, etc.
- Making your own R package, which is essentially a bundle of R functions, help on these functions, some datasets. This is the standard way to share your method with your peers. Search "Writing R extensions" for a free reference manual.

Useful advanced topics not covered in this short course (II)

- R can call C/Fortran in an R function to boost performance. It can also call Python functions (via `rPython` package) or other scripting language via `system()`. It can also be called from many other languages such as Perl or Python.
- R can talk to SQL databases. (`RODBC` package)
- R can even be used to develop GUI applications. `fgui`, `RGtk2`, etc.
- Making your own R package, which is essentially a bundle of R functions, help on these functions, some datasets. This is the standard way to share your method with your peers. Search "Writing R extensions" for a free reference manual.

Useful advanced topics not covered in this short course (II)

- R can call C/Fortran in an R function to boost performance. It can also call Python functions (via `rPython` package) or other scripting language via `system()`. It can also be called from many other languages such as Perl or Python.
- R can talk to SQL databases. (`RODBC` package)
- R can even be used to develop GUI applications. `fgui`, `RGtk2`, etc.
- Making your own R package, which is essentially a bundle of R functions, help on these functions, some datasets. This is the standard way to share your method with your peers. Search “Writing R extensions” for a free reference manual.

Useful advanced topics not covered in this short course (II)

- R can call C/Fortran in an R function to boost performance. It can also call Python functions (via `rPython` package) or other scripting language via `system()`. It can also be called from many other languages such as Perl or Python.
- R can talk to SQL databases. (`RODBC` package)
- R can even be used to develop GUI applications. `fgui`, `RGtk2`, etc.
- Making your own R package, which is essentially a bundle of R functions, help on these functions, some datasets. This is the standard way to share your method with your peers. Search “Writing R extensions” for a free reference manual.

Bibliography I