⟨ S t a t W t h 1 7 ⟩

Introduction to R and RStudio

Werner Gurker

Institut für Stochastik und Wirtschaftsmathematik

Technische Universität Wien

References

[1] E. L. Cano, J. M. Moguerza, and M. P. Corcoba: Quality Control with R – An ISO Standards Approach, Springer, 2015

[2] G. Grolemund: Hands-On Programming with R, O'Reilly, 2014

[3] N. J. Horton and K. Kleinman: Using R and RStudio for Data Management, Statistical Analysis, and Graphics, 2nd Ed., CRC Press, 2015

[4] J. Verzani: Using R for Introductory Statistics, 2nd Ed., CRC, 2014

[5] H. Wickham and G. Grolemund: R for Data Science – Import, Tidy, Transform, Visualize, and Model Data, O'Reilly, 2017

System: Windows 7 Pro (64bit)

R: Version R-3.4.1 (64bit)

RStudio: Version 1.0.153 (64bit)

Software for Statistics

There exist a wide range of software packages for **Statistics** in general.

Most of the available software packages are proprietary and commercial.

There are more and more **Free and Open Source Software** (FOSS) options for any purpose, in particular **R**.

Meanwhile, the **R** software and programming language is widely spread; it has become the **de facto standard for data analysis** (universities, companies (small and large), . . .).

More and more job positions include **R** skills as a requirement.

What is R ? (1)

R is an open-source statistical environment modeled after S / S–Plus (www.insightful.com) created at the Bell Labs in the 1970s by John Chambers, Rick Becker, and Allan Wilks.

The R Project was started by Robert Gentleman and Ross Ihaka of the Statistics Department of the University of Aukland in 1995, and quickly gained a widespread audience.

The R Project for Statistical Computing (cran.r-project.org) is maintained by the R Development–Core Team, and supported by a number of people, institutions, and organizations from all over the world. (See The R Foundation for further details.)

In addition, R can be considered a community. Users organize themselves in R User's Groups (RUGs). An updated list can be found on the blog of Revolution Analytics (www.revolutionanalytics.com).

What is R ? (2)

R is platform-independent, it is available for Linux, Mac, and Windows. A general definition of R is as follows:

> **R** is a system for statistical computation and graphics. It consists of a language plus a runtime environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.

It is a **system** for statistical computation and graphics, i. e., it is more than just a statistical package.

It is also a **programming language**, i. e., it can be extended with new functionality. Advanced programming features as debugging or system interaction are available.

The **run-time environment** allows to use R in an interactive manner.

Writing and running **script files** is the natural way to use R.

Some Features of R

- It is Free and Open Source.
- It runs in almost any system and configuration.
- There is a base functionality for a wide range of statistical computations and graphics: descriptive statistics, statistical inference, time series, data mining, multivariate plotting, advanced graphics, optimization, etc.
- The base installation can be extended by installing contributed packages (at the time of writing, more than 11000 !) devoted to special topics.
- It has Reproducible Research and Literate Programming capabilities.
- Interfacing with other languages such as Python, C, or Fortran is possible as well as wrapping other programs within scripts.
- There is a wide range of options to get help and support: extensive documentation, communities, etc.

How to Obtain R ?

The **R Project** web side (`www.r-project.org`) is the main source of information to start with **R**.

Apart from two sections in the central part (**Getting Started**, **News**) you find the following sections on the left hand side menu:

Download (with a link to the CRAN repository), **R Project**, **R Foundation**, **Help With R**, **Documentation** (FAQs, Manuals, etc.); **Links** (links to related projects such as Bioconductor)

The **CRAN** web page has links to download and install the software for Linux, Mac, and Windows.

Another resource at CRAN are the **Task Views**. These are collections of resources related to special topics that bring together functions, packages, links, etc., classified and commented by the maintainer of the Task View.

CRAN Tasks Views (1)

Name	Topic
Bayesian	Bayesian Inference
ChemPhys	Chemometrics, Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, Analysis
Cluster	Cluster Analysis, Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Environmetrics	Analysis of Ecological, Environmental Data
ExperimentalDesign	Design of Experiments, Analysis of Experimental Data
ExtremeValue	Extreme Value Analysis
Finance	Empirical Finance
FunctionalData	Functional Data Analysis
Genetics	Statistical Genetics
Graphics	Graphic Displays, Dynamic Graphics, Graphic Devices
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning, Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis

CRAN Tasks Views (2)

Name	Topic
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	Official Statistics, Survey Methodology
Optimization	Optimization, Mathematical Programming
Pharmacokinetics	Analysis of Pharmacokinetic Data
Phylogenetics	Phylogenetics, Especially Comparative Methods
Psychometrics	Psychometric Models and Methods
ReproducibleResearch	Reproducible Research
Robust	Robust Statistical Methods
SocialSciences	Statistics for the Social Sciences
Spatial	Analysis of Spatial Data
SpatioTemporal	Handling and Analyzing Spatio-Temporal Data
Survival	Survival Analysis
TimeSeries	Time Series Analysis
WebTechnologies	Web Technologies and Services
gR	gRaphical Models in R

R Interfaces

The base installation of **R** comes with a **CLI** (Command Line Interface) which allows interacting with **R** by means of expressions (i. e., commands) and scripts. Also included in the base installation of **R** is a (simple) **GUI** (Graphical User Interface). There are a number of projects with regard to more refined **R** interfaces. We can find two types of interfaces:

- **MDB GUIs:** Interfaces with Menus and Dialog Boxes to perform (usually only limited) statistical analysis. The most popular GUIs of this kind are **R Commander** (package: `Rcmdr`) and **Deducer** (package: `Deducer`).
- **IDE:** Interfaces with an Integrated Development Environment which allow (the experienced user) to exploit all the capabilities of **R**. The most popular environments include **RStudio**, **Emacs + ESS**, and **Eclipse + StatET**.

R Expressions

The way to interact with R is through R expressions (or R commands). R is interactive in that it responds to inputs which can be expressions of several types:

> arithmetic expressions; logical expressions; functions; assignments

Although R works with in-memory data, there is obviously a need to work with expressions containing files in several ways:

> read data files (of various formats); write data (in various formats); save plots (in various formats); create script files (containing, for example, a complete statistical analysis); create reports (including text, code, results, data, graphics)

R Infrastructure

The R infrastructure comprises the following elements:

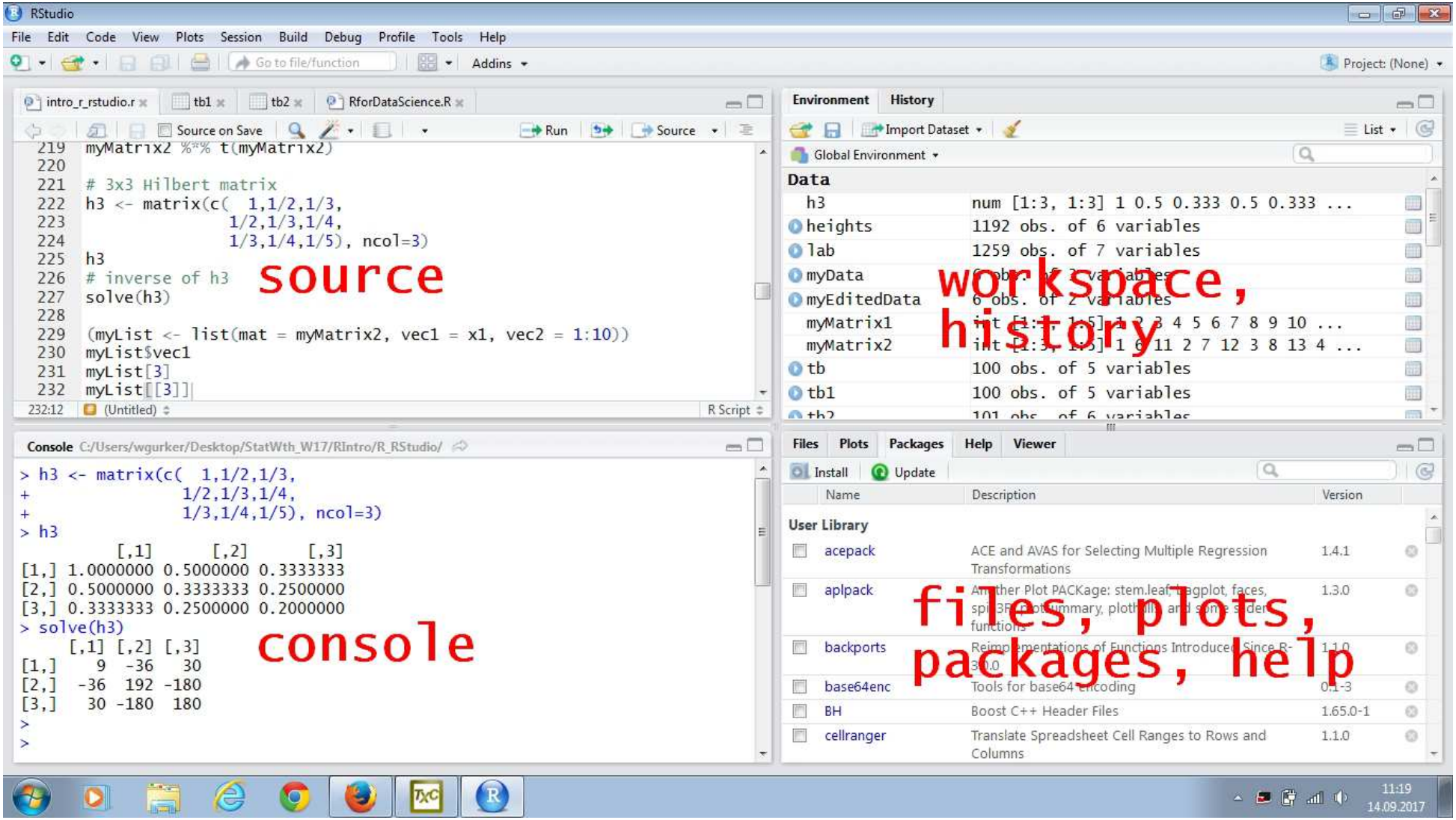
console; editor; graphical output; history; workspace; working directory

The first three of these elements can be accessed using the base **R GUI**; the other elements are hidden (and can be accessed via R commands).

In the following we will use **RStudio** which allows to access all of these elements (and many more) in a simple and intuitive way.

RStudio is a Java-based application; the basic desktop version can be downloaded for free for various platforms at `www.rstudio.com`. (First install R and then **RStudio**; in both cases accept the default options.)

RStudio: Default Layout (1)



RStudio: Default Layout (2)

[LL] **R Console:** prompt, output from expressions (or system related)

[UL] **R Source:** scripts, text files, data files, etc.

[UR]
- **R Environment:** workspace (lists available objects)
- **R History:** commands history (can be searched and used again)

[LR]
- **Files:** file explorer (can be linked to the working directory)
- **Plots:** graphics device (the plots generated are shown here)
- **Packages:** packages available in the system (there packages are installed, uninstalled, or updated)
- **Help:** R documentation (including installed contributed packages)
- **Viewer:** for web applications

The Command Line (1)

Note: Rather than showing screenshots of the **R Console**, we typeset the command line and show the output.

```
> 2 + 2
[1] 4
```

Average of five numbers:

```
> (1 + 3 + 2 + 12 + 8)/5
[1] 5.2
```

R uses standard conventions for mathematical operations. Here we find the distance between two points $(1, 3)$ and $(2, 1)$:

```
> ( (2 - 1)^2 + (1 - 3)^2 )^(1/2)
[1] 2.236068
```

The Command Line (2)

Multiple Commands: We can place more than one command on the command line at once. Separate them by a semicolon (;).

```
> 2^3; 1 + 2 + 3
[1] 8
[1] 6
```

The Prompt: The command line has two states, one being ready for input (>), the other expecting a continuation of the current input (+). Complete the command line or leave it by pushing the ESC-button.

```
> (1 + 3 + 2 + 12 +
+
+ 8)/5
[1] 5.2
```

The Command Line (3)

Errors: Sometimes we type in a command that does not make sense to R's interpreter. Most of the time the error messages are quite informative (for the newbie some may appear a little bit cryptic). Sometimes R gives only a warning (but does not stop).

```
> 2^^3
Error: unexpected '^' in "2^^"

> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

Command History: After one issues a command, it is recorded in R's history (see also the UR-pane of RStudio). We may scroll through the previous commands (by using the arrow keys), edit and re-execute them.

The Command Line (4)

Variables: The power of R goes well beyond that of a calculator. In particular, names can be to assigned to values with an **assignment operator** (<- is preferred, but = can also be used).

```
> x <- 2
> y <- x^2 - 2*x + 1
> y
[1] 1
```

Variable names may be long or short, we may use letters and numbers (start with a letter), and use . or _ . Note that case is important. All the following assignments (and many more) are valid (and different):

```
> myData <- 5
> Data_ <- 5
> my.Data <- 5
> my_Data <- data. <- 5
```

The Command Line (5)

Tab completion: When a command is partially entered and the tab key is pressed, a list of possible completions is shown. Use the mouse (or the arrow/enter keys) to choose from this list.

Built-in variables: R has a few built-in variables, for example `pi` or `T/F` (for the logical `TRUE/FALSE`, respectively). These names may have new values bound to them (for example, `pi <- 1000`), but it is not recommended to do so.

Functions: The R language is comprised of numerous built-in functions, several are for the familiar mathematical operations. For example:

```
> x <- pi
> sin(x)
[1] 1.224606e-16
> sqrt(x)
[1] 1.772454
```

The Command Line (6)

Functions are called by their name followed by a pair of parentheses. If there is more than one argument (which is usually the case), these are separated by commas. For example:

```
> log(x)          # base e = exp(1)
[1] 1.14473
> log(x, 10)      # base 10
[1] 0.4971499
```

Combine values: One of the most commonly used functions has the short name c. This function combines values together (giving a vector; see below). For example:

```
> x <- c(74, 122, 235, 111, 292)
> mean(x)         # take the average (or mean) value
[1] 166.8
```

The Command Line (7)

Vectorized functions: R has several functions which perform their task not only for a single number but rather do the same thing for a vector of numbers. The standard mathematical functions are of this kind:

```
> x + x
[1] 148 244 470 222 584
> sqrt(x)
[1]  8.602325 11.045361 15.329710 10.535654 17.088007
> x - mean(x)
[1] -92.8 -44.8  68.2 -55.8 125.2
```

In this last example the sizes of `x` and `mean(x)` did not match. First, R **recycles** values from the smaller object to create new matching objects, and then performs the vectorized subtraction. This is an important feature of R (which may be confusing sometimes).

The Command Line (8)

Default/Named arguments: To make it easier to use functions with many arguments, some of them have default values. For example, the `mean` function has two additional arguments, `trim` (default: `trim = 0`), and `na.rm` (default: `na.rm = FALSE`). Suppose we have a new data object, where one value is missing:

```
> x1 <- c(74, 122, NA, 235, 111, 292)
> mean(x1)
[1] NA
> mean(x1, na.rm = TRUE)  # first remove missing values
[1] 166.8
> mean(x1, trim = 0.5, na.rm = TRUE)
[1] 122
```

In this example we used two named arguments. Additional arguments can be matched by position or by keyword. In the last command, either could have been used. (For ease of use, the second method is preferred.)

The Command Line (9)

Generic functions: The same function name may refer to very different function definitions, depending on the type (more precisely, on the **class**; see below) of its first argument. Functions of this kind are called **generic**. We illustrate with the `summary` function:

```
> x1 <- c(74, 122, NA, 235, 111, 292)
> y <- c(FALSE, FALSE, TRUE, TRUE)
> summary(x1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   74.0   111.0   122.0   166.8   235.0   292.0       1
> summary(y)
   Mode   FALSE    TRUE
logical       2       2
```

Depending on the argument (`x` is a vector of numbers, `y` a vector of logical values) the function behaves quite different. In the first case it provides us with some numerical summaries, in the second case it is merely a count.

The Command Line (10)

Help system: The base installation of R comprises a fairly small set of base functionality and can be extended by numerous additional packages. For the most part, the data sets, functions or packages (base or add-ons) are accompanied by (more or less elaborated) documentations. R's help system allows one to access these help pages in many different ways.

The most basic access is via ?, for example, `?mean`. Note that the `mean` function is generic and the Usage section (of the R documentation) shows what is available by default. (See the accompanying R-file for some more examples.)

Using RStudio, a simple way of getting help is to put the mouse pointer on the function name and pressing the F1 button.

Another way of getting help is using a web browser, in this example, by typing (something similar to) `R: mean` in the search field of the browser.

The Workspace (1)

Working with R one typically creates many objects and functions. R will maintain these objects in a global **Workspace**. When R searches for an object at the command line, this is the first place on its path that it will look for.

RStudio lists most items in the global workspace along with a short summary in the **Workspace pane**. Alternatively, the `ls` function my be used at the command line to list the objects and functions the user has defined. To get a short summary of an object the `summary` or the `str` function may be used:

```
> ls()
[1] "x"  "x1" "y"
> str(x)
 num [1:5] 74 122 235 111 292 2
```

The Workspace (2)

Sometimes we may wish to remove some objects from the workspace. This can be done through the `rm` function.

```
> rm(x)              # remove a single object (x)
> rm(list=ls())      # remove all objects (first check ls()!)
```

Alternatively you may use the 'broom' icon on the Workspace pane.

Sessions: The global workspace and history file contain the currently defined objects and the commands that were used to create them. Both may be useful to keep and R can do so from session to session. When one quits R, a prompt to 'Save workspace image' is given. Depending on your settings, R will write the contents of the workspace to a file (`.RData`) to be read back when R is started again. (Some authors recommend not to do so and always start with an empty session. They argue that this forces the R user to concentrate on code and scripts and not on objects in the global environment.)

RStudio Projects (1)

We may keep all the files associated with a particular project (data files, R scripts, analytical results, figures, etc.) in an **RStudio project**. This procedure may be useful, for example, when we run several (different) projects in parallel, but we do not want to mix them up.

Although we will not use this framework in the lecture, let's consider a small **Example**: Click File –> New Project, create an Empty Project, call it `RIntro`, and choose a place where you want to put it in (in this example, the Desktop). Once this process is complete, you'll get a new RStudio project. Check that the 'home' directory of the project is the current directory:

```
> getwd()
[1] "C:/Users/wgurker/Desktop/RIntro"
```

RStudio Projects (2)

Open a new R script, enter the following commands, and save the file, calling it, for example, `mpg.R`. (Don't worry about the details, we will discuss them below.)

```
mpg <- ggplot2::mpg
plot(hwy ~ displ,
     type = "p",
     pch = 19,
     cex = 1.3,
     col = "red",
     data = mpg)
```

Run the script, and save the plot, calling it `mpg.pdf` (use Export –> Save as PDF). Quit RStudio and check the folder associated with your project. Double click the `.Rproj` file, and notice that you get where you left off.

External Packages (1)

Base R can be extended through numerous external packages. Check how many are available at the moment:

```
> Sys.Date()
[1] "2017-09-14"
> nrow(available.packages())
[1] 11377
```

Packages are primarily available through **CRAN**, the worldwide repository of packages and related sources. (There is a mirror at the WU–Wien; this is the main CRAN repository)

RStudio provides a pane for interacting with packages. From here one can load or unload currently installed packages, by selecting or deselecting the appropriate box. Once loaded, the functions and data sets of this package are available for use.

External Packages (2)

To install packages not listed in the pane, the following information pieces are required:

- The package **name**. (The autocompletion function is helpful in case you don't know exactly the name of the package.)
- The **repository** where to install from. (Use a repository next to you.)
- The **library** to install the package into. (Usually this can be left to the default.)

Note that packages usually have (possibly many!) **dependencies** on other packages. (The default settings are to automatically install any dependent packages.)

Like R, packages are **versioned**. (Using the Update button from time to time may be wise; but be aware that some of your commands may cause an error in updated package versions.)

External Packages (3)

The following functions perform the core functionality to install and load add-on packages from the command line. For example, let's install, load, and unload the qcc package:

```
# install the package on the system
> install.packages("qcc")
# attach (load) the package to the workspace
> library("qcc")
# alternatively, we can use
> require("qcc")
# detach (unload) the package from the search path
> detach("package:qcc", unload=TRUE)
```

Note: Each time R is closed and restarted the required packages have to be reloaded. This may be done using the check boxes in the Package pane. Alternatively, it may be more convenient to load the packages in the R scripts as they are needed in the code.

Data Sets (1)

Many packages include accompanying data sets. In addition, base R has a `datasets` package which is loaded automatically at start.

Usually the data sets in a package are available in the user's search path, though they don't appear in the Workspace pane by default. For example, let's look at the first 50 values of the `rivers` data set (in the `datasets` package):

```
> head(rivers, n = 50)
 [1]  735  320  325  392  524  450 1459  135  465  600
[11]  330  336  280  315  870  906  202  329  290 1000
[21]  600  505 1450  840 1243  890  350  407  286  280
[31]  525  720  390  250  327  230  265  850  210  630
[41]  260  230  360  730  600  306  390  420  291  710
```

Data Sets (2)

The data function: The `rivers` object of the previous example cannot be edited directly, any edits will produce a copy in the user's workspace (shown in the Workspace pane). A copy will also be made if one brings the data set into the workspace using the `data` function. Here are some examples:

```
# not shown in Workspace pane (RStudio)
> rivers   # [output not shown]
> rivers[1:5]
[1] 735 320 325 392 524
# editing produces a copy in workspace
> rivers[1] <- NA
# remove edited data set
> rm(rivers)
# load a copy into workspace
> data(rivers)
```

Data Sets (3)

The `data` function may also be used to search a package for available data sets; for example, `data(package = "UsingR")` (look what's shown in the upper left pane). Next, without actually loading the `UsingR` package, let's load the `alltime.movies` data set in this package into the workspace:

```
> data(alltime.movies, package = "UsingR")
> head(alltime.movies)
                                  Gross Release.Year
Titanic                             601         1997
Star Wars                           461         1977
E.T.                                435         1982
Star Wars: The Phantom Menace       431         1999
Spider-Man                          404         2002
Jurassic Park                       357         1993
```

Note: The data set is stored as what is called a **data frame**. More will be said about this kind of R objects below.

Data Classes and Types (1)

Data objects in R can be thought of in two different senses, with regard to their **class** and their **type** (or **mode**). Some basic **data types** are:

- **logical:** TRUE/FALSE
- **integer:** integer number
- **double:** real number
- **character:** string character

There are other data types (for example, complex); use the documentation of the `typeof` function to learn more about them.

```
> c(typeof(pi), mode(pi))
[1] "double"  "numeric"
> c(typeof("today"), mode("today"))
[1] "character" "character"
> c(typeof(rivers), mode(rivers))
[1] "double"  "numeric"
```

Data Classes and Types (2)

On the other hand, these basic data can be organized in various data structures. The most important classes available in R are:

- **vector:** one dimensional entity of ordered values (of the same type)
- **matrix:** vector organized in rows and columns (more generally, as a multi-dimensional **array**)
- **list:** list of objects that can be of different types and lengths; results of statistical computations are usually returned as lists
- **data frame:** data set organized in columns (of the same length but possibly of different types) and rows
- **factor:** useful to handle categorical data (also contains information about levels and labels)
- **function:** functions are themselves objects in R which can be stored in the project's workspace

Data Classes and Types (3)

There are many more classes in R (and new classes can be created by the programming capabilities of R). For example, objects of class 'ts' are useful for working with time series.

The class of an object is used to allow for an **object-oriented** style of programming in R. For example, if an object has class 'data.frame' it will be printed and plotted in a certain way, etc.

```
> class(rivers)
[1] "numeric"
> summary(rivers)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  135.0   310.0   425.0   591.2   680.0  3710.0

> class(mean)
[1] "function"
```

Data Classes and Types (4)

```
> class(alltime.movies)
[1] "data.frame"
> summary(alltime.movies)
     Gross          Release.Year
 Min.   :172.0   Min.   :1937
 1st Qu.:184.0   1st Qu.:1990
 Median :216.0   Median :1997
 Mean   :240.2   Mean   :1993
 3rd Qu.:260.0   3rd Qu.:2001
 Max.   :601.0   Max.   :2003

> (myFactor1 <- factor(rep(1:5, 2), labels = letters[1:5]))
 [1] a b c d e a b c d e
Levels: a b c d e
> class(myFactor1)
[1] "factor"
```

Vectors (1)

The most basic classes in R are vectors. In particular, more complex data structures are composed of vectors (for example, the columns of a data frame are vectors).

Creating vectors: There are several ways of creating vectors. One uses interactively the `scan` function:

```
> x1 <- scan()
1: 10
2: 20
3: 30
4:
Read 3 items
```

Now we have created a vector whose name is `x1`. Using RStudio, the object is shown under the Environment tab (UR pane). Additionally, we get some information about this object. (Alternatively, use `str(x1)`.)

Vectors (2)

Clearly, creating vectors interactively is not always practical. Here are some other possibilities:

```
> x1 <- c(10, 20, 30, 40, 50); x1
[1] 10 20 30 40 50
> x1 <- seq(from = 10, to = 50, by = 10); x1
[1] 10 20 30 40 50
> x2 <- 1:10; x2
 [1]  1  2  3  4  5  6  7  8  9 10
> x3 <- c(rep("pinetree", 3), rep("oaktree", 2)); x3
[1] "pinetree" "pinetree" "pinetree" "oaktree"  "oaktree"
> x4 <- c(seq(from = 0, to = 1, by = 0.2), 5:9); x4
 [1] 0.0 0.2 0.4 0.6 0.8 1.0 5.0 6.0 7.0 8.0 9.0
> x5 <- seq_along(x4); x5
 [1]  1  2  3  4  5  6  7  8  9 10 11
```

Check that all these new vectors are in the workspace.

Vectors (3)

In the examples above we used the `rep` function. This function can be used to repeat some values of a vector a specified number of times:

```
> rep(c(1,2,3), times = c(3,2,1))
[1] 1 1 1 2 2 3
```

Logical vectors can also be created:

```
> logicalVector <- 1:6 > 3
> logicalVector
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
> sum(logicalVector)
[1] 3
```

Note that `TRUE` and `FALSE` are treated as 1 and 0, respectively, when operating with them. This is useful, for example, to get the number of elements that are true in a logical vector.

Vectors (4)

Length and names: To find the length (= number of elements) of a vector, we can use the `length` function:

```
> length(x1)
[1] 5
```

To label the elements of a vector we can use the `names` function. In the following example we also use the `paste` function which is useful if some characters are common to all labels:

```
> names(x1) <- paste("week", 1:5, sep=""); x1
week1 week2 week3 week4 week5
   10    20    30    40    50
> names(x1)
[1] "week1" "week2" "week3" "week4" "week5"
```

Vectors (5)

Below we will use the `precip` dataset (package: `datasets`). This is a named vector of length 70 (use `?precip` for further details):

```
> precip
      Mobile       Juneau      Phoenix
        67.0         54.7          7.0
 Little Rock  Los Angeles  Sacramento
        48.5         14.0         17.2
     . . . . .
> length(precip)
[1] 70
> names(precip)
 [1] "Mobile"      "Juneau"
 [3] "Phoenix"     "Little Rock"
 [5] "Los Angeles" "Sacramento"
     . . . . .
```

Vectors (6)

Indexing: Data object in R are indexed, and we can access each element of a vector (or any other data object; see below) through its index or its name (if it exists). Here are some examples (first guess what you will get):

```
x1[3]
x1[3] <- 50; x1
x1[c(1,3)]
x1[c(-2)]
x1["week3"]
# extend a vector
x1 <- c(x1, 60)
x1[10] <- 100; x1
# reduce a vector
x1 <- x1[-c(6:10)]; x1
# use a logical expression
x1[x1 > 20]
```

Vectors (7)

Additional examples using the `precip` dataset:

```
> precip[c("Seattle Tacoma", "New York")]
Seattle Tacoma       New York
          38.8           40.2

# find all cities with an average precipitation of
# more than 50 inches
> precip[precip > 50]
     Mobile       Juneau Jacksonville        Miami
       67.0         54.7         54.5         59.8
 New Orleans     San Juan
       56.8         59.2
```

Vectors (8)

Suppose monthly sales (in 10,000s) of CDs in a certain year were: 79, 74, 161, 127, 133, 210, 99, 143, 249, 249, 368, 302. We enter the data as a named vector (the `scan` function may be useful in this case), and form two vectors containing the months with 31 days, and the remaining ones.

```
> cd <- c(79,74,161,127,133,210,99,143,249,249,368,302)
> names(cd) <- month.abb; cd
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
 79  74 161 127 133 210  99 143 249 249 368 302
> m31 <- c(1,3,5,7,8,10,12)
> cd31 <- cd[m31]; cd31
Jan Mar May Jul Aug Oct Dec
 79 161 133  99 143 249 302
> cd30 <- cd[-m31]; cd30
Feb Apr Jun Sep Nov
 74 127 210 249 368
```

Vectors (9)

Ordering: Two functions are related to the ordering of vectors, `sort` and `order`. First we use the `sample` function to create a random vector.

```
> set.seed(1234)
> x6 <- sample(1:1000, 10, replace = TRUE); x6
 [1] 114 623 610 624 861 641  10 233 667 515
> sort(x6)
 [1]  10 114 233 515 610 623 624 641 667 861
> sort(x6, decreasing = TRUE)
 [1] 861 667 641 624 623 610 515 233 114  10
> ii <- order(x6); ii
 [1]  7  1  8 10  3  2  4  6  9  5
> x6[ii]
 [1]  10 114 233 515 610 623 624 641 667 861
```

Vectors (10)

Operations: There are two types of operations over a vector:

- Operations over all elements of a vector as a whole (can be a computation, a plot, etc.)
- Operations over each element of a vector (resulting in a vector of the same length)

```
> mean(x6)  # first type
[1] 489.8
> x6 + 100  # second type (using recycling)
 [1] 214 723 710 724 961 741 110 333 767 615
> sqrt(x6)  # second type
 [1] 10.677078 24.959968 24.698178 24.979992 29.342802
 [6] 25.317978  3.162278 15.264338 25.826343 22.693611
```

Matrices (1)

A rectangular collection of values of the same type can be stored in a matrix. (Hence, a matrix is a vector organized in rows and columns.) There are several ways to **create** a matrix. The most common way is through the `matrix` function:

```
> myMatrix1 <- matrix(c(1:15), nrow=3, ncol=5); myMatrix1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
> myMatrix2 <- matrix(c(1:15), nrow=3, byrow=T); myMatrix2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
```

Matrices (2)

We can **extract** (and replace) parts of the matrix in the same way as we did in vectors. Here are some examples:

```
> myMatrix1[3, 2]
[1] 6
> myMatrix1[1, ]
[1]  1  4  7 10 13
> myMatrix1[2:3, ]
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    5    8   11   14
[2,]    3    6    9   12   15
> myMatrix1[, 5] <- myMatrix2[, 5]; myMatrix1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10    5
[2,]    2    5    8   11   10
[3,]    3    6    9   12   15
```

Matrices (3)

We can assign **names** to rows and columns:

```
> colnames(myMatrix2) <- paste("var", 1:5, sep="")
> rownames(myMatrix2) <- paste("case", 1:3, sep="")
> myMatrix2
      var1 var2 var3 var4 var5
case1    1    2    3    4    5
case2    6    7    8    9   10
case3   11   12   13   14   15
```

Now elements, rows or columns can be **accessed** using names:

```
> myMatrix2["case3", ]
var1 var2 var3 var4 var5
  11   12   13   14   15
```

Matrices (4)

Many **operations** and **functions** can be applied to matrices. Here is a small selection of examples:

```
> rowSums(myMatrix2)  # row sums
[1] 15 40 65
> rowMeans(myMatrix2) # row means
[1]  3  8 13
> colMeans(myMatrix2) # column means
[1]  6  7  8  9 10
> t(myMatrix2)        # transposition
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```

Matrices (5)

```
> myMatrix2 * myMatrix2          # element-wise multiplication
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    9   16   25
[2,]   36   49   64   81  100
[3,]  121  144  169  196  225
> myMatrix2 %*% t(myMatrix2)    # true matrix multiplication
     [,1] [,2] [,3]
[1,]   55  130  205
[2,]  130  330  530
[3,]  205  530  855
> solve(h3)                      # inverse of 3x3 Hilbert
     [,1] [,2] [,3]              # matrix (see R script)
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
```

Lists (1)

Lists are data structures that can contain any other R objects of different types and lengths. The elements of a list can be **named**.

```
> (myList <- list(mat = myMatrix2, vec1 = x1, vec2 = 1:10))
$mat
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15

$vec1
week1 week2 week3 week4 week5
   10    20    30    40    50

$vec2
 [1]  1  2  3  4  5  6  7  8  9 10
```

Lists (2)

Similar to vectors, the components of a list can be **accessed** by index or by name. In the latter case the $-$ operator is used.

```
> myList$vec1
week1 week2 week3 week4 week5
   10    20    30    40    50
> myList[3]
$vec2
 [1]  1  2  3  4  5  6  7  8  9 10

> myList[[3]]
 [1]  1  2  3  4  5  6  7  8  9 10
```

The difference between `[3]` and `[[3]]` is that when using double brackets, we get the original object within the list (a vector in this case), whereas using single brackets, we get an object of class list. Let's consider some more examples.

Lists (3)

```
> myList[[3]][1]                    # extract vec2[1]
[1] 1
> myList[c(1, 2)]                   # extract more than one
$mat                                # component of myList
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15

$vec1
week1 week2 week3 week4 week5
   10    20    30    40    50

> myList$mat[ , 2]                  # extract elements of
[1]  2  7 12                        # inner components
```

Data Frames (1)

The usual way of working with data is to organize them in a rectangular scheme of rows and columns, where columns represent **variables** which are measured or observed for a set of **items**, represented by rows.

Objects of this kind are called **data frames** and are of class 'data.frame'. Note that in contrast to matrices (which share a similar structure) where all elements have to be of the **same** type, the columns of data frames can be of **different** types.

Usually, data are imported to data frames from files of various kinds, such as text files, spreadsheets, databases, etc. (In R there are various functions to perform this.) Sometimes, data frames are created from other R objects or 'from scratch' using the `data.frame` function. Let's consider a small example.

Data Frames (2)

```
> myData <- data.frame(
+   type = c("A", "A", "B", "C", "C", "C"),
+   weight = c(10.1, 20.3, 15.2, 13.4, 23.2, 8.1))
> myData
  type weight
1    A   10.1
2    A   20.3
3    B   15.2
4    C   13.4
5    C   23.2
6    C    8.1
> str(myData)
'data.frame': 6 obs. of  2 variables:
 $ type  : Factor w/ 3 levels "A","B","C": 1 1 2 3 3 3
 $ weight: num  10.1 20.3 15.2 13.4 23.2 8.1
```

Data Frames (3)

As can be seen from the previous slide, using `str(myData)` shows how to access the different parts of the data frame. Note that accessing by `$name` is equivalent to using `[[]]`. For example:

```
> myData$type     # results in a vector
[1] A A B C C C
Levels: A B C
> myData[2]       # results in a data frame
  weight
1   10.1
2   20.3
3   15.2
4   13.4
5   23.2
6    8.1
```

Data Frames (4)

Some more examples:

```
> myData[3, ]
  type weight
3    B   15.2
> myData[myData$weight < 15, ]
  type weight
1    A   10.1
4    C   13.4
6    C    8.1
```

Note that columns and rows of a data frame always have **names**, even in case we don't use names when creating a data frame. (The default is that columns are named as V1, V2, etc., and rows are named as 1, 2, etc.) Row and column names may be assigned or changed in the same way as we did for matrices.

Data Frames (5)

Let's consider an example (first we make a copy of our data frame):

```
> myEditedData <- myData
> colnames(myEditedData)[2] <- "itemWeight"
> rownames(myEditedData) <- paste("case",
+                                 rownames(myEditedData),
+                                 sep = "_")
> myEditedData
       type itemWeight
case_1    A       10.1
case_2    A       20.3
case_3    B       15.2
case_4    C       13.4
case_5    C       23.2
case_6    C        8.1
```

Data Frames (6)

Data frames can be **ordered**, **filtered** (i. e., subsetted), **aggregated** (i. e., building subtotals of numerical by categorical variables), and **edited** (i. e., changing values, adding new columns, etc.). Using our illustrative data frame `myData`, we consider some examples.

```
> myData[order(myData$weight), ]
  type weight
6    C    8.1
1    A   10.1
4    C   13.4
3    B   15.2
2    A   20.3
5    C   23.2
```

Data Frames (7)

```
> subset(myData, weight > 15)
  type weight
2    A   20.3
3    B   15.2
5    C   23.2

> aggregate(weight ~ type, data = myData, sum)
  type weight
1    A   30.4
2    B   15.2
3    C   44.7
```

In the last example we used a special type of expression, a **formula**. A formula is an expression with two sides, separated by the symbol ~. It is mainly used to specify **models** in the form of `y ~ model`. In the example the 'model' is formed by the criteria by which we want to sum the data.

Data Frames (8)

```
# add a new column
> myData$randomorder <- sample(1:6)
> myData
  type weight randomorder
1    A   10.1           2
2    A   20.3           6
3    B   15.2           3
4    C   13.4           4
5    C   23.2           1
6    C    8.1           5

# remove the new column
> myData$randomorder <- NULL
```

Data Frames (9)

Above we added (and removed) a new column. Suppose we want to add a **computed** column. (Note that the procedure is similar to what we do in spreadsheets with formulas.)

```
> myData$proportion <- myData$weight/sum(myData$weight)
> myData
  type weight proportion
1    A   10.1  0.1118494
2    A   20.3  0.2248062
3    B   15.2  0.1683278
4    C   13.4  0.1483942
5    C   23.2  0.2569214
6    C    8.1  0.0897010
```

Note that these few examples only give a first impression of what can be done with data frames.

Tibbles (1)

In more recent texts on data analysis with R you may find that so-called 'tibbles' are used instead of data frames. In fact, tibbles *are* data frames, but they tweak some of the unfavorable features of the older concept. At first sight, the main difference of tibbles versus data frames is how they are printed. Let's consider two examples:

```
> library("tibble")
> as.tibble(myData)
# A tibble: 6 x 2
    type weight
  <fctr>  <dbl>   # <<-- types of variables are shown here
1      A   10.1
2      A   20.3
3      B   15.2
4      C   13.4
5      C   23.2
6      C    8.1
```

Tibbles (2)

```
> ggplot2::mpg
# A tibble: 234 x 11
   manufacturer       model displ  year   cyl       trans   drv
          <chr>       <chr> <dbl> <int> <int>       <chr> <chr>
 1         audi          a4   1.8  1999     4   auto(l5)     f
 2         audi          a4   1.8  1999     4 manual(m5)     f
 3         audi          a4   2.0  2008     4 manual(m6)     f
 4         audi          a4   2.0  2008     4   auto(av)     f
 5         audi          a4   2.8  1999     6   auto(l5)     f
 6         audi          a4   2.8  1999     6 manual(m5)     f
 7         audi          a4   3.1  2008     6   auto(av)     f
 8         audi a4 quattro   1.8  1999     4 manual(m5)     4
 9         audi a4 quattro   1.8  1999     4   auto(l5)     4
10         audi a4 quattro   2.0  2008     4 manual(m6)     4
# ... with 224 more rows, and 4 more variables: cty <int>,
#   hwy <int>, fl <chr>, class <chr>
```

Data Import (1)

In this section we briefly discuss how to import **external** data sets, that is, data sets that are not part of R packages or have been created by the user. Of course, we are faced with a wide range of possibilities. (Check the "R Data Import/Export" manual for a more full discussion.)

Spreadsheet data: Perhaps the most common source of data are spreadsheets, such as Excel. R has some add-on packages for interacting directly with Excel (for example, the `xlsx` package).

For a simple data exchange from a spreadsheet into R, we can use a text-file exchange. The basic idea being that the spreadsheet program writes out the data to a file which is read into R. Common formats for such exchange files are `csv` (comma-separated values), `tsv` (tab-separated values), or `fwf` (fixed-width format).

Note that also the 'Import Dataset' tab of RStudio (UR) can be used to perform this task. We consider two illustrative examples (from [2] and [3]).

Data Import (2)

```
# download the data file
download.file(
  url = "http://emilio.lcano.com/qcrbook/lab.csv",
  destfile = "lab.csv")
# read in the file
# you may also use the 'Import Dataset' tab (RStudio)
lab <- read.csv("lab.csv")

# cf. the R script for the second example
```

Be prepared that **raw data** like these probably contain **errors**! Since our data sets are rather large, it may not be an easy task to find out where (and of what kind) these errors are (missing values, wrong date formats, etc.). **Tidying** up a messy data set is an important task of its own! (Note that there are some packages, for example `tidyr`, which may be helpful.)

Data Import (3)

Web-based data sets: Data sets of all kinds are to be found on the Internet. Many sites provide data in some regular format (though these formats can be very different), other sites have data embedded in tables in a web page. Some of the functionality needed to access these data is provided through base R, but much is based on add-on packages. Here we briefly consider two add-ons, the `quandl` and the `XML` package.

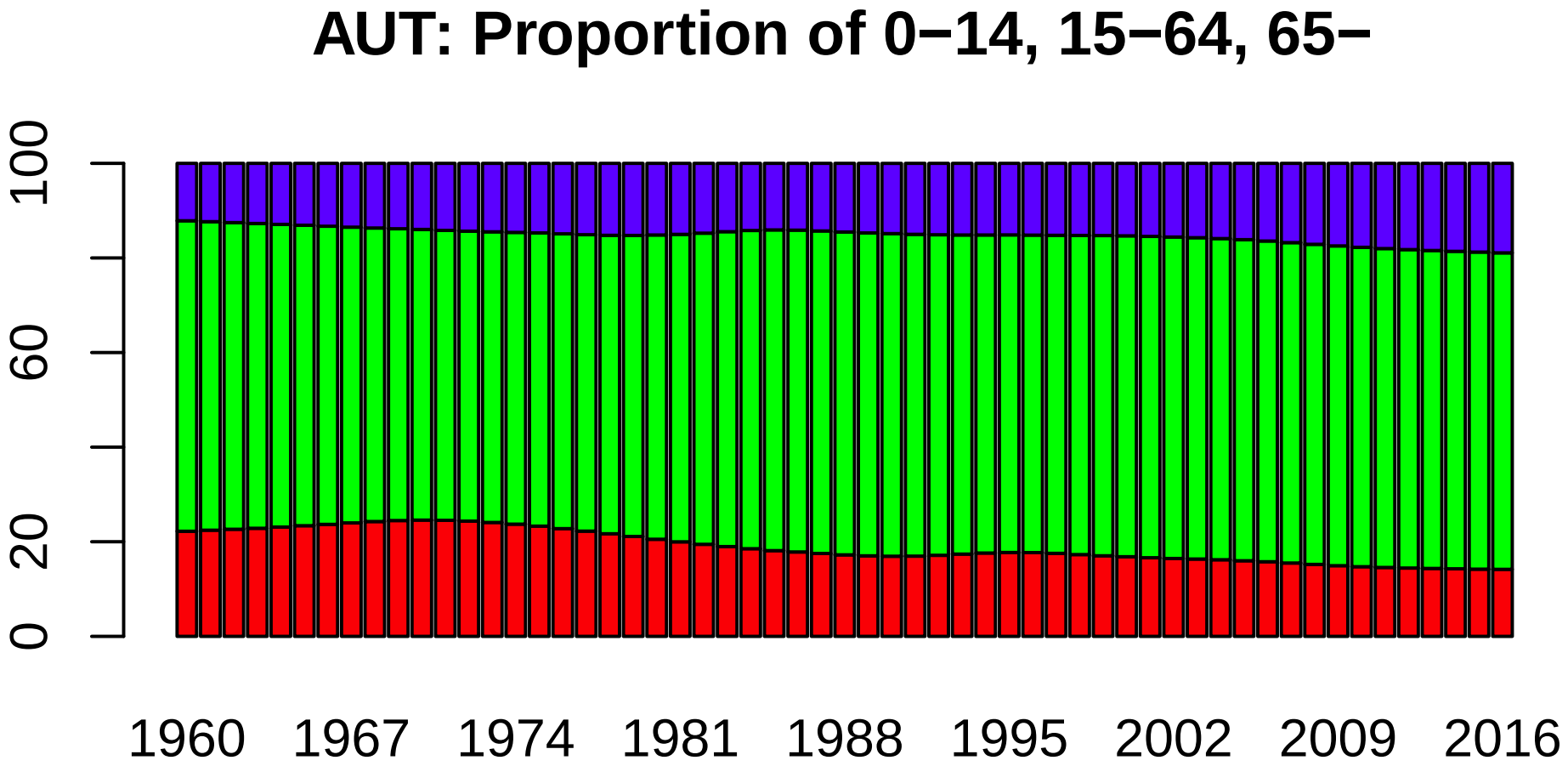
The quandl package: `Quandl.com` is a web site that indexes time-series data from numerous sources. It has millions of data sets, and even better, an open interface for downloading (and uploading) data. To download a file is as easy as browsing the site to the data of interest, and noticing the assigned code.

In the following example we refer to data on demographics of the Austrian population from the nineteen-sixties till 2016 by broad age groups (0–14, 15–64, 65–). We download three data sets and merge them into a single file. Then we use the `barplot` function to visualize the data.

Data Import (4)

```
require(Quandl)
# download the data sets
# Note: WWDI = World Bank World Development Indicators
at_0014 <- Quandl("WWDI/AUT_SP_POP_0014_TO_ZS")
at_1564 <- Quandl("WWDI/AUT_SP_POP_1564_TO_ZS")
at_65up <- Quandl("WWDI/AUT_SP_POP_65UP_TO_ZS")
at_all <- Reduce(function(x, y) merge(x, y, by="Date"),
                 list(at_0014, at_1564, at_65up))
# prepare for visualization
names(at_all) <- c("Date", "[0,14]","[15,64]","[65,)")
heights <- t(at_all[,-1])
colnames(heights) <- format(at_all[,"Date"], format="%Y")
barplot(heights, main="AUT-Proportion of 0-14, 15-64, 65-",
        col=rainbow(3))
```

Data Import (5)



Data Import (6)

Parsing HTML files: Frequently the data of interest is a table within an HTML page. For such data there are a variety of web-scraping techniques available in R. Using functions from the `XML` (and the `xml2`) package, the structure of a web page can be broken into pieces. Working directly with these pieces may not be practical, but the `readHTMLTable` function does the work for us.

For example, suppose that we want to import a table (seen on Wikipedia) of highest-grossing films (in Canada and the United States) into R. Using the `readHTMLTable` function directly results in an error. Thus we first use some parsing functions from the `xml2` package.

Note: Note that the apparently 'numerical' columns are all imported as characters (with special symbols "$", ",") and not numbers. Thus, before we can proceed with our statistical analysis, we have to convert the data into an usable format.

Data Import (7)

```
require(XML)
require(xml2)
url_base = "https://en.wikipedia.org/wiki/"
url_add1 <- "List_of_highest-grossing_films_in_"
url_add2 <- "Canada_and_the_United_States"
url_all <- paste(url_base, url_add1, url_add2, sep="")
u <- read_html(url_all)
doc <- htmlParse(u)
tableNodes <- getNodeSet(doc, "//table")
# first table
tb1 <- readHTMLTable(tableNodes[[1]])
View(tb1)
# second table
tb2 <- readHTMLTable(tableNodes[[2]])
View(tb2)
```

Graphics (1)

Standard plots (e. g., histograms, barplots, scatterplots, etc.) can easily be made with the `graphics` package. It comes with base R and we will use it throughout the lecture. The procedure is to start with a simple plot and then add more details.

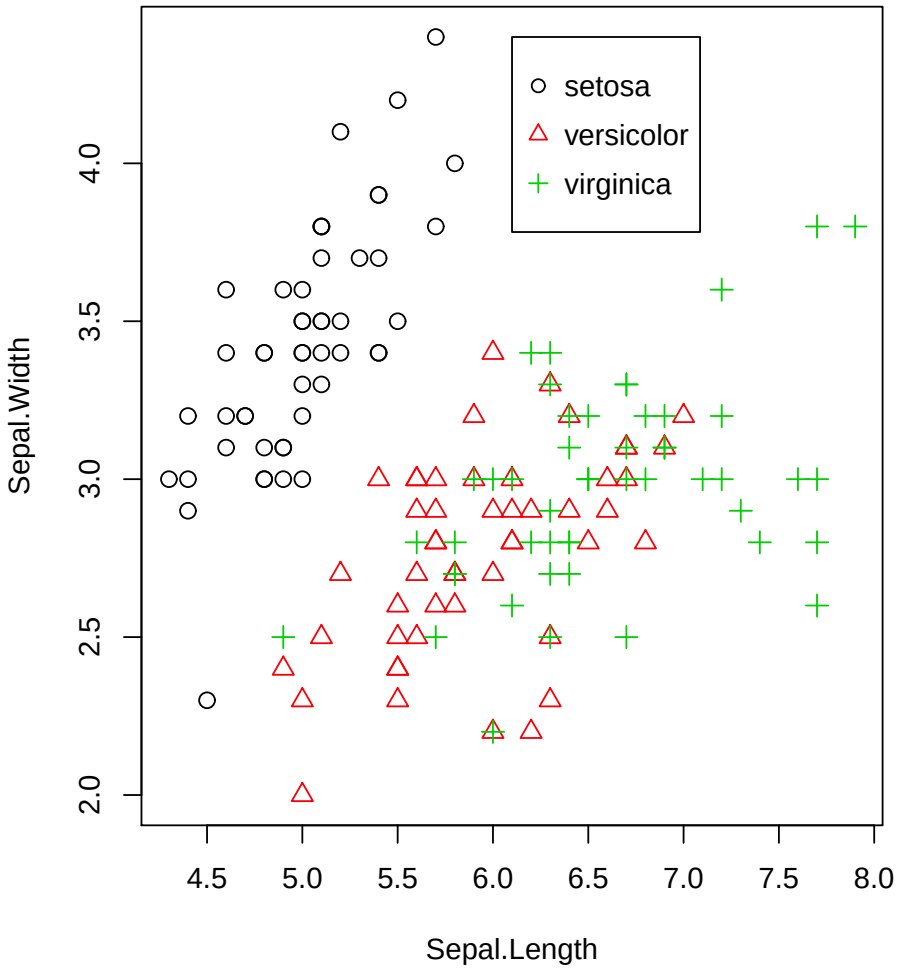
Over the years there have been various extensions to the base graphics interface that enhance its abilities to make more complex plots (e. g., for representing multivariate data) with a few commands. We briefly discuss two of these add-ons, the `lattice` package (comes with base R) and the `ggplot2` package (an add-on of relatively recent origin).

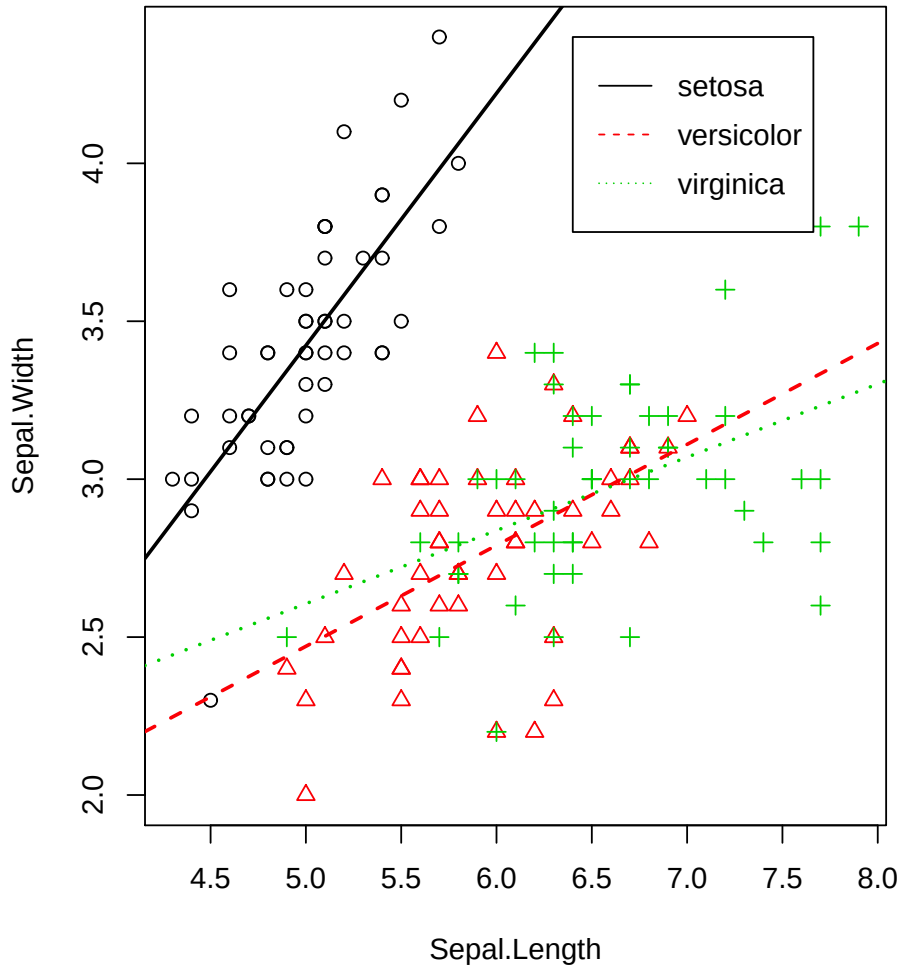
graphics : In the following example we use the `iris` data set, consisting of measurements of sepal length and width and petal length and width for 50 flowers from each of 3 species of iris. Suppose we want to give a graphical representation of sepal length and width, thereby differentiating between the species. We use only standard graphics commands.

Graphics (2)

```
# first plot (left)
with(iris, plot(Sepal.Length, Sepal.Width,
                pch=as.numeric(Species),
                col=as.numeric(Species), cex=1.2))
legend(6.1, 4.4, c("setosa","versicolor","virginica"),
       cex=1, pch=1:3, col=1:3)
# second plot (right)
fm <- Sepal.Width ~ Sepal.Length
plot(fm, iris, pch=as.numeric(Species),
     col=as.numeric(Species))
out <- mapply(function(i, x) abline(lm(fm, data=x),
              lty=i, lwd=2, col=i), i=1:3,
              x=split(iris, iris$Species))
legend(6.4, 4.4, levels(iris$Species), cex=1,
       lty=1:3, col=1:3)
```

Graphics (3)


setosa
versicolor
virginica
Sepal.Width
Sepal.Length
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


setosa
versicolor
virginica
Sepal.Width
Sepal.Length
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

Graphics (4)

lattice : The `lattice` package is inspired by the concept of **trellis plots** and can create a number of elegant plots with an emphasis on multivariate data with various dependencies. However, working with the `lattice` package is not as easy as working with the `graphics` package.

Lattice has a different though similar naming scheme for various standard graphics, including `histogram`, `barchart`, etc. In place of `plot` we use `xyplot`. Referring to our previous example, using the following command produces a scatterplot of sepal width and length for each species:

```
xyplot(Sepal.Width ~ Sepal.Length | Species, data=iris)
```

This basic plot can be modified in various ways. For example, the panels can be arranged in a row, regression lines can be added, etc. For this latter task so-called **panel functions** can be used (or defined). A panel function is what is called to draw each panel.

Graphics (5)

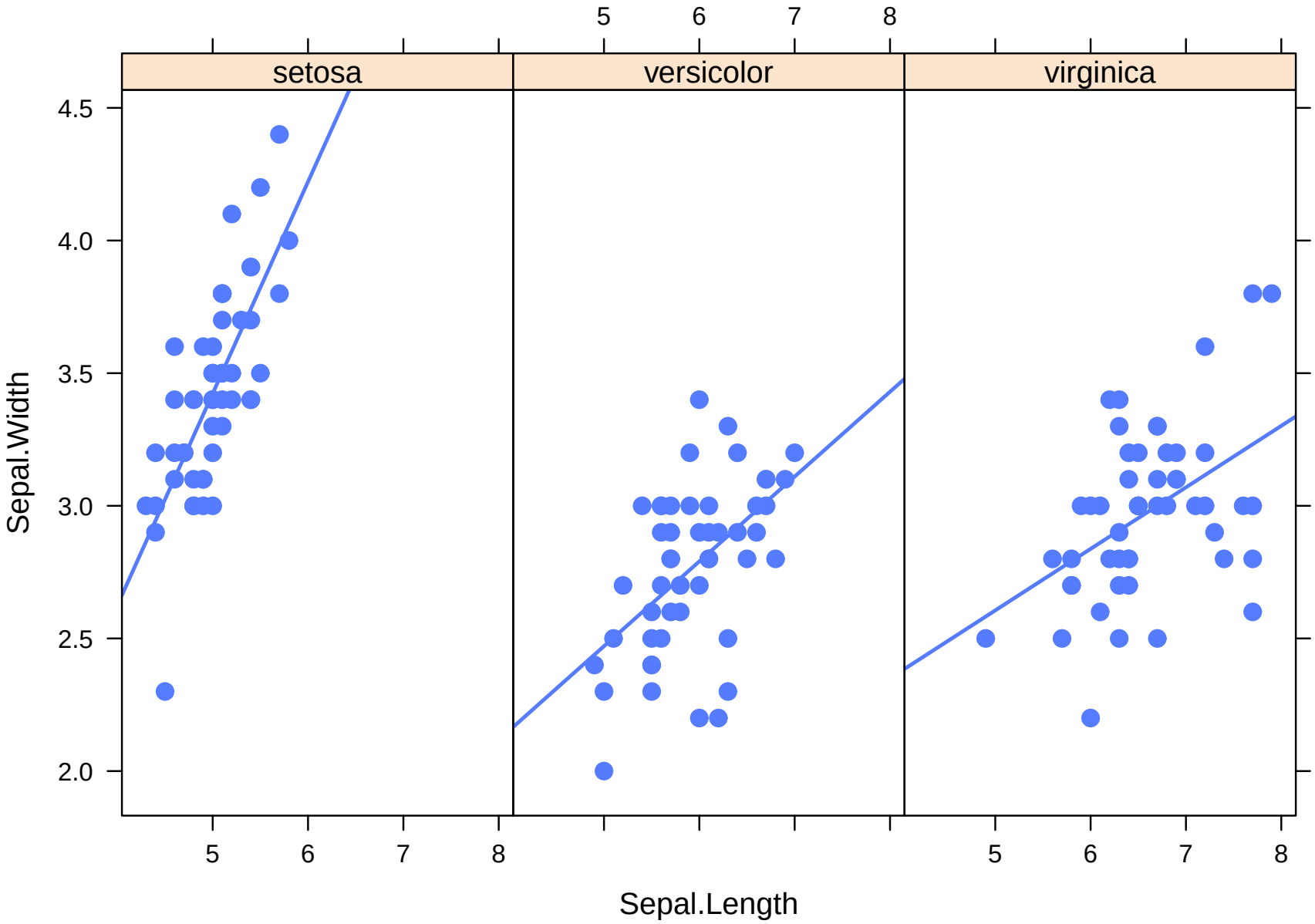
A specification such as the following plots the points (`panel.xyplot`) and the regression line (`panel.lmline`):

```
panel = function(x, y) {
  panel.xyplot(x, y)
  panel.lmline(x, y)
}
```

As this is a common task, there is a more convenient way using the `type` argument (`"p"` = points, `"r"` = regression line). The following command produces a somewhat nicer graphic shown on the next slide.

```
xyplot(Sepal.Width ~ Sepal.Length | Species,
       data=iris, pch=19, cex=1.1, lwd=2,
       layout=c(3,1), type=c("p", "r"))
```

Graphics (6)


setosa
versicolor
virginica
Sepal.Width
Sepal.Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8

Graphics (7)

ggplot2 : This is more than an add-on package, it implements a coherent system for describing and building graphs, called **grammar of graphics**. (The site `ggplot2.org` hosts the documentation in an easy to access format.) There are many benefits to making graphs with `ggplot2`, but perhaps the biggest is the visual appeal of the graphs produced.

Two terms are important in working with `ggplot2`:

Aesthetics: Aesthetics map variables in a data set into properties that can be perceived on a graph. For example, size, shape, and color are aesthetics. Moreover, values for `x` and `y` are aesthetics. Aesthetics are declared through the `aes` function.

Geoms: This is short for geometrical objects and refers to the functions that do the actual rendering of the data. These declare what should be drawn in the figure. For example, placing points on a graph is requested by `geom_ponts`, drawing lines by `geom_line`, etc.

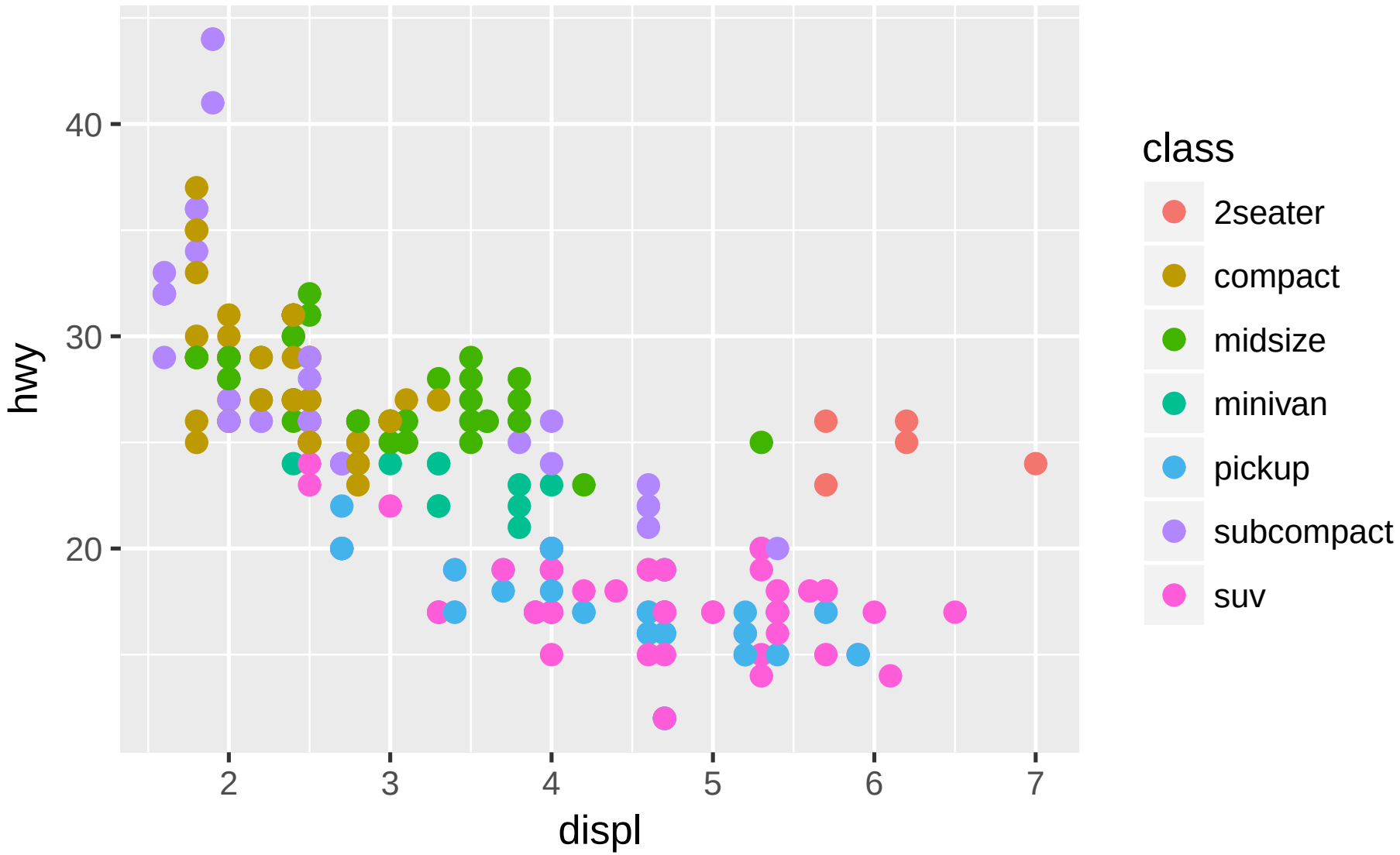
Graphics (8)

For our first example we use the `ggplot2::mpg` data frame (in fact, a tibble), containing observations on 38 models of cars. Amongst others, the variables `displ` (engine size, in liters), `hwy` (fuel efficiency on the highway, in miles per gallon), and `class` (type of car, midsize, suv, etc.) are observed.

Suppose we want to plot `hwy` against `displ`, differentiating between the types of the cars. We have several possibilities; below we map the colors of the points to the `class` variable to reveal the type of each car (clearly, we could also use different symbols or the like).

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy,
             color = class), size = 2.7))
```

Graphics (9)


hwy
40
30
20
2
3
4
5
6
7
displ
class
2seater
compact
midsize
minivan
pickup
subcompact
suv

Graphics (10)

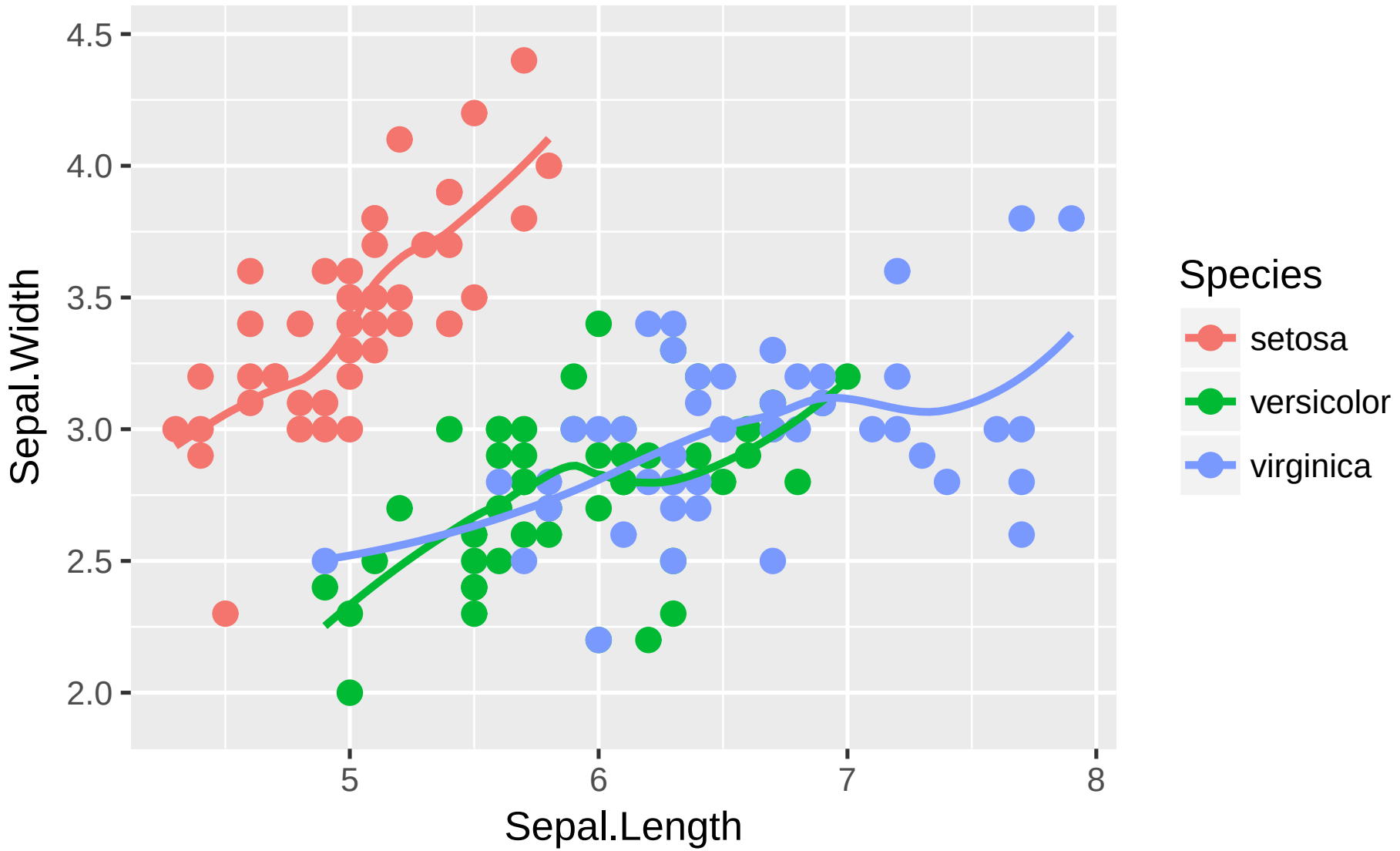
Let's consider again the `iris` data set. Suppose we want to produce a similar plot as on the right side of pane Graphics (3), this time using a 'smoother' instead of regression lines.

Several geoms may be added (with "+"). This, however, introduces some duplication in the code. We can avoid this by passing global mappings to `ggplot` that apply to each geom in the graph.

```
ggplot(data = iris,
  mapping = aes(x = Sepal.Length, y = Sepal.Width)) +
geom_point(mapping = aes(color = Species), size = 2.7) +
geom_smooth(mapping = aes(group = Species, color = Species))
```

The following slide shows the result, illustrating the versatile possibilities we have with the `ggplot2` package.

Graphics (11)


Sepal.Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal.Length
Species
setosa
versicolor
virginica