SCS2111-Laboratory II

Lecture 1-2

An Introduction to R

08/09/2015 & 15/09/2015

AMP

What is R?

- R is a free software environment for statistical computing and graphics
- R was created by Ross Ihaka and Robert Gentleman at the University of Auckland
- A GNU project which is similar to the **S** language,
  - developed at Bell Laboratories by John Chambers and colleagues.
  - **R** can be considered as a different implementation of **S**.
- The source code for the R software environment is written primarily in C, Fortran, and R.
- Pre-compiled binary versions are provided for various OS
- http://www.r-project.org/index.html

Why learn R?

- R is FREE, easy to use, and open source.
  - Commercial options: SAS, SPSS
- The R language is widely used among statisticians and data miners for developing statistical software and data analysis
- The "de facto" standard for data analysis and data mining

- A complete programming language
- Comes with a large library of pre-defined functions
- Better suited for advanced users who want all the power in their hands
  - R supports matrix arithmetic
  - R's data structures include vectors, matrices, arrays, data frames (similar to tables in a relational database) and lists.
  - R's extensible object system includes objects for (among others): regression models, time-series and geo-spatial coordinates.

The 2015 Top Ten **Programming Languages**

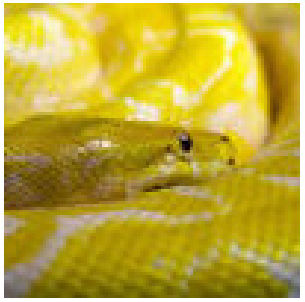
IEEE Spectrum - Jul 20, 2015

What are the most popular **programming languages**? ... The big mover is **R**, a statistical computing language that's handy for analyzing and ...

R Rises in IEEE Ranking of Top **Programming Languages**

ADT Magazine - Jul 21, 2015

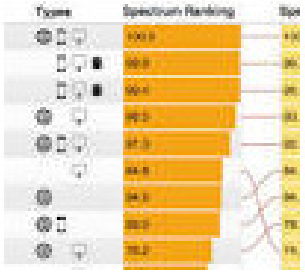
IEEE Spectrum has followed up last year's report on the top **programming languages** with a new study that sees **R** making a big jump in the ...



In data science, **the R language** is swallowing Python

InfoWorld - Jul 24, 2015

It's always precarious to compare **programming languages**, given their ... While **R** is a language developed by and for statisticians, Python has ...

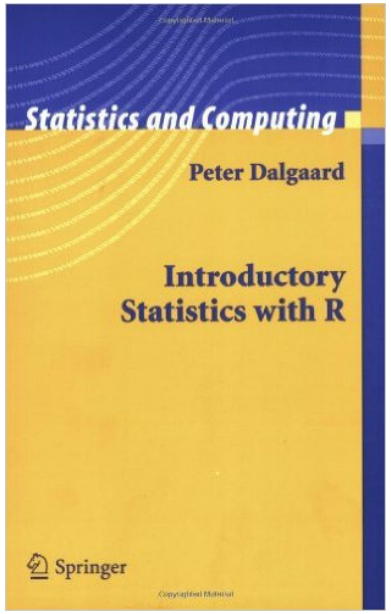


The Most Popular **Programming Languages** of 2015

ProgrammableWeb - Aug 4, 2015

While the top 5 remain unchanged, C has moved within touching distance of Java, and statistical **programming language R** has jumped from ...

Learning Resources



- "Introductory Statistics with R", by Peter Dalgaard, Springer (2002)
- *An Introduction to R. Notes on R: A Programming Environment for Data Analysis and Graphics*, by W. N. Venables, D. M. Smith.
  - http://math.arizona.edu/~hzhang/math574m/R-intro.pdf
- An Introduction R: Introduction and examples, by Deepayan Sarkar
  - http://www.isid.ac.in/~deepayan/R-tutorials/labs/01_introduction_lab.pdf

Interacting with R

- R is an interpreted language; users typically access it through a command-line interpreter.
- There are also several graphical front-ends for it.
- Unlike languages like C, Fortran, or Java, R is an interactive programming langauge.
- This means that R works interactively, using a question-and-answer model:
  - Start **R**
  - Type a command and press **Enter**
  - **R** executes this command (often printing the result)
  - **R** then waits for more input
  - Type **q()** to exit

Here are some simple examples:

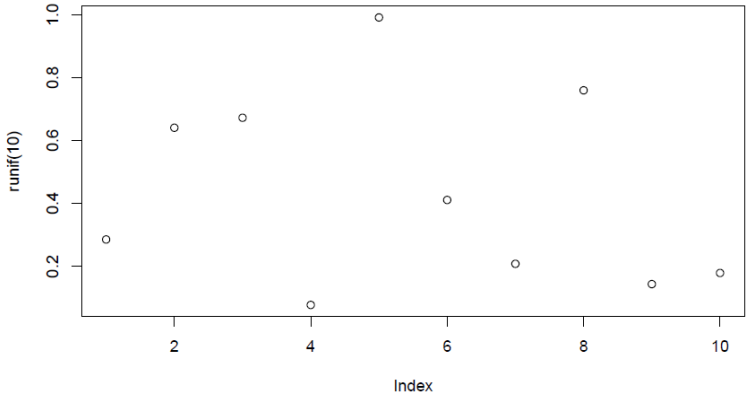
- Taken from AN INTRODUCTION TO R by Deepayan Sarkar

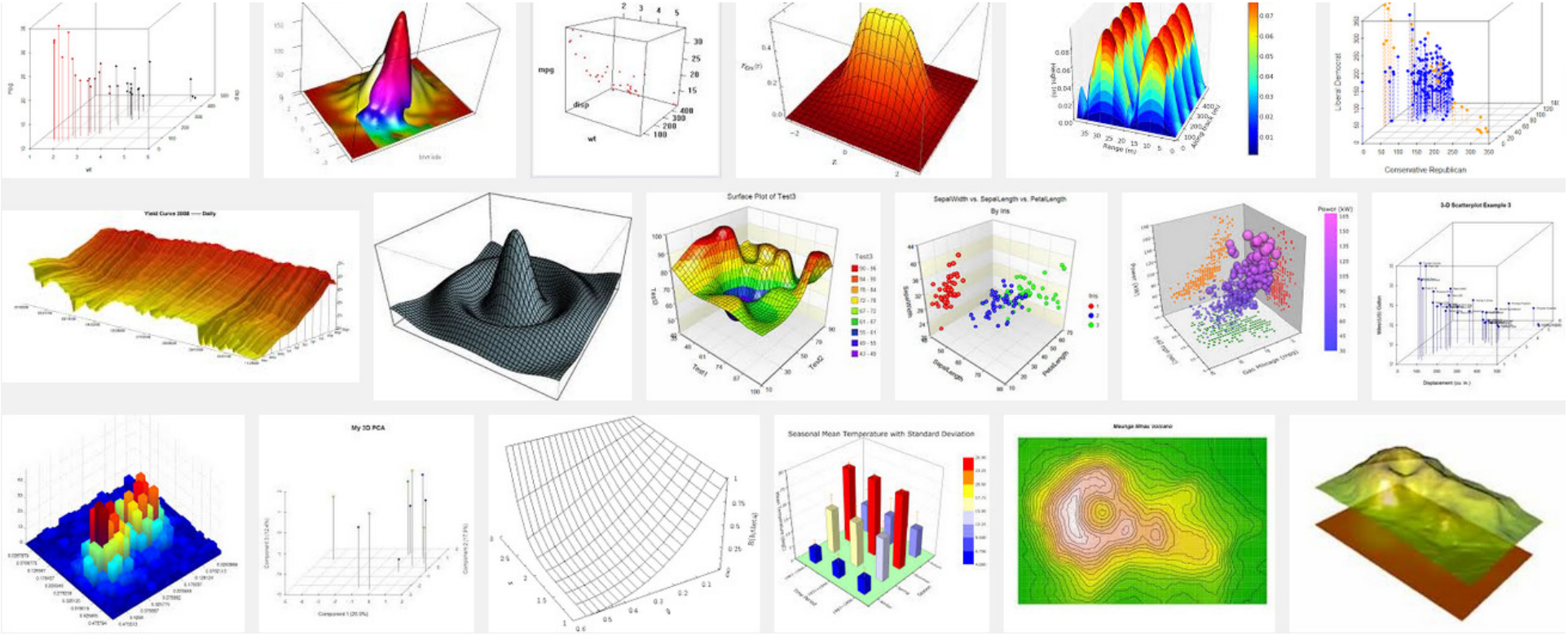
```
> 2 + 2
[1] 4
> exp(-2) ## exponential function
[1] 0.1353353
> log(100, base = 10)

[1] 2
> runif(10)
 [1] 0.39435130 0.98811744 0.07357143 0.16689946 0.80572031 0.05292909
 [7] 0.70498250 0.18781961 0.07865185 0.21618324
```

- The last command generates ten U(0; 1) random variables; the result (which is printed) is a vector of 10 numbers. **exp()**, **log()**, and **runif()** are functions.

Plots

```
> plot(runif(10))
```





Variables

- R has symbolic variables which can be assigned values.
- Assignment is done using the '<-' operator.
- The more C-like '=' also works (with some exceptions).

```
> s <- "this is a character string"
> s
[1] "this is a character string"
```

```
> x <- 2
> x + x
[1] 4
> yVar2 = x + 3
> yVar2
[1] 5
```

- Variable names can be almost anything, but they should not start with a digit, and should not contain spaces. Names are case-sensitive.
- Some common names are already used by R (c, q, t, C, D, F, I, T) and should be avoided.

Vectorized arithmetic

- The elementary data types in R are all vectors; even the scalar variables we defined above are stored as vectors of length one.
- The **c(...)** construct can be used to create vectors:

```
> weight <- c(60, 72, 57, 90, 95, 72)
> weight
[1] 60 72 57 90 95 72
```

- To generate a vector of regularly spaced numbers, use

```
> seq(0, 1, length = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

- The c() function can be used to combine vectors as well as scalars,

```
> x <- seq(0, 1, length = 6)
> c(x, 1:10, 100)
 [1]   0.0   0.2   0.4   0.6   0.8   1.0   1.0   2.0   3.0   4.0   5.0   6.0
[13]   7.0   8.0   9.0  10.0 100.0
```

- Common arithmetic operations (including +, -, *, /, ^) and mathematical functions (e.g. sin(),cos(), log()) work element-wise on vectors, producing another vector:

```
> height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
> height^2
[1] 3.0625 3.2400 2.7225 3.6100 3.0276 3.6481
> bmi <- weight / height^2
> bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.736
> log(bmi)
[1] 2.975113 3.101093 3.041501 3.216102 3.446107 2.9824
```

- When two vectors are not of equal length, the shorter one is recycled.
  - E.g.: The following adds 0 to all the odd elements and 2 to all the even elements of 1:10:

```
> 1:10 + c(0, 2)
 [1]  1  4  3  6  5  8  7 10  9 12
```

Summaries

- Many functions summarize a data vector by producing a scalar from a vector, e.g.,

```
> sum(weight)
[1] 446
> length(weight)
[1] 6
> avg.weight <- mean(weight)
> avg.weight
[1] 74.33333
```

- Simple summary statistics (mean, median, s.d., variance) can be computed from numeric vectors using appropriately named functions:

```
> x <- rnorm(100)
> mean(x)
[1] -0.1354077
> sd(x)
[1] 1.007307
> var(x)
[1] 1.014668
> median(x)
[1] -0.06083453
```

- Quantiles can be computed using the quantile() function.

- IQR() computes the inter-quartile range (**midspread** or **middle fifty)**.

```
> xquants <- quantile(x)
> xquants
         0%         25%         50%         75%        100%
-3.14440776 -0.74831291 -0.06083453  0.50980136  2.19369423
> xquants[4] - xquants[2]
     75%
1.258114
> IQR(x)
[1] 1.258114
> quantile(x, probs = c(0.2, 0.4, 0.6, 0.8))
        20%         40%         60%         80%
-1.0308886 -0.4388473  0.1236059  0.7357803
```

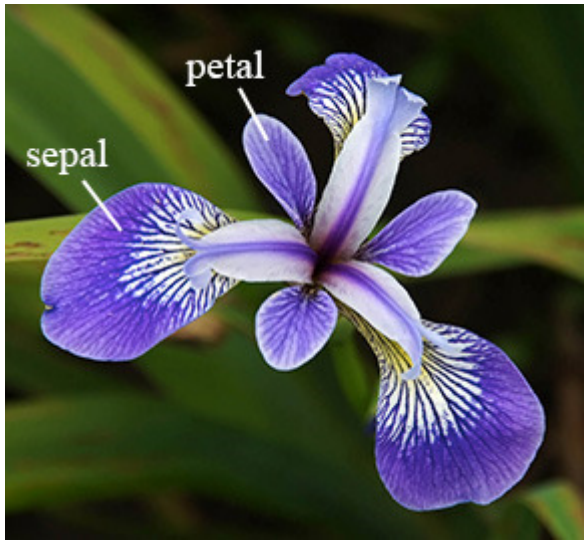
- The five-number summary (minimum, maximum, and quartiles) is given by fivenum(). A slightly extended summary is given by summary().

```
> fivenum(x)
[1] -3.14440776 -0.75013378 -0.06083453  0.51742360  2.19369423
> summary(x)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-3.14400 -0.74830 -0.06083 -0.13540  0.50980  2.19400
```

Object-oriented progamming: classes and methods

- Let's illustrate using a real dataset, one of the many datasets built into R (The well-known Iris data).
  - The dataset contains measurements on 150 flowers, 50 each from 3 species: Iris setosa, versicolor and virginica.

Iris setosa



Iris setosa

Iris virginica

The Iris data

- It is typically used to illustrate the problem of classification : Given the four measurements for a new flower, can we predict its Species?
- Like most datasets, iris is not a simple vector, but a composite “data frame" object made up of several component vectors.
- We can think of a data frame as a matrix-like object, with each row representing an observational unit (in this case, a flower), and columns representing multiple measurements made on the unit.

- The Iris data : The head() function extracts the first few rows, and the $ operator extracts individual components.

```
> head(iris) # The first few rows
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
> iris$Sepal.Length
  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
 [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
 [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
 [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
 [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
 [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
[109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
[127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
[145] 6.7 6.7 6.3 6.5 6.2 5.9
```

- A more concise description is given by the str() function (short for “structure").

```
> str(iris)
'data.frame':        150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1
```

- As we can see,
  - the first four components of iris are numeric vectors,
  - but the last is a “factor". These are how R represents categorical variables.

- Let us now see the effect of calling summary() for different types of objects.
  - Note the different formats of the output.
  - Species is summarized by the frequency distribution of its values because it is a categorical variable, for which mean or quantiles are meaningless.
  - The entire data frame iris is summarized by combining the summaries of all its components.

```
> summary(iris$Sepal.Length)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.300   5.100   5.800   5.843   6.400   7.900
> summary(iris$Species)
    setosa versicolor  virginica
        50         50         50
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

- R achieves this kind of object-specific customized output through a fairly simple object-oriented paradigm.
- Each R object has a class (“numeric", “factor", etc.).

```
> class(iris$Sepal.Length)
[1] "numeric"
> class(iris$Species)
[1] "factor"
> class(iris)
[1] "data.frame"
```

- summary() is what is referred to as a generic function, with class-specific methods that handle objects of various classes.
- When the generic summary() is called, R figures out the appropriate method and calls it.

- The rules are fairly intuitive.
- The last call gives the list of all available methods.

```
> methods(summary)
 [1] summary.aov             summary.aovlist         summary.aspell*
 [4] summary.connection      summary.data.frame      summary.Date
 [7] summary.default         summary.ecdf*           summary.factor
[10] summary.glm             summary.infl            summary.lm
[13] summary.loess*          summary.manova          summary.matrix
[16] summary.mlm             summary.nls*            summary.packageStatus*
[19] summary.PDF_Dictionary* summary.PDF_Stream*     summary.POSIXct
[22] summary.POSIXlt         summary.ppr*            summary.prcomp*
[25] summary.princomp*       summary.srcfile         summary.srcref
[28] summary.stepfun         summary.stl*            summary.table
[31] summary.tukeysmooth*

   Non-visible functions are asterisked
```

- Objects of class “factor” are handled by summary.factor(),
- “data.frame”s are handled by summary.data.frame().
- There is no summary.numeric(), so numeric vectors are handled by summary.default().

Getting Help

- help.start() starts a browser window with an HTML help interface.

- help(topic) displays the help page for a particular topic. Every R function has a help page.

```
> help(plot)
> ?plot
```

```
> help(plot, help_type = "html")
```

- help.search("search string") performs a subject/keyword search.

```
> help.search("logarithm")
> ??logarithm
```

- To directly run the examples given in help pages, use the example() function

```
> example(plot)
```

- The apropos() function, lists all functions (or other variables) whose name matches a specified character string.

```
> apropos("plot")
 [1] "assocplot"           "barplot"             "barplot.default"
 [4] "biplot"              "boxplot"             "boxplot.default"
 [7] "boxplot.matrix"      "boxplot.stats"       "cdplot"
[10] "coplot"              ".__C__recordedplot"  "fourfoldplot"
[13] "interaction.plot"    "lag.plot"            "matplot"
[16] "monthplot"           "mosaicplot"          "plot"
[19] "plot.default"        "plot.density"        "plot.design"
[22] "plot.ecdf"           "plot.function"       "plot.lm"
[25] "plot.mlm"            "plot.new"            "plot.spec"
[28] "plot.spec.coherency" "plot.spec.phase"     "plot.stepfun"
[31] "plot.ts"             "plot.TukeyHSD"       "plot.window"
[34] "plot.xy"             "preplot"             "qqplot"
[37] "recordPlot"          "replayPlot"          "savePlot"
[40] "screeplot"           "spineplot"           "sunflowerplot"
[43] "termplot"            "ts.plot"
```

- Further Reading:
  - For a useful list of "standard" packages in R, see http://cran.fhcrc.org/doc/contrib/refcard.pdf
  - Browse through the index of help pages in specific packages, produced by

```
> library(help = base)
> library(help = graphics)
```

etc., and read the topics that seem interesting.

Importing Data

- Use the read.table() function or one of its close relatives.

  > ?read.table

  E.x.: Download the Exam scores data from
  http://www.bristol.ac.uk/cmm/learning/support/datasets/

- Read in the data (from SCI.dat) and give appropriate names to the columns (read SCI.asc).
- How should you handle the missing values?
- Some of the variables read in as numeric are actually categorical variables.
  - How would you convert them to factors?

Packages

- Each package is a collection of functions (and data) with a common theme;
- the core of R itself is a package called base.
- A collection of packages is called a library.

- To see the pre-installed packages

```
> ip <- installed.packages()
> rownames(ip)
 [1] "base"         "boot"         "class"        "clust
 [6] "compiler"     "datasets"     "foreign"      "graph
[11] "grid"         "KernSmooth"   "lattice"      "MASS"
[16] "methods"      "mgcv"         "nlme"         "nnet"
[21] "rpart"        "spatial"      "splines"      "stats
[26] "survival"     "tcltk"        "tools"        "trans
> 
```

- Some packages are already attached when R starts up. Use search() function to see
- Other packages need be attached using the library() function.
- To attach, use library() function
  - E.g.: > library(class)

- More from the Comprehensive R Archive Network (CRAN) at http://cran.fhcrc.org/web/packages/

- To download and install, use install.packages()
  - E.g.: > install.packages("ISwR")
  - > library(help = ISwR) - list of all help pages in the package

Session management and serialization

- R has the ability to save objects, to be loaded again later.
- Whenever exiting, R asks to save all the variables created by the user, and restores them when starting up the next time (in the same directory).
- This is actually a special case of a very powerful feature of R called serialization.

- All R objects, however complex, can be saved as a file on disk, and re-read in a later session.
  - See ?save and ?load for details.

- >getwd() gives the current working directory
- >setwd() sets the wd

Expressions and Objects

- R works by evaluating expressions typed at the command prompt.
- Expressions involve variable references, operators, function calls, etc.
- Most expressions, when evaluated, produce a value,
  - which can be either assigned to a variable (e.g. x <- 2 + 2),
  - or is printed in the R session.
- Some expressions are useful for their side-effects
  - e.g., plot() produces graphical output.
- Evaluated expression values can be quite large, and often need to be re-used,
  - so it is good practice to assign them to variables rather than print them directly.

Expressions and Objects …

- Objects are anything that can be assigned to a variable.
- In the following example, c(1, 2, 3, 4, 5) is an expression that produces an object, whether or not the result is stored in a variable:

```
> sum(c(1, 2, 3, 4, 5))
[1] 15
> x <- c(1, 2, 3, 4, 5)
> sum(x)
[1] 15
```

- R has several important kinds of objects;
  - for example: functions, vectors (numeric, character, logical), matrices, lists, and data frames.

Functions

- Function calls look like a name followed by some arguments in parentheses. > plot(height, weight)
- All arguments have a formal name.
- Several ways to specify arguments:
  - By position: > plot(height, weight)
  - By name: > plot(x = height, y = weight)
  - With default values: > plot(height)

    Arguments will often have default values. If they are not specified in the call, these default values will be used.

“functions are first-class citizens" in R

- Functions are just like other objects in R
  - they can be used to return values in other functions,
  - they can be assigned to variables,
  - they can be used as arguments in other function calls.

New function objects

- defined using the construct expression function(arglist)
- The args() function, gives the arguments that a function accepts (along with their default values)
- Try > myfun
  - Gives the full definition of the function

```
> myfun <- function (a=1, b=2, c)
+ return(list(sum=a+b+c, product=a*b*c))
> myfun(6,7,8)
$sum
[1] 21

$product
[1] 336

> myfun(10, c=3)
$sum
[1] 15

$product
[1] 60

> args(myfun)
function (a = 1, b = 2, c)
NULL
```

Special Argument (ellipsis)

```
> args(plot.default)
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
    ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
    panel.last = NULL, asp = NA, ...)
NULL
```

- An exception:
  - Special argument ..., do not have a formal name
  - called "…", dots, dot-dot-dot or three-dots

- The three-dots allows:
  - an arbitrary number and variety of arguments :
    - indicates that the function can accept any number of further arguments.
    - What happens to those arguments is determined by the function.
  - passing arguments on to other functions

- Read more…. [p.12-14 of Deepayan & Sarkar, An intro to R]

Data Types in R

- There are five types of constants:
- numeric :
  - Valid numeric constants: 1, 10, 0.1, .2, 1e-7, 1.2e+7,
    (1e2= 100, 1e-2 = 0.01, -1e2 = -0.01)
  - can also be hexadecimal, starting with '0x' or '0X' followed by zero or more digits, 'a-f' or 'A-F',
    (0x0 = 0, 0xF = 15,
    0xFA = 15*16^1 + 10*16^0 = 250)
  - Hexadecimal floating point constants are supported using C99 syntax, e.g. '0x1.1p1',
    (0x0.1p0 = 1/16 * 2^0 = 0.0625,
    0x0.1p1 = 1/16 * 2^1 = 0.125)

  Values which are too large or too small to be representable will overflow to Inf or underflow to 0.0.

- Integer :
  - created by using the qualifier L (e.g. : 123L)
  - can be used with (non-complex) numbers given by hexadecimal or scientific notation
    - Valid integer constants: 1L, 0x10L, 1000000L, 1e6L
  - However, if the value is not a valid integer, a warning is emitted and the numeric value created.
    - Valid numeric constants: 1.1L, 1e-3L, 0x1.1p-2
  - Syntax errors: 12iL, 0x1.1
  - ?NumericConstants

```
> typeof(2)
[1] "double"
> typeof(2L)
[1] "integer"
> 2 == 2L
[1] TRUE
> identical(2, 2L)
[1] FALSE
```

- Logical : either TRUE or FALSE
- complex : A numeric constant immediately followed by i is regarded as an imaginary complex number.
  - 2i, 4.1i, 1e-2i

```
> 2i
[1] 0+2i
> 4.1i
[1] 0+4.1i
> 1e-2i
[1] 0+0.01i
> 2+1e-2i
[1] 2+0.01i
```

- String : delimited by a pair of single (‘’) or double (‘"’) quotes and can contain all other printable characters.
  - Quotes and other special characters within strings are specified using *escape sequences (e.g.: \n)*:

Special Types

- In addition, there are four special constants,
  - NULL : used to indicate the empty object
  - NA : for absent (“Not Available”) data values
  - Inf : denotes infinity
  - NaN: is not-a-number
- E.x.: Determine 1/0, 0/1, 0/0, -2/0

Special Types

- In addition, there are four special constants,
  - NULL : used to indicate the empty object
  - NA : for absent (“Not Available”) data values
  - Inf : denotes infinity
  - NaN: is not-a-number
- E.x.: Determine 1/0, 0/1, 0/0, -2/0

```
> 1/0
[1] Inf
> 0/1
[1] 0
> 0/0
[1] NaN
> -2/0
[1] -Inf
```

Vectors

- The basic data types in R are all vectors.
- The simplest varieties are numeric, character, and logical (TRUE or FALSE):

```
> c(1, 2, 3, 4, 5)
[1] 1 2 3 4 5
> c("Spring", "Summer", "Autumn", "Winter")
[1] "Spring" "Summer" "Autumn" "Winter"
> c(TRUE, TRUE, FALSE, TRUE)
[1]  TRUE  TRUE FALSE  TRUE
> c(1, 2, 3, 4, 5) > 3
[1] FALSE FALSE FALSE  TRUE  TRUE
```

- The length of any vector can be determined by the length() function:

```
> gt.3 <- c(1, 2, 3, 4, 5) > 3
> gt.3
[1] FALSE FALSE FALSE  TRUE  TRUE
> length(gt.3)
[1] 5
> sum(gt.3)
[1] 2
```

- This happens because of coercion from logical to numeric.

- seq() creates a sequence of equidistant numbers

```
> seq(4,10,0.5)
 [1]  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
> seq(length=10)
 [1]  1  2  3  4  5  6  7  8  9 10
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> args(seq.default)
function (from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
NULL
```

Note the *Partial Matching* of the length argument

- c() concatenates one or more vectors

```
> c(1:5, seq(10,20, length=6))
 [1]  1  2  3  4  5 10 12 14 16 18 20
```

- rep() replicates a vector

```
> rep(1:5, 2)
 [1] 1 2 3 4 5 1 2 3 4 5
> rep(1:5, length =12)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2
> rep(c('one', 'two'), c(6,3))
[1] "one" "one" "one" "one" "one" "one" "two" "two" "two"
```

- E.x.: Here are two simple numeric vectors containing *NA* and *Inf* values.

  > x <- c(1:5, NA, 7:10, NULL)

  > y <- c(1:5, Inf, 7:10)

  - Use R to find the mean and median of these vectors.

  - How can you make R ignore the NA values?

  - What is the length of the NULL object?

Arrays, Vectors

```
> a <- 1:10
> b <- array(1:10)
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> b
 [1]  1  2  3  4  5  6  7  8  9 10
> class(a)
[1] "integer"
> class(b)
[1] "array"
> a==b
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> identical(a,b)
[1] FALSE
```

Matrices and Arrays

- Matrices (and more generally arrays of any dimension) are stored in R as a vector with a dimension attribute:

```
> x <- 1:12
> dim(x) <- c(3, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> nrow(x)
[1] 3
> ncol(x)
[1] 4
```

```
> dim(x) <- c(2, 2, 3)
> x
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

- Replacement functions.
  - E.g.: dim(x) <- c(2, 2, 3)
  - changes the "dim" attribute and removes any "dimnames" *and* "names" attributes.

- the matrix() function

```
> x <- matrix(1:12, nrow = 3, byrow = TRUE)
> rownames(x) <- LETTERS[1:3]
> x
  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
```

- Matrices do not need to be numeric, there can be character or logical matrices as well:

```
> matrix(month.name, nrow = 6)
     [,1]       [,2]
[1,] "January"  "July"
[2,] "February" "August"
[3,] "March"    "September"
[4,] "April"    "October"
[5,] "May"      "November"
[6,] "June"     "December"
```

- Transpose : t() function

```
> x <- matrix(1:12, nrow = 3, byrow = TRUE)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> t(x)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

- Also see aperm() for array permutations

Matrix multiplication (%*%)

```
> x <- matrix(1:12, c(3,4), byrow=TRUE)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> x*x
```

Matrix multiplication (%*%)

```
> x <- matrix(1:12, c(3,4), byrow=TRUE)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> x*x
     [,1] [,2] [,3] [,4]
[1,]    1    4    9   16
[2,]   25   36   49   64
[3,]   81  100  121  144
```

Matrix multiplication (%*%)

```
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> t(x)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> x%*%t(x)
```

Matrix multiplication (%*%)

```
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> t(x)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> x%*%t(x)
     [,1] [,2] [,3]
[1,]   30   70  110
[2,]   70  174  278
[3,]  110  278  446
>
```

Creating matrices from vectors

- The cbind (column bind) and rbind (row bind functions can create matrices from smaller matrices or vectors

```
> y <- cbind(A = 1:4, B = 5:8, C = 9:12)
> y
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
> rbind(y,0)
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
[5,] 0 0  0
>
```

The short vector(0) is replicated

Factors

- Factors are how R handles categorical data (e.g., eye color, gender, pain level).
  - Such data are often available as numeric codes, but should be converted to factors for proper analysis.

```
> pain <- c(0, 3, 2, 2, 1)
> fpain <- factor(pain, levels = 0:3)
> fpain

[1] 0 3 2 2 1
Levels: 0 1 2 3

> levels(fpain) <- c("none", "mild", "medium", "severe")
> fpain

[1] none   severe medium medium mild
Levels: none mild medium severe

> as.numeric(fpain)

[1] 1 4 3 3 2
```

- Factors can also be created from character vectors.
  - levels are sorted alphabetically by default

```
> text.pain <- c("none", "severe", "medium", "medium", "mild")
> factor(text.pain)
[1] none   severe medium medium mild
Levels: medium mild none severe
```

  - Hint : use the gl() function

```
> ?gl
> a <- gl(5,2)
> a
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> levels(a)
[1] "1" "2" "3" "4" "5"
> levels(a) <- c("1", "2", "2", "4", "5")
> levels(a)
[1] "1" "2" "4" "5"
```

Lists

- Lists are very flexible data structures used extensively in R.
- A list is a vector, but the elements of a list do not need to be of the same type.
  - Each element of a list can be any R object, including another list.
- lists can be created using the list() function.
- List elements are usually extracted by name (using the $ operator).

```
> x<- list(a=1, b=2)
> x$a
[1] 1
> x[1]
$a
[1] 1
```

```
> list(1,'s',c(1:5))
[[1]]
[1] 1

[[2]]
[1] "s"

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

- Functions are R objects too.
  - the fun element of x is the seq() function,
  - And can be called like any other function.

```
> x <- list(fun = seq, len = 10)
> x
$fun
function (...)
UseMethod("seq")
<bytecode: 0x0000000014ce6150>
<environment: namespace:base>

$len
[1] 10

> x$fun(length=x$len)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Data Frames

- Data frames are R objects that represent (rectangular) data sets.
- Each column of a data frame has to be a either a factor or a numeric, character, or logical vector.
- Each of these must have the same length.
- similar to matrices because they have the same rectangular array structure;
  - the only difference is that different columns of a data frame can be of different data types.
- Data frames are created by the data.frame() function.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
>
```

Logical Comparisions

- All the usual logical comparisons are possible:

less than	<	less than or equal to	<=
greater than	>	greater than or equal to	>=
equals	==	does not equal	!=

- Element-wise boolean operations are also possible.

AND	&
OR	|
NOT	!

Remember the Iris Data?

Iris setosa

Iris setosa

Iris virginica

```
> str(iris)
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1
```

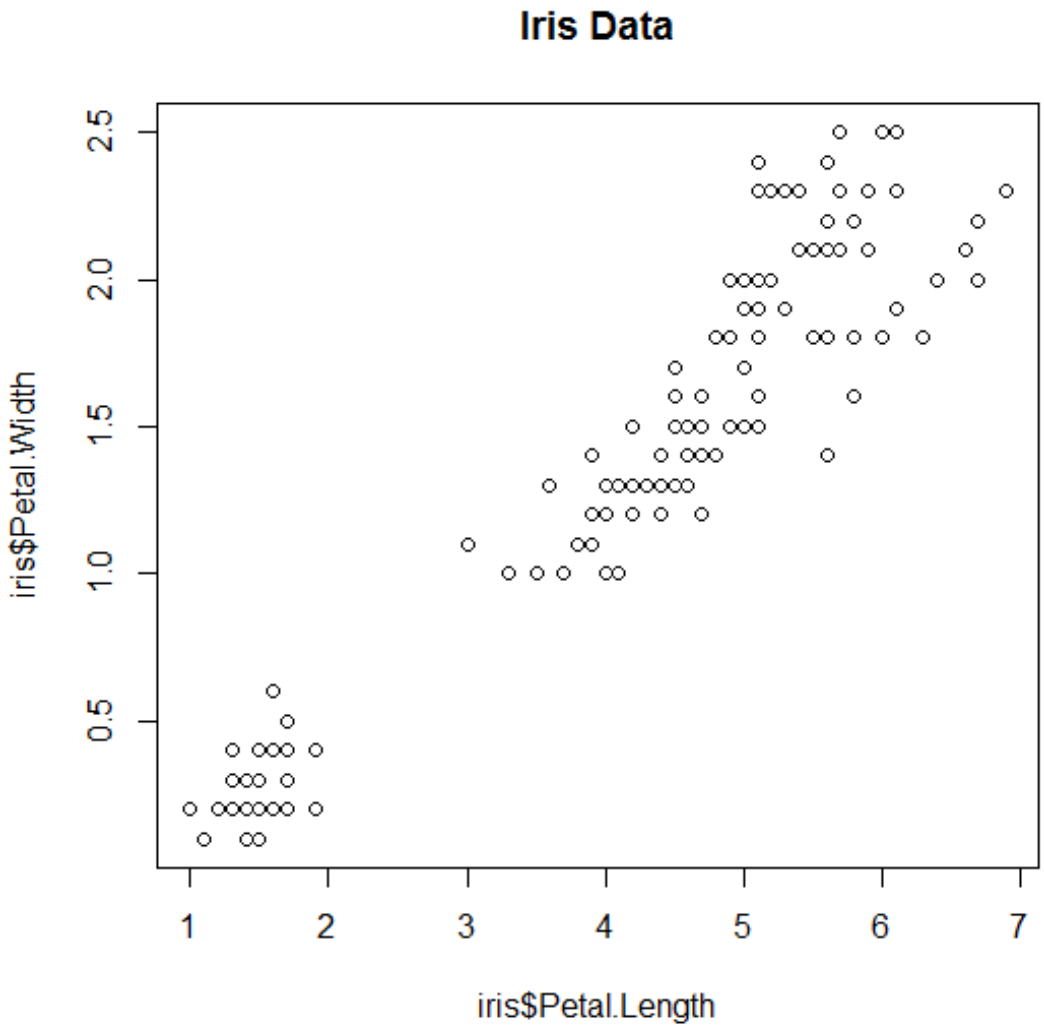
- The iris variable is a data.frame – its like a matrix but the columns may be of different types, and we can access the columns by name

```
> class(iris)
[1] "data.frame"
> colnames(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
> iris$Petal.Length
  [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4
 [19] 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2
 [37] 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7 4.5 4.9 4.0
 [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0
 [73] 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0
 [91] 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1 5.9 5.6 5.8 6.6 4.5 6.3
[109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0
[127] 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9
[145] 5.7 5.2 5.0 5.2 5.4 5.1
```

  - Or iris[,"Petal.Length"] or iris[,3], treating the data frame like a matrix/array

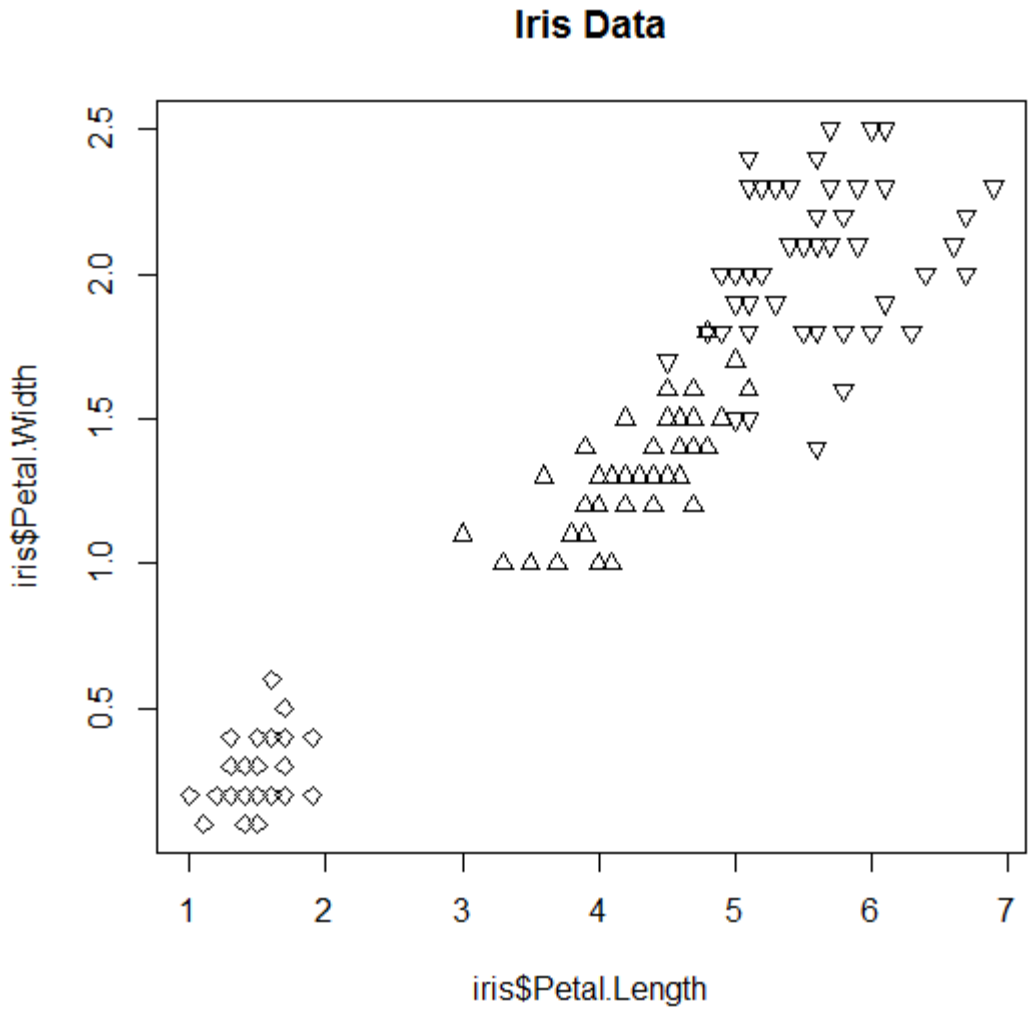
Simple Scatter Plots

```
> plot(iris$Petal.Length, iris$Petal.Width, main="Iris Data")
```



```
> plot(iris$Petal.Length, iris$Petal.Width,
    pch=c(23,24,25)[unclass(iris$Species)], main="Iris Data")
```

- the pch argument (plot character)
- pch=21 for filled circles,
- pch=22 for filled squares,
- pch=23 for filled diamonds
- pch=24 or pch=25 for up/down triangles.



Doing this would change *all* the points...

the trick is to create a list mapping the species to say 23, 24 or 25 and use that as the pch argument:

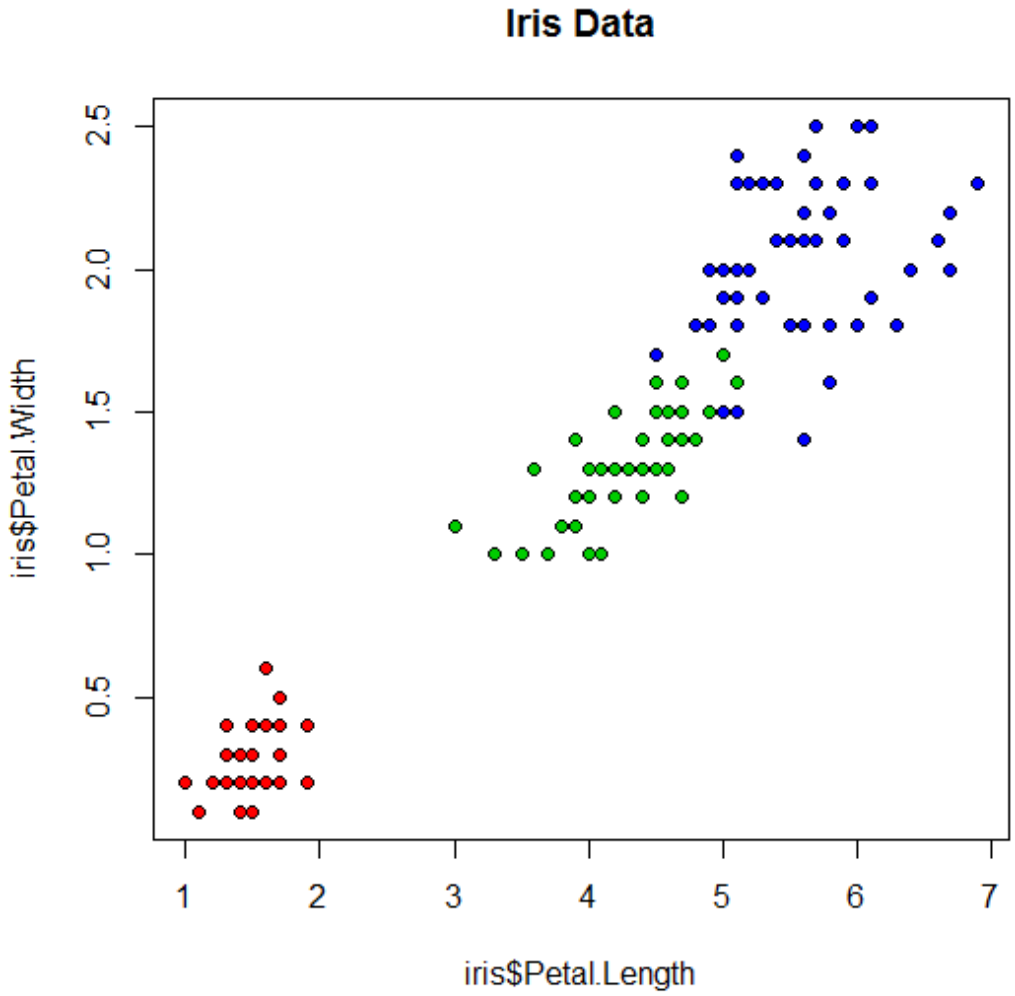
- This works by using c(23,24,25) to create a vector, and then selecting elements 1, 2 or 3 from it.
  - How? unclass(iris$Species) turns the list of species from a list of categories (a "factor" data type in R terminology) into a list of ones, twos and threes:

```
> unclass(iris$Species)
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [75] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3
[112] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
[149] 3 3
attr(,"levels")
[1] "setosa"     "versicolor" "virginica"
> c(23,24,25)[unclass(iris$Species)]
  [1] 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 2
 [26] 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 2
 [51] 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 2
 [76] 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 2
[101] 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 2
[126] 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 2
```

Use the same trick to generate a list of colours

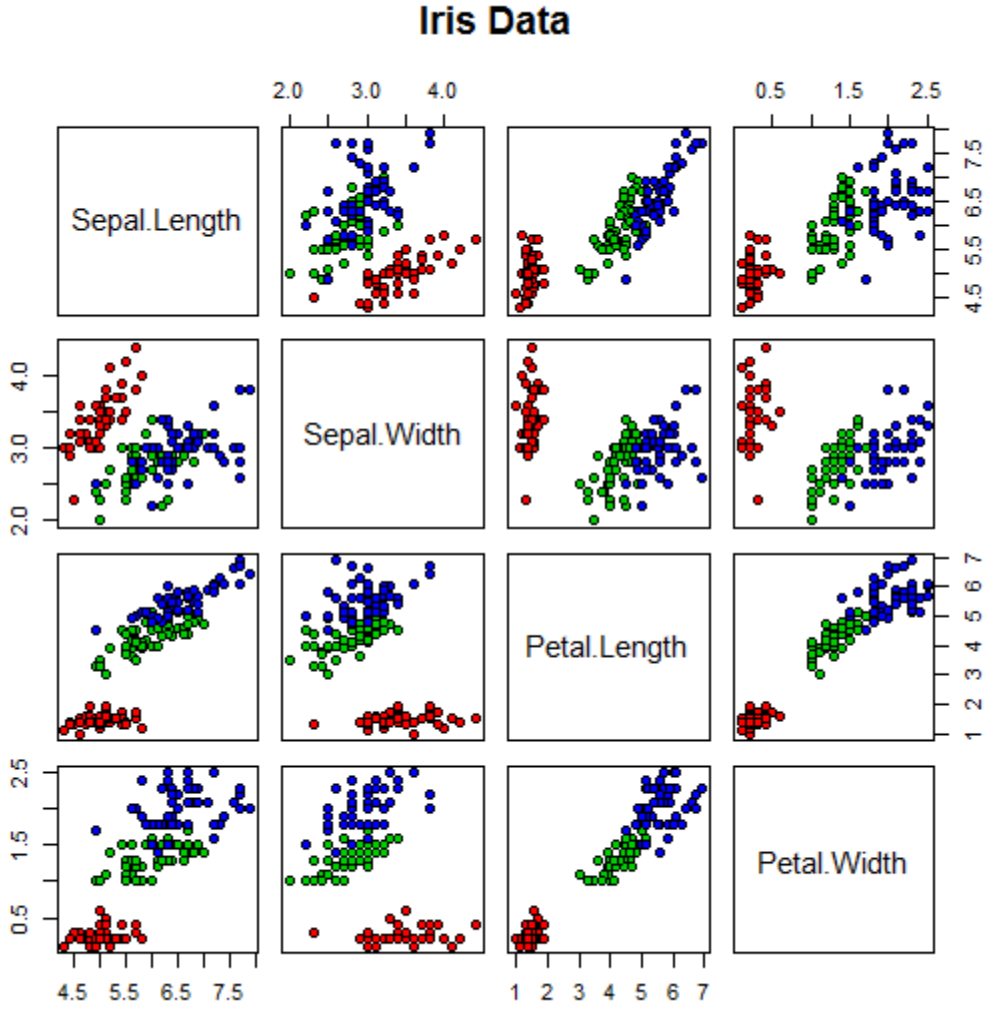
```
> plot(iris$Petal.Length, iris$Petal.Width, pch=21,
   bg=c("red","green3","blue")[unclass(iris$Species)]
   , main="Iris Data")
```

Using different colours,
It becomes clear that
the three species have
very different petal sizes.

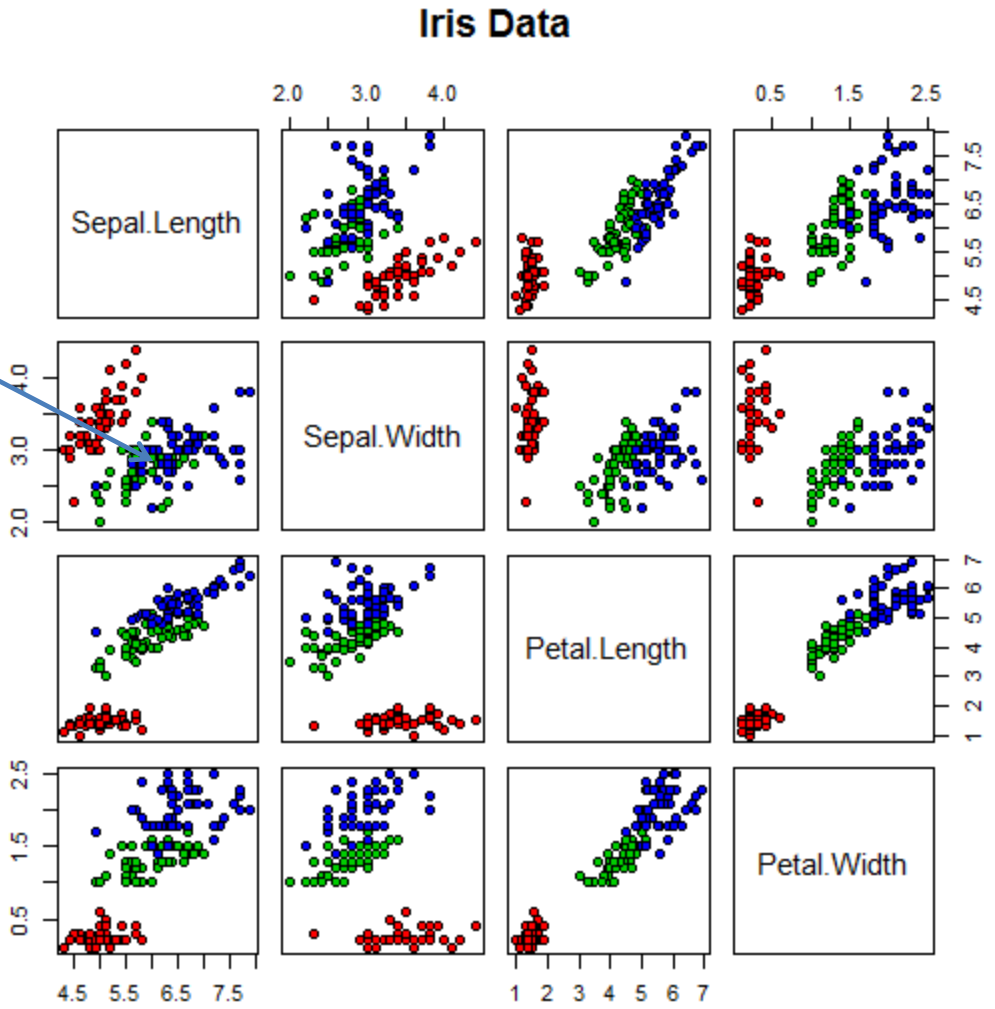


Draftsman's or Pairs Scatter Plots

```
> pairs(iris[1:4], main = "Iris Data", pch = 21, bg =
    c("red", "green3", “blue")[unclass(iris$Species)])
```



- Looks like most of the variables could be used to predict the species
  - except that using the sepal length and width alone would make distinguishing *Iris versicolor* and *virginica* tricky (green and blue)



http://www2.warwick.ac.uk/fac/sci/moac/people/students/peter_cock/r/iris_plots/

Sepal Area vs. Petal Area

http://www.statlab.uni-heidelberg.de/data/iris/

Only 3 misclassifications!!