Module 2: vectors and data frames

BIOSTAT 213: Introduction to Computing in the R Software Environment

Yea-Hung Chen, PhD, MS

Monday, September 30, 2019

Numeric objects, continued

Sequences

- to quickly generate sequences of integers, use the colon:

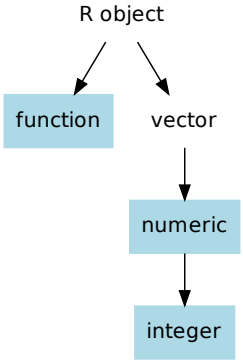
```
> 1:4
[1] 1 2 3 4
> 10:2
[1] 10  9  8  7  6  5  4  3  2
> -1:3
[1] -1  0  1  2  3
```

Integer objects

- as a side note, it turns out that there is a subtype of numeric objects known as integer objects:

```
> class(1:4)
[1] "integer"
> is.numeric(1:4)
[1] TRUE
> is.vector(1:4)
[1] TRUE
```

Integer objects



Integer objects

- interestingly, objects are not stored as integer if defined using `c()`:

```
> x<-c(1,2,3,4)
> class(x)
[1] "numeric"
> x<-1:4
> class(x)
[1] "integer"
```

Sequences

- for sequences in general, use `seq()`:

```
> seq(from=0,to=6,by=0.5)
 [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0
> seq(0,6,0.5)
 [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0
```

Sequences

- for a decreasing sequence, specify a negative argument to by:

```
> seq(10,-10,-2)
 [1]  10   8   6   4   2   0  -2  -4  -6  -8 -10
```

Sequences

- the `to` argument does not need be the exact endpoint:

```
> seq(2.2,6.1,0.48)
[1] 2.20 2.68 3.16 3.64 4.12 4.60 5.08 5.56 6.04
```

Sequences

- for a specific number of results, use the `length.out` argument instead of by:

```
> seq(from=2,to=3,length.out=5)
[1] 2.00 2.25 2.50 2.75 3.00
> seq(2,3,length.out=5)
[1] 2.00 2.25 2.50 2.75 3.00
```

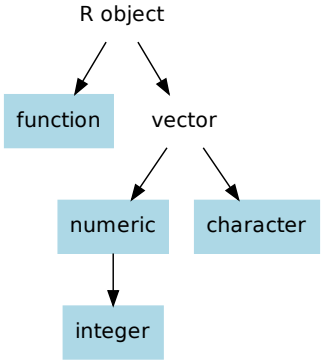
Character objects

Character objects

```
> this.course<-'Biostat 213'
> class(this.course)
[1] "character"
> is.vector(this.course)
[1] TRUE
```

- character objects store text
- like numeric objects, character objects are vectors

Character objects



Character objects

```
> this.course<-'Biostat 213'
```

- you must always use quotes
- you can use single or double quotes, but you must end with whatever you start with

Character objects

- you may put single quotes inside of double quotes, or vice versa:

```
> note1<-"it's ok to put single quotes inside double quotes"
> note2<-'yea-hung said, "you can put double inside single"'
```

Character objects

- for multiple elements, use `c()`, as seen last week with numeric objects:

```
> last.name<-c('Washington','Adams','Jefferson','Madison')
> last.name
[1] "Washington" "Adams"      "Jefferson"  "Madison"
> class(last.name)
[1] "character"
```

Character objects

- not a numeric object:

```
> numeric.no<-c('3.2','4.4','-1.2')
> numeric.no
[1] "3.2"  "4.4"  "-1.2"
> class(numeric.no)
[1] "character"
```

Character manipulation

- for all-lower-case conversion, use `tolower()`:

```
> last.name
[1] "Washington" "Adams"      "Jefferson"  "Madison"
> tolower(last.name)
[1] "washington" "adams"      "jefferson"  "madison"
```

- note that as generally true in R, the object is not automatically redefined:

```
> last.name
[1] "Washington" "Adams"      "Jefferson"  "Madison"
```

- what could you do if you wanted `last.name` to be redefined?

Character manipulation

- for all-upper-case conversion, use `toupper()`:

```
> last.name
[1] "Washington" "Adams"      "Jefferson"  "Madison"
> toupper(last.name)
[1] "WASHINGTON" "ADAMS"      "JEFFERSON"  "MADISON"
```

Character manipulation

- to connect character strings, use `paste()`:

```
> paste('hello','world',sep=' ')
[1] "hello world"
> paste('hello','world')
[1] "hello world"
```

- the `sep` argument indicates the separation

Character manipulation

- `paste()` works even if the arguments are of unequal length:

```
> paste('res',1:5,sep='')
[1] "res1" "res2" "res3" "res4" "res5"
```

- as with mathematical operations, R will attempt to recycle the shorter vector

Character manipulation

- the collapse argument will paste the elements in a vector:

```
> not.a.sentence<-c('Welcome','to','Biostat','213')
> not.a.sentence
[1] "Welcome" "to"      "Biostat" "213"
> paste(not.a.sentence,collapse=' ')
[1] "Welcome to Biostat 213"
```

- what's the difference between the results above?

Character manipulation

- for the number of characters, use `nchar()`:

```
> last.name
[1] "Washington" "Adams"      "Jefferson"  "Madison"
> nchar(last.name)
[1] 10  5  9  7
```

Character manipulation

- to select portions of text, use `substr()` (short for substring):

```
> substr(last.name,start=1,stop=3)
[1] "Was" "Ada" "Jef" "Mad"
> substr(last.name,1,5)
[1] "Washi" "Adams" "Jeffe" "Madis"
```

Character to numeric

- to convert character objects to numeric objects, use `as.numeric()`:

```
> numeric.no
[1] "3.2"  "4.4"  "-1.2"
> class(numeric.no)
[1] "character"
> numeric.yes<-as.numeric(numeric.no)
> numeric.yes
[1]  3.2  4.4 -1.2
> class(numeric.yes)
[1] "numeric"
```

Character to numeric

- note that `as.numeric()` will return missing values if there are not numbers in the character object

```
> last.name
[1] "Washington" "Adams"      "Jefferson"  "Madison"
> as.numeric(last.name)
Warning: NAs introduced by coercion
[1] NA NA NA NA
```

- we will discuss missing values further on a later date

Numeric to character

- to convert numeric objects to character objects, use `as.character()`:

```
> phone<-c(4154762300,4154762310)
> phone<-as.character(phone)
> phone
[1] "4154762300" "4154762310"
> class(phone)
[1] "character"
> phone<-paste(substr(phone,1,3),substr(phone,4,6),
+              substr(phone,7,10),sep='-')
> phone
[1] "415-476-2300" "415-476-2310"
```

Useful functions for vectors

Lengths

```
> radius
[1]  2 15  5  8 12 16
```

- values in vectors are often referred to as elements
- to count the number of elements in a vector, use `length()`:

```
> length(radius)
[1] 6
```

Sorting

- to sort objects, use `sort()`:

```
> radius
[1]  2 15  5  8 12 16
> sort(radius)
[1]  2  5  8 12 15 16
> sort(radius,decreasing=TRUE)
[1] 16 15 12  8  5  2
> radius
[1]  2 15  5  8 12 16
```

Sorting

```
> last.name
[1] "Washington" "Adams"      "Jefferson"  "Madison"
> sort(last.name)
[1] "Adams"      "Jefferson"  "Madison"    "Washington"
> last.name
[1] "Washington" "Adams"      "Jefferson"  "Madison"
```

Order

- the `order()` function provides the positions that would sort the object:

```
> radius
[1]  2 15  5  8 12 16
> order(radius)
[1] 1 3 4 5 2 6
> sort(radius)
[1]  2  5  8 12 15 16
```

Repetition

- to repeat values, use `rep()`:

```
> rep(x=1,times=5)
[1] 1 1 1 1 1
> rep(1,5)
[1] 1 1 1 1 1
> rep('hello',5)
[1] "hello" "hello" "hello" "hello" "hello"
```

Repetition

- the first argument of `rep()` can contain more than one element:

```
> rep(1:5,2)
 [1] 1 2 3 4 5 1 2 3 4 5
```

- the `each` argument will repeat values with the original order preserved:

```
> rep(1:5,each=2)
 [1] 1 1 2 2 3 3 4 4 5 5
> rep(1:5,times=2)
 [1] 1 2 3 4 5 1 2 3 4 5
```

Duplicates

- to identify duplicates, use `duplicate()`:

```
> duplicated(c(5,2,2,5,2,5))
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

Unique values

- to list unique values, use `unique()`:

```
> unique(c(5,2,2,5,2,5))
[1] 5 2
```

Intersect

- to check for intersect, use `is.element()`:

```
> is.element(5,1:6)
[1] TRUE
> is.element(1:6,5)
[1] FALSE FALSE FALSE FALSE  TRUE FALSE
```

Intersect

```
> some.animals<-c('elephant','dog','cat','giraffe')
> more.animals<-c('dog','fish','bird')
> is.element(more.animals,some.animals)
[1]  TRUE FALSE FALSE
```

Working directories

Working directories

```
> radius<-c(2,15,5,8,12,16)
```

- so far, we have been just "manually" entering in data
- in general, we will be dealing with data stored in external data files

Working directories

- to open ("load" or "import") an external data file in R, it is useful to specify ("set" or "change") the working directory
- the working directory is the folder on your computer that R will use when opening and saving files
- it is also the directory where R will save your objects (more on this later)

Working directories

- on your computer, you probably have different folders for different projects
- for example, I might have the following folders on my computer:
  - `tanzania`: a folder for a project in Tanzania
  - `laos`: a folder for a project in Laos
- so, when working on the Laos project it would make sense for me to tell R that I want the working directory to be `laos`

Changing the working directory

- if you already have a script in the desired working directory, an easy way to change the working directory is as follows:
  1. Close RStudio, if it is already open.
  2. Use your computer's file manager (for example, Finder on Apple or File Explorer on Windows) to navigate to the desired working directory.
  3. Open a script (for example, double-click). The script should open in RStudio, if file associations are properly set.
- RStudio should automatically set the working directory

Changing the working directory

- if scripts do not automatically open in RStudio on your computer, consider the following instructions:
  - instructions for changing file associations on macOS
  - instructions for changing file associations on Windows

Changing the working directory

- another way change the working directory is: Session > Set Working Directory > Choose Directory...
- equivalently, you can use `setwd()` (short for set working directory):

```
> setwd('/home/yea-hung/research/laos')
```

Checking the working directory

- the working directory is always displayed in RStudio's Console pane
- alternatively, use: `getwd()` (short for get working directory)

Data frames

CSV files

- CSV (comma-separated values) file: a plain-text data file, with values separated by commas

The NHANES data file

- `nhanes.csv` contains data from the 2015–2016 implementation of the National Health and Nutrition Examination Survey (NHANES)
- the file contains "original" variables as well as some variables I added
- descriptions of the original variables are available here
- I restricted the data to individuals 18–69 years of age ("adults")

Importing CSV files

- to import a CSV file, use the `read.csv()` function:

```
> ii<-read.csv('nhanes.csv')
```

Importing CSV files

```
> ii<-read.csv('nhanes.csv')
```

- in the example above:
  - `read.csv`: the name of the function
  - the parentheses: a space to type in the filename
  - `nhanes.csv`: the filename, which must be in quotes
  - `<-`: a left arrow, indicating that you want to save the data
  - `ii`: an arbitrary name for the data
- as with the examples last week, you can imagine placing (`<-`) some food ingredient in a jar and then labeling the jar with some arbitrary name (`ii`)

Importing CSV files

```
> ii<-read.csv('nhanes.csv')
```

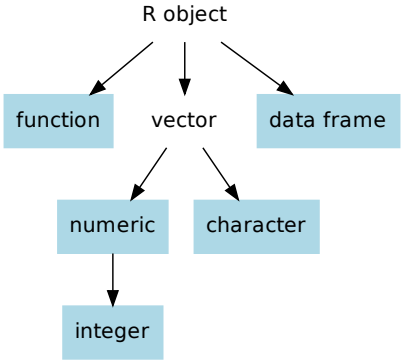
- this works because I set the working directory to be the directory on my computer that contains `nhanes.csv`

Data frames

```
> class(ii)
[1] "data.frame"
```

- ii is a data frame

Data frames



Data frames

- data frame: two-dimensional object, analogous to spreadsheets
- data frames have column names
- they can also have row names

Exploring data frames

- in RStudio, you can view the data by clicking on the object name in the Environment pane

Exploring data frames

- `names()` for column names
- `head()` for the first few rows
- `tail()` for the last few rows
- continuing the jar analogy, `head()` and `tail()` are like asking for just a pinch from the jar (because it is generally too cumbersome to view the entire data set in the console)

Exploring data frames

- `dim()` for the number of rows and columns
- `nrow()` for the number of rows
- `ncol()` for the number of columns

Exploring data frames

```
> dim(ii)
[1] 4209  691
> nrow(ii)
[1] 4209
> ncol(ii)
[1] 691
```

Accessing variables in data frames

- the `mean()` function will find the mean of a set of numbers

```
> mean(ridageyr)
```

- but, the above results in an error, even though there is a variable called `ridageyr` in the NHANES data

Accessing variables in data frames

```
> mean(ii$ridageyr)
[1] 42.93751
```

- in the example above:
  - `mean`: a function for finding the arithmetic mean
  - the parentheses: a space for typing in the set of numbers (in this case, we are referring to a variable in the data frame)
  - `ii`: the name of the data frame
  - `$`: indicates that `ridageyr` is a column in `ii`
  - `ridageyr`: the variable name
- I often refer to the pattern above as dollar-sign notation

Accessing variables in data frames

- continuing the jar analogy: the dollar sign allows you to systematically use just part of the jar
- you can imagine having a jar of jelly beans, labeled `jellybeans`
- in R's language, `jellybeans$licorice` might refer to all the licorice jelly beans

Adding variables to data frames

- to add variables to the data frame, use object assignment:

```
> ii$id<-1:nrow(ii)
```

- in the example above:
  - `1:nrow(ii)`: sequential integers
  - `<-`: a left arrow, indicating object assignment
  - `ii`: the name of the data frame
  - `$`: the dollar sign
  - `id`: the variable name (the variable does not yet exist in `ii`)

Adding variables to data frames

- a check of the work:

```
> head(ii$id)
[1] 1 2 3 4 5 6
> tail(ii$id)
[1] 4204 4205 4206 4207 4208 4209
```

- this example also shows that `head()` and `tail()` work on vectors, not just on data frames

Adding variables to data frames

- if, for some reason, you want to add a single value as a variable to the data frame, you can just specify the single value:

```
> ii$one<-1
```

- in other words, the following is equivalent but unnecessary:

```
> ii$one<-rep(1,length(ii$ridageyr))
```

Replacing variables in data frames

- to replace variables in data frames, use object assignment:

```
> head(ii$ridageyr)
[1] 62 53 56 42 22 32
> ii$ridageyr<-45
> head(ii$ridageyr)
[1] 45 45 45 45 45 45
```

- notice that R does not provide any warning

Data frame variables and object types

- each variable in a data frame has a class:

```
> head(ii$bmxbmi)
[1] 27.8 30.8 42.4 20.3 28.0 28.2
> class(ii$bmxbmi)
[1] "numeric"
```

Manually creating data frames

- to manually create a data frame, use `data.frame()`:

```
> students<-data.frame(first=c('Bob','Mary','Jane'),
+                      hw1=c(86,92,95),hw2=c(92,84,93))
> students
  first hw1 hw2
1   Bob  86  92
2  Mary  92  84
3  Jane  95  93
```

- each argument represent a column (variable) of the desired data frame

Saving work

Suggestions for saving work

- if certain conditions are met (more on this in the optional slides), objects that you assign will be available to you later, even after you quit
- however, I discourage relying on this as a strategy for saving work
- instead, I suggest saving work by saving code (to scripts)
- you can then re-run your scripts to recover objects
- a reason for this suggestion is that even if you recover previously assigned objects, you might not remember what changes you've made to those objects

Suggestions for saving work

- if you have scripts that take a long time to run (highly unlikely for now), you may consider saving your workspace
- the workspace is a generic term to refers to all of the objects that you have defined in any given session
- to save the workspace, use `save.image()`:

```
> save.image('all_objects.RData')
```

- `save.image()` will save the objects to a file, known as an image
- images should have a `.RData` extension
- otherwise, the filename is arbitrary

Suggestions for saving work

- to open the image at a later time, use `load()`:

```
> load('all_objects.RData')
```

- equivalently, use: File » Open File

Suggestions for saving work

- if you have scripts that take a long time to run (highly unlikely for now), you may alternatively consider saving a single object
- to save a single object, use `saveRDS()`:

```
> saveRDS(object=some_object,file='single_object.rds')
```

- or, without argument names:

```
> saveRDS(some_object,'single_object.rds')
```

- the file should have a `.rds` extension
- otherwise, the filename is arbitrary

Suggestions for saving work

- to open the file at a later time, use `readRDS()`:

```
> some_object<-readRDS('single_object.rds')
```

- notice that you must use object assignment
- as usual, the name on the left-hand side of the object assignment is arbitrary

Summary

- I suggest saving work by saving and re-running scripts
- if you have scripts that take a long time to run (highly unlikely for now), you may consider saving objects using `save.image()` or `saveRDS()`

Optional slides: hidden images

- as noted last week, when you quit, you will be prompted to indicate whether you want to save the workspace image
- if you select Save, the workspace will be saved to an image simply named `.RData` (in other words, with no preceding filename!)
- this is also what happens if you use `save.image()` without any arguments

Optional slides: hidden images

- if you return to RStudio and set the working directory via the method of opening an existing script, RStudio will automatically set the working directory and automatically open the `.RData` in the working directory, if one exists

Optional slides: hidden images

- if you set the working directory via the drop-down menu or by using `setwd()`, you may need to manually open `.RData`
- a challenge here though is that your operating system may hide that file by default, so you may not see it if you browse for it
- to elaborate, in general, files with a filename that begins with a dot (such as `.RData`) are often known as dot files
- many operating systems hide dot files by default, but it is generally possible to change that default

Exercise

Exercise

You may work in groups of 2–3. Save your code in a script.

1. Identify the folder on your computer that you would like to use for the day. Set the working directory to this folder, using one of the methods discussed today.
2. Download the NHANES data file, `nhanes.csv` from the Collaborative Learning Environment. Depending on your operating system or browser, you may need to right-click. Save the file to the folder identified above.
3. Import the data into R.

More on the next slide...

Exercise

4. Explore the data frame using RStudio's built-in data viewer.
5. Explore the data frame using `names()`, `head()`, `tail()`, and `dim()`.

More on the next slide...

Exercise

6. The variable `bmxwt` stores weight in kilograms. Use this variable to add a new variable to the data frame for weight in pounds (1 kg = 2.20462 lb).
7. The variable `bmxht` stores height in centimeters. Use this variable to add a new variable to the data frame for height in feet (1 cm = 0.0328084 ft).

More on the next slide...

Exercise

8. A popular anecdote is that Gauss (or Einstein, or some other mathematical genius), was "punished" in class and asked to sum every integer between 1 and 100. As the anecdote goes, Gauss realized that he could do this via the following formula: $(100 \times 101)/2$. Compute the sum in R, without using this formula, and without manually typing all the integers between 1 and 100. Compute the sum in R again, this time using the formula. Verify that the two answers match. Important hint: you may use the `sum()` function to find the sum of numbers in a vector. We'll discuss statistical functions further next week.

More on the next slide...

Exercise

9. Write code to find the first 10 powers of 2, beginning with 1 (ie, 2^1, 2^2, etc), preferably all in one line. Hint: you may need some parentheses.
10. Type `1:10+4`. In another line, type `1:(10+4)`. Run these two lines of code, and compare the results. What do the results suggest about orders of operation? In other words, is `:` or `+` prioritized first?

More on the next slide...

Exercise

11. R contains a predefined character object, `letters`, that contains the 26 letters of the English alphabet. The length of the vector is 26: each member of the vector is one of the 26 letters of the alphabet. Write code that will combine the letters together into a single character string, representing the alphabet, without spaces.

12. I did not write the message on slide 9 from Module 1 by writing the sentence 10 times (or copying and pasting 9 times). Write code that can generate that message.