R Tutorial at the WZB

2 - R Basics I

Markus Konrad

November 01, 2018

Today's schedule

1. Review of last week's tasks
2. Objects and assignments
3. Fundamental data structures and operations
4. Functions

Review of last week's tasks

Solution for tasks #1

now online on
https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/

R Basics I

Objects and assignments

- an object is bascially anything that can be named and assigned a value

```
x <- 2
```

- creates an object named `x` and assigns the value `2`
- `<-` is an assignment operator *
- the whole line is formally called a statement
- it is evaluated (interpreted) once you press `ENTER`
- a shortcut for writing `<-` in RStudio is `ALT+-`

A little shortcut used in the slides:

```
(y <- x + 3)
```

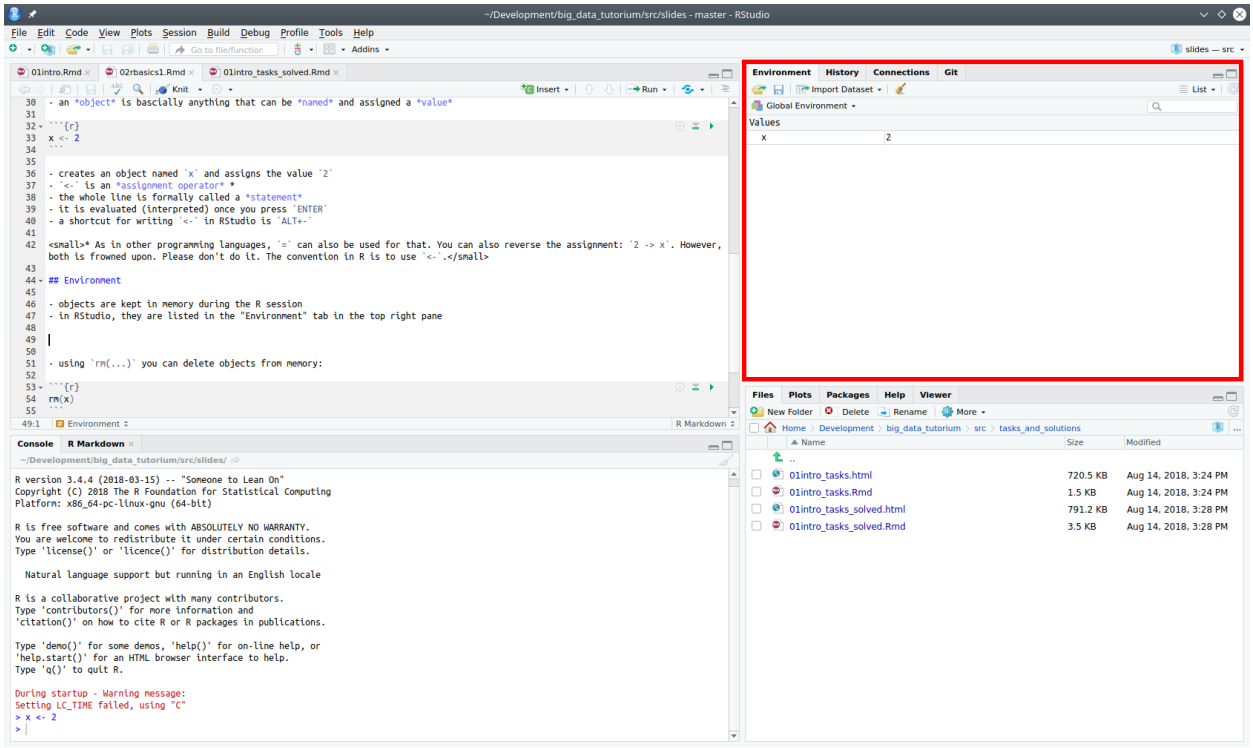
```
## [1] 5
```

→ parentheses around assignment directly report the result underneath

* As in other programming languages, `=` can also be used for that. You can also reverse the assignment: `2 -> x`. However, both is frowned upon. Please don't do it. The convention in R is to use `<-`.

Environment

- objects are kept in memory during the R session
- in RStudio, they are listed in the "Environment" tab in the top right pane



- using `rm(...)` you can delete objects from memory:

```
rm(x)
```

"Objects" vs. "Variables"

In Computer Science and programming, such objects are often called "variables", because they can change their value while a program or script is running.

This is quite different from what variables mean in Social Sciences. Please don't confuse this.

Naming things

You can name objects as you like. However, there are some rules:

> Identifiers consist of a sequence of letters, digits, the period (‘.’) and the underscore. They must not start with a digit or an underscore, or with a period followed by a digit.
>
> — R Language Definition

So this is fine:

```
another_one <- 1
another_2 <- 2
my.object.2... <- 123
```

Naming things

You can name objects as you like. However, there some rules:

> Identifiers consist of a sequence of letters, digits, the period (‘.’) and the underscore. They must not start with a digit or an underscore, or with a period followed by a digit.
>
> — R Language Definition

This is not:

```
_fail <- 1
## Error: unexpected input in "_"
```

```
2fail_again <- 'testing...'
## Error: unexpected symbol in "2fail_again"
```

Naming things

Which of these are **not** valid object names?

1. `coffee2go`
2. `x__`
3. `x__._`
4. `_`
5. `._`
6. `.x`
7. `.1`
8. `.1coffee`
9. `güzel`
10. `2go`

Making comments

Everything starting with a # will be ignored by R. This is used for making source code comments or temporarily disabling some code:

```
# comment above
c <- a + b   # comment beside

# d <- c * b   # I disabled this statement
```

When working with the console, that's not really useful. But we'll need it later when we'll write real R scripts.

Fundamental data structures and operations

Vectors

R's most fundamental data structure: the vector. It stores **one-dimensional data** of a **single type**.

A vector is created with `c(...)`.

Four types of vectors that you'll need:

1. Numeric vector: `c(3.2, 0, -8.6, 3e-2)` *
2. Logical vector: `c(FALSE, TRUE, TRUE, FALSE, TRUE)`
3. Character string vector: `c("she", "said", '"ok?"')`
4. Vector of factors: `factor(c("B", "B", "A", "C", "B"))`

Single-element vectors are called scalars:
`x <- 3` is the same as `x <- c(3)`

* Special type of numeric vector – the integer vector: `c(3L, 20L, -1L, 3L)`

Arithmetical operations

All arithmetical operations on vectors are performed **element-wise**:

```
a <- c(1.2, 0, -0.6, 0.25)
b <- c(2, 3, 10, 0)
a + b
```

```
## [1] 3.20 3.00 9.40 0.25
```

```
a - b
```

```
## [1]  -0.80  -3.00 -10.60   0.25
```

```
a * b
```

```
## [1]  2.4  0.0 -6.0  0.0
```

```
a / b
```

```
## [1]  0.60  0.00 -0.06   Inf
```

(note how the division by zero produced an infinite number)

Arithmetical operations

You can create more complex expressions:

```
a^2 + b^2
```

```
## [1]   5.4400   9.0000 100.3600   0.0625
```

```
a + b / 2
```

```
## [1] 2.20 1.50 4.40 0.25
```

```
(a + b) / 2
```

```
## [1] 1.600 1.500 4.700 0.125
```

Vector recycling

What if the sizes of the vectors do not match?

```
a
```

```
## [1]  1.20  0.00 -0.60  0.25
```

```
c <- c(2, 3, 10)
a * c
```

```
## Warning in a * c: longer object length is not a multiple of shorter objec
## length
```

```
## [1]  2.4  0.0 -6.0  0.5
```

A warning is issued, still both vectors are multiplied; the shorter vector c is "recycled":

- $a_1 c_1 = 1.2 \cdot 2 = 2.4$
- $a_2 c_2 = 0 \cdot 3 = 0$
- $a_3 c_3 = -0.6 \cdot 10 = -6$
- $\mathbf{a_4 c_1 = 0.25 \cdot 2 = 0.5}$

Vector recycling

No warning is issued, if the larger vector is a multiple of the smaller vector:

```
a
```

```
## [1]  1.20  0.00 -0.60  0.25
```

```
d <- c(2, 3)
a * d
```

```
## [1]  2.40  0.00 -1.20  0.75
```

→ `d` is recycled to form `c(2, 3, 2, 3)`

Multiplication by scalar is also a form of vector "recycling":

```
e <- 100
a * e
```

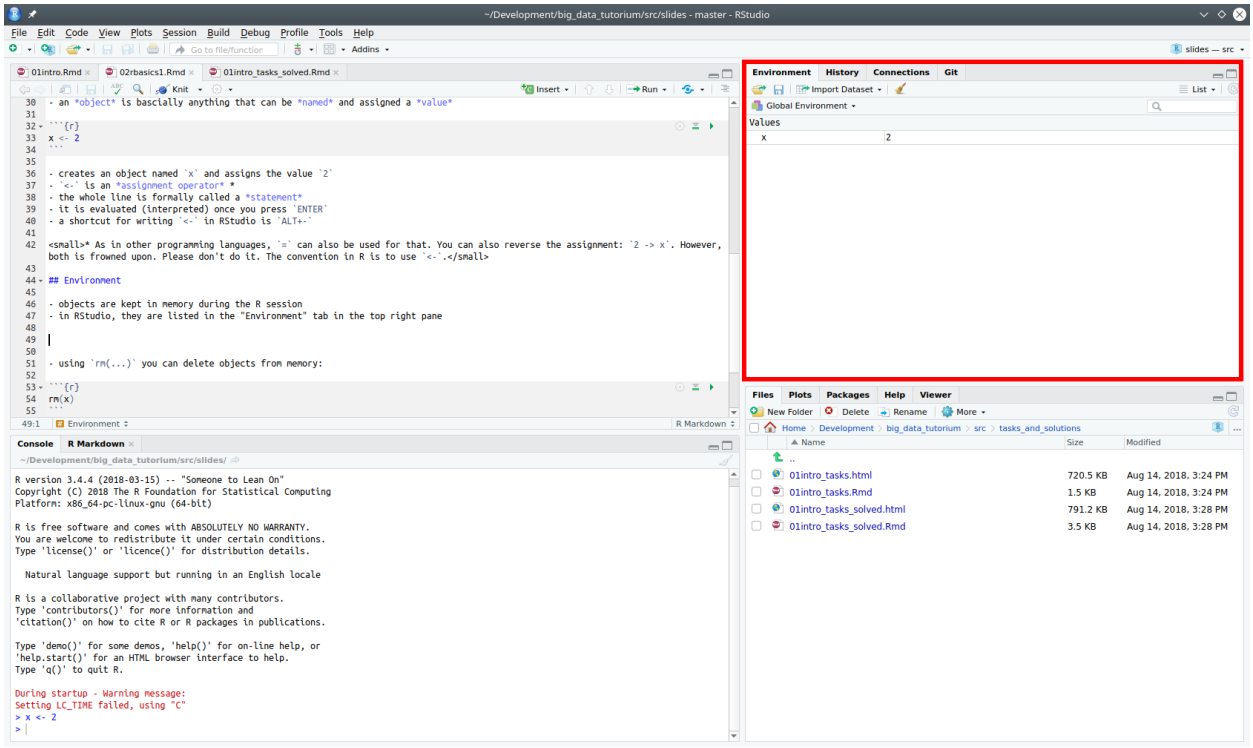
```
## [1] 120   0 -60  25
```

→ `e` is recycled to form `c(100, 100, 100, 100)`

Side note: Working with RStudio

The command history

- all commands entered in the console are recorded to "command history"
- whole history visible in "History" tab on top right



Side note: Working with RStudio

Using the history in the console

- **you can browse through the history in the Console using the `UP` and `DOWN` keys**
- you can also use `CTRL+R` to interactively search through the history
- this saves a lot of typing, e.g. when you want to correct a previous command

Side note: Working with RStudio

Autocompletion

- auto-completion of commands suggests objects after writing a few letters
- **use the TAB key to activate auto-completion**
- this saves a lot of typing when you have long object names

```
my_long_object_name <- c(1, 2, 3)
```

- now just type `my_` in the console and press TAB - what happens when you type `me` and then activate auto-completion?

Vector concatenation

The function `c(...)` can also be used to concatenate vectors:

```
a
```

```
## [1]  1.20  0.00 -0.60  0.25
```

```
b
```

```
## [1]  2  3 10  0
```

```
c(a, b)
```

```
## [1]  1.20  0.00 -0.60  0.25  2.00  3.00 10.00  0.00
```

Vector concatenation

Vectors are always flat, i.e. one-dimensional. Hence, nesting the `c()` function will concatenate several vectors:

```
c(1, 2, 3, c(4, 5))
```

```
## [1] 1 2 3 4 5
```

You can append (or prepend) data to an already defined vector by appending and then re-assigning it to the same object name:

```
a
```

```
## [1]  1.20  0.00 -0.60  0.25
```

```
a <- c(a, 1, 2, 3)
```

```
a
```

```
## [1]  1.20  0.00 -0.60  0.25  1.00  2.00  3.00
```

Logical vectors

Logical vectors or boolean vectors contain only binary logical values TRUE and FALSE*. Logical operations like AND, OR, XOR can be applied.

```
is_female <- c(FALSE, TRUE, TRUE, FALSE, TRUE)
younger30 <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
```

Logical AND:

```
is_female & younger30
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE
```

Logical OR:

```
is_female | younger30
```

```
## [1]  TRUE  TRUE  TRUE FALSE  TRUE
```

Use case: Select observations according to several criteria, i.e. "participant is female and younger than 30".

* R defines the abbreviations T / F for TRUE / FALSE. Please don't use them. You may confuse them with a variable/object T or F.

Character string vectors

Character strings are used to store textual data.

```
s <- c("hello", "string")
s
```

```
## [1] "hello"  "string"
```

You can use either double (`"`) or single (`'`) quotation marks to denote a string.

```
t <- c("world", 'example')
```

There are many functions to operate on strings, for example `paste`:

```
paste(s, t)
```

```
## [1] "hello world"    "string example"
```

Use cases:

- storing names, addresses, comments, IDs, etc.
- performing quantitative text analysis

Factors

Factors are used to store categorical variables. They can be initialized with numeric or string vectors:

```
group <- factor(c("B", "B", "A", "C", "B"))
group
```

```
## [1] B B A C B
## Levels: A B C
```

Internally, each value is mapped to an integer (a category "code").

```
as.integer(group)
```

```
## [1] 2 2 1 3 2
```

Factor levels

```
group
```

```
## [1] B B A C B
## Levels: A B C
```

Levels denote the available categories for a variable. `levels(x)` returns the levels of a factor object `x`:

```
levels(group)
```

```
## [1] "A" "B" "C"
```

Levels can be specified explicitely:

```
factor(c('yes', 'no', 'yes', 'yes'), levels = c('yes', 'no', 'maybe'))
```

```
## [1] yes no  yes yes
## Levels: yes no maybe
```

Ordered factors

Ordinal variables can be stored as ordered factors with `ordered(...)`

```
consent <- ordered(c('low', 'low', 'very low', 'high'),
                   levels = c('very low', 'low', 'neutral', 'high', 'very hi
consent
```

```
## [1] low      low      very low high
## Levels: very low < low < neutral < high < very high
```

```
levels(consent)
```

```
## [1] "very low"  "low"       "neutral"   "high"      "very high"
```

You can also use numeric values as factors:

```
score <- ordered(c(3, 1, 1, 5), levels = 1:5)
score
```

```
## [1] 3 1 1 5
## Levels: 1 < 2 < 3 < 4 < 5
```

Type conversion

Conversion is the **explicit** process of changing a (vector's) data type.

Explicit type conversion can be done with a family of `as.<TYPE>(...)` functions, e.g. to convert a factor to an integer vector:

```
consent
```

```
## [1] low      low      very low high
## Levels: very low < low < neutral < high < very high
```

```
as.integer(consent)
```

```
## [1] 2 2 1 4
```

Or a logical vector to an integer vector:

```
younger30
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE
```

```
as.integer(younger30)
```

```
## [1] 1 1 0 0 0
```

Type conversion

You'll often have to convert strings (e.g read from a file) to numbers. The following is a character string vector:

```
(values <- c('1.2', '-3', "0.001"))
```

```
## [1] "1.2"   "-3"    "0.001"
```

Arithmetic operations fail, because we can't multiply a string with a number:

```
values * 3
## Error in values * 3 : non-numeric argument to binary operator
```

We have to convert the string vector to a numeric vector first:

```
as.numeric(values) * 3
```

```
## [1]  3.600 -9.000  0.003
```

Type coercion

Coercion is the **implicit** process of a (vector's) data type being changed.

```
c(0.2, TRUE, 4)
```

```
## [1] 0.2 1.0 4.0
```

> All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be coerced to the most flexible type. Types from least to most flexible are: logical, integer, double, and character.
>
> — Wickham 2014: Advanced R

```
c(0.2, TRUE, 4, '3.1')
```

```
## [1] "0.2"  "TRUE" "4"    "3.1"
```

Type coercion

Coercion happens automatically, esp. when using mathematical functions, which can come in handy:

```
younger30
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE
```

```
sum(younger30)
```

```
## [1] 2
```

→ `sum()` here counts all occurrences of `TRUE` because `TRUE` becomes `1` and `FALSE` becomes `0` when coerced to a numeric type.

Type coercion

Factors are not always coerced:

```
score
```

```
## [1] 3 1 1 5
## Levels: 1 < 2 < 3 < 4 < 5
```

```
mean(score)
## argument is not numeric or logical: returning NA
```

Explicit conversion helps:

```
mean(as.integer(score))
```

```
## [1] 2.5
```

Missings

`NA` is a special value reserved for denoting missing values. They can occur in all types of vectors:

```
(age <- c(32, 68, NA, 55, 35, NA, 55, 56))
```

```
## [1] 32 68 NA 55 35 NA 55 56
```

```
(smoker <- c(TRUE, NA, NA, TRUE, FALSE, FALSE, FALSE, TRUE))
```

```
## [1]  TRUE    NA    NA  TRUE FALSE FALSE FALSE  TRUE
```

`is.na(...)` indicates which elements are missing:

```
is.na(age)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
```

We can combine this with `sum(...)` to find out, how many elements are missing in a vector:

```
sum(is.na(age))
```

```
## [1] 2
```

Missings

All operations that involve a calculation with a NA value result in NA:

```
a <- c(1, NA, 3)
b <- c(NA, 8, 9)
a + b
```

```
## [1] NA NA 12
```

```
sum(a)
```

```
## [1] NA
```

```
mean(b)
```

```
## [1] NA
```

We'll later learn how to ignore NAs for some calculations.

Vector length

A vector's size or length can be found out with `length`:

```
smoker
```

```
## [1]  TRUE    NA    NA  TRUE FALSE FALSE FALSE  TRUE
```

```
length(smoker)
```

```
## [1] 8
```

→ counts the number of elements in a vector **including** `NA` values

Data frames

Data frames are the most common data structure for storing **tabular data**. They consist of columns and rows. Think of the columns as variables and rows as observations.

Vector

- 1 column or row of data
- 1 type (numeric or text)

Matrix

- multiple columns and/or rows of data
- 1 type (numeric or text)

Data Frame

- multiple columns and/or rows of data
- multiple types

Each column is a vector with a specific type. All columns (vectors) have the same length.

Creating data frames

There are multiple ways to create a data frame:

1. By passing vectors

```
(cities <- data.frame(city = c('Berlin', 'Paris', 'Madrid'),
                      pop = c(3711930, 2206488, 3141991),
                      area = c(892, 105, 604)))
```

```
##     city     pop area
## 1 Berlin 3711930  892
## 2  Paris 2206488  105
## 3 Madrid 3141991  604
```

2. By loading data from a file

Note that there are many formats to load data from (Excel, CSV, Stata, etc.)

```
(cities <- read.csv('02rbasics1-data/cities.csv', stringsAsFactors = FALSE))
```

```
##     city     pop area
## 1 Berlin 3711930  892
## 2  Paris 2206488  105
## 3 Madrid 3141991  604
```

Creating data frames

3. By conversion from another data type

```
consent   # this is a factor
```

```
## [1] low      low      very low high
## Levels: very low < low < neutral < high < very high
```

```
as.data.frame(consent)
```

```
##    consent
## 1      low
## 2      low
## 3 very low
## 4     high
```

4. By using a builtin dataset

```
data("airquality")
head(airquality, 2)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
```

Data frame inspection

```
cities
```

```
##     city     pop area
## 1 Berlin 3711930  892
## 2  Paris 2206488  105
## 3 Madrid 3141991  604
```

Finding out the number of columns:

```
ncol(cities)
```

```
## [1] 3
```

Finding out the number of rows:

```
nrow(cities)
```

```
## [1] 3
```

Data frame inspection

Getting/setting the column names:

```
colnames(cities)
```

```
## [1] "city" "pop"  "area"
```

```
colnames(cities) <- c('city', 'population_cityarea', 'area_km2')
cities
```

```
##     city population_cityarea area_km2
## 1 Berlin             3711930      892
## 2  Paris             2206488      105
## 3 Madrid             3141991      604
```

Data frame inspection

Getting/setting the row names:

```
rownames(cities)
```

```
## [1] "1" "2" "3"
```

```
rownames(cities) <- c('BLN', 'PRS', 'MDRD')
cities
```

```
##        city population_cityarea area_km2
## BLN  Berlin             3711930      892
## PRS   Paris             2206488      105
## MDRD Madrid             3141991      604
```

Setting row names is usually not necessary as you can use a column as row identifier.

Accessing columns in data frames

```
cities
```

```
##        city population_cityarea area_km2
## BLN  Berlin             3711930      892
## PRS   Paris             2206488      105
## MDRD Madrid             3141991      604
```

Remember: **Each data frame column is a vector.** You can access a column and thereby obtain its data vector using the `$` sign:

```
cities$population_cityarea
```

```
## [1] 3711930 2206488 3141991
```

You can work with them as usual:

```
max(cities$population_cityarea)
```

```
## [1] 3711930
```

How would you calculate the mean of the area in the `cities` data frame?

Column names

- column name rules are more relaxed than object name rules
- e.g. you can use spaces but you have to use a special syntax for access then:

```
cities$`city name`    # assuming that I renamed column "city" to "city name"
## [1] "Berlin" "Paris"  "Madrid"
```

- **it's generally a good idea to avoid spaces and other special characters also in column names** - arkward column names can happen during data import → **you should rename the columns before further processing!**

Adding and removing columns

New columns can be added by assigning a vector to a new column name:

```
cities$pop_density <- cities$population_cityarea / cities$area_km2
cities
```

```
##        city population_cityarea area_km2 pop_density
## BLN  Berlin             3711930      892    4161.357
## PRS   Paris             2206488      105   21014.171
## MDRD Madrid             3141991      604    5201.972
```

There are many ways to drop a column, one is to assign the special value `NULL`:

```
cities$pop_density <- NULL
cities
```

```
##        city population_cityarea area_km2
## BLN  Berlin             3711930      892
## PRS   Paris             2206488      105
## MDRD Madrid             3141991      604
```

Accessing rows/observations

Accessing rows (i.e. observations) is called subsetting or filtering. Subsetting does not yield a vector but another data frame (a subset of the source data frame).

You will learn how to do this with a package called dplyr in one of the next sessions.

Tasks

Tasks

1. Install SWIRL courses that we'll need for the next sessions. You should already have installed the package swirl. If not, you need to do that first (see previous session's slides). Next, install the interactive R courses as described **on the next slide**.
2. Complete **lessons 1 and 4** of SWIRL Course **"R Programming"**. Lesson 3 is optional, lesson 2 should be skipped.
3. Take a look at the last 5 tweets of WZB_Berlin
    1. Create two vectors `retweets` and `likes` that contain the respective numbers from the last 5 tweets
    2. Create a third vector `tweet_ids` that contains the letters a to e as identifiers for the tweets
    3. Check the data type of all three vectors using the function `class(...)`
    4. Look at 5 more tweets, append the respective data to the vectors
    5. Create a dataframe `tweetstats` from the three vectors
    6. Add an additional variable/column to `tweetstats` named `interactions` which is the sum of retweets and likes for each observation

(Continued on next slide)

Tasks

4. As in the previous session's tasks, we'll work with the `cats` dataset from the package MASS.
    1. Load the package and the dataset.
    2. How do you bring up the dataset documentation / help for the dataset?
    3. Identify the number of rows and columns in the dataset by using the respective R functions.
    4. Identify the column names using the respective R function.
    5. What are the data types of the columns in the dataset? Again, use `class(...)` to answer this question.
    6. What if you recorded two more variables: Age and whether the cat has heart problems. Which data types would you choose for each variable?
    7. Create a new column `wt_ratio` which is the ratio of heart and body weight. Make sure to bring both variables to a common unit of measurement (i.e. both in grams or kilograms).

Notes on installing SWIRL courses

First, load the package:

```
library(swirl)
```

If an error message shows up, then you probably didn't install the package. You need to do this first with the command `install.packages('swirl')`.

Next, install three courses with the follwing commands (see next slide, in case one of this commands fails):

```
install_course_github("swirldev", "R Programming")
install_course_github("swirldev", "Getting and Cleaning Data")
install_course_github("swirldev", "Exploratory Data Analysis")
```

You can start SWIRL with the following command which will guide you through the first steps:

```
swirl()
```

One additional note: When you receive the question "Would you like to receive credit for completing this course on Coursera.org?", select "No".

Notes on installing SWIRL courses

In case installing a SWIRL course fails, do the follwing:

1. Download the ZIP file `swirl_courses-master.zip` from the following address: https://github.com/swirldev/swirl_courses/archive/master.zip
2. Move this file to your current working directory (you can find that out via `getwd()`).
3. Execute the following commands:

```
install_course_zip("swirl_courses-master.zip", multi=TRUE,
                   which_course = "R Programming")
install_course_zip("swirl_courses-master.zip", multi=TRUE,
                   which_course = "Getting and Cleaning Data")
install_course_zip("swirl_courses-master.zip", multi=TRUE,
                   which_course = "Exploratory Data Analysis")
```