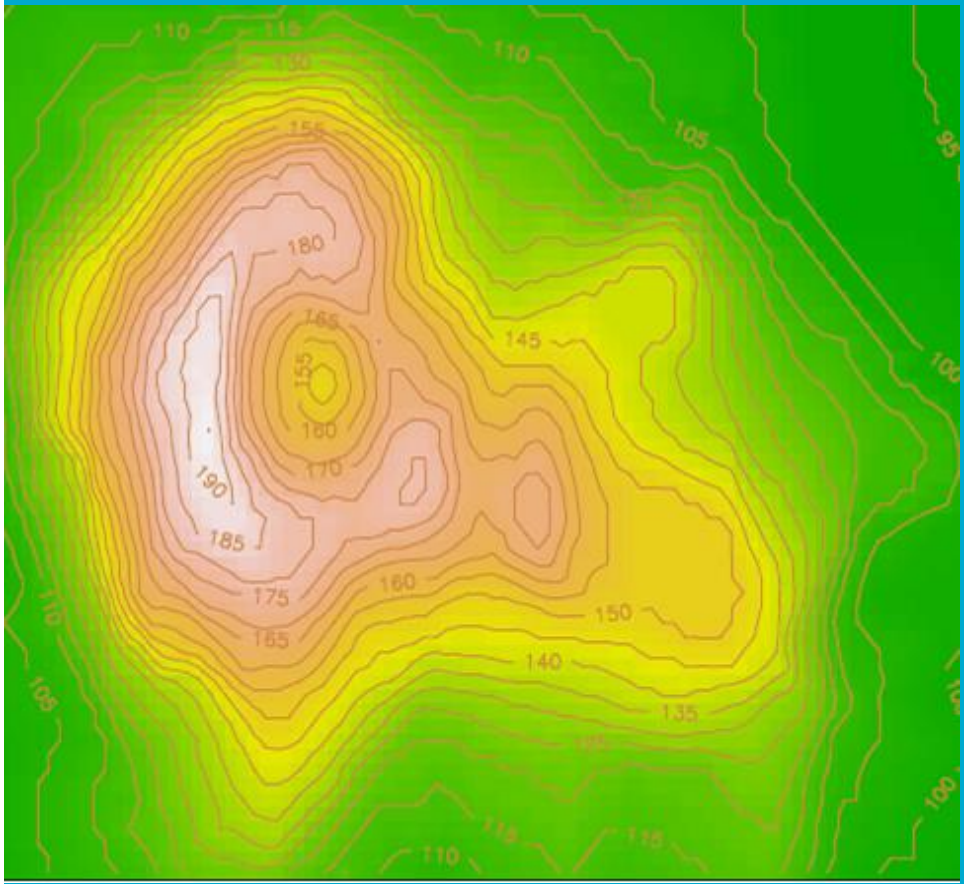
An Introduction to R

Part 1

Petra Kuhnert | 31 May 2024



I would like to begin by acknowledging the Quandamooka people as the Traditional Owners of the land that we're meeting on today, and pay my respect to their Elders past and present.

What we will cover

- A short history on R
- How to get help
- R Objects
- Grammar of Graphics
- Manipulating objects in R (base)
- R packages for data science
- Key References

A Short History on R

A brief history

- It all started from S
- Initially a language for technology transfer inside Bell Labs (c. 1977).
  - Primary developers: John Chambers (language), Richard Becker, Allan Wilks, Bill Cleveland (graphics)
- Main characteristics
  - Interactive, interactive, interactive
  - Flexible, encourage innovation and extension
  - Exploratory rather than production-line
  - Moving with technology
- Escaped into the wider world c. 1984

Commercialisation and birth of OOP

- Commercialised (c. 1985) from U. Washington in Seattle by Doug Martin and colleagues, and developed further by StatSci, Inc., later MathSoft, Inc, currently Insightful.
- S becomes expensive.
- S3 (1992) extends the system in three major directions:
  - An informal object oriented system is introduced. S becomes a language for manipulating objects. Objects become the vehicle for information transmission,
  - Modelling software using the new system: Linear models, AOV, GLMs, GAMs, Loess, Trees, some Multivariate.
  - Bill Cleveland's Trellis graphics launched, suggesting new standards in graphical presentation

Formal objects, methods and classes

- S4 introduced in 1998.
  - Formal object structure – representation as collection of slots",
  - Formal method dispatch – on on signatures, combinations of classes
- Commercial considerations require backward compatibility with S3.
- S3 refuses to die!

The rise and rise of R

- Auckland, New Zealand, c. 1985. Ron Gentleman and Ross Ihaka write a Scheme interpreter.
- proto-R is built on Scheme for teaching statistics on the Mac!
- R is released as open-source under GPL.
- R-1.0.0 (c. 2000). A Core Team oversees code development; TU Wien hosts primary distribution. A package system is used, updates via internet. Essentailly all features S-PLUS.
- John Chambers and Duncan Temple Lang join R-Core. S4 methods package developed.
- Major releases scheduled for (approimately) every 6 months, with minor releases following. (Currently R-4.4.0)

Viz and data wrangling

- In mid 2000's Hadley Wickham introduces ggplot2, revolutionising data visualisation in R.
  - ggplot2 is based on the Grammar of Graphics, making complex multi-layered plots easy to create.
- Development of dplyr, simplifying data manipulation with a consistent set of verbs (filter, select, mutate, etc.).
- Introduction of tidyverse, a collection of R packages designed for data science, which includes ggplot2, dplyr, tidyr, readr, purrr, and others.

- Late 2010s to Present:
  - Continuous improvements and expansions to the tidyverse ecosystem

More data wrangling improvements

- Continuous improvements and expansions to the tidyverse ecosystem.
- Development of new packages
  - tibble (for modern data frames),
  - stringr (for string manipulation),
  - forcats (for factor management)
- RStudio, an integrated development environment (IDE) for R, becomes the go-to tool for R users, enhancing the user experience with features for coding, visualization, and data management.

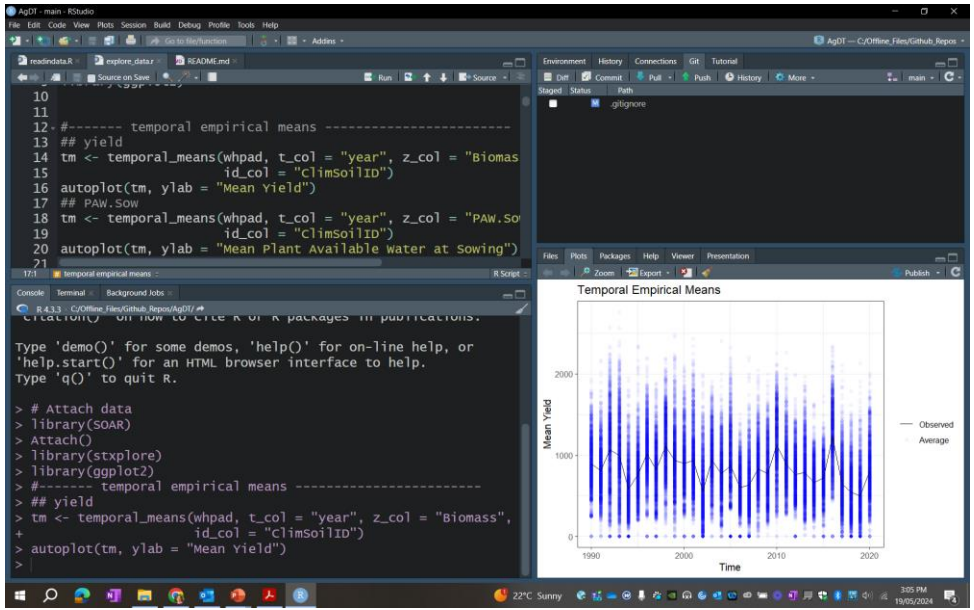
- Modern Day:
  - R is widely used in academia, research, and industry for statistical

RStudio

- An integrated development environment (IDE) for R
- Key Features:
  - **Script Editor:** Syntax highlighting, code completion.
  - **Console:** Run commands, view outputs.
  - **Environment/History:** Manage variables, view command history.
  - **Plots:** Real-time graphical outputs.
  - **Files/Packages:** Navigate directories, manage packages.
  - **Help/Viewer:** Access documentation, view web content.

RStudio

- Data Science Enhancements:
  - **R Markdown:** Create dynamic documents.
  - **Quarto: Unified authoring system for documents, presentations, and more.**
  - **Shiny:** Develop interactive web apps.
  - **Version Control:** Integrated Git/SVN support.

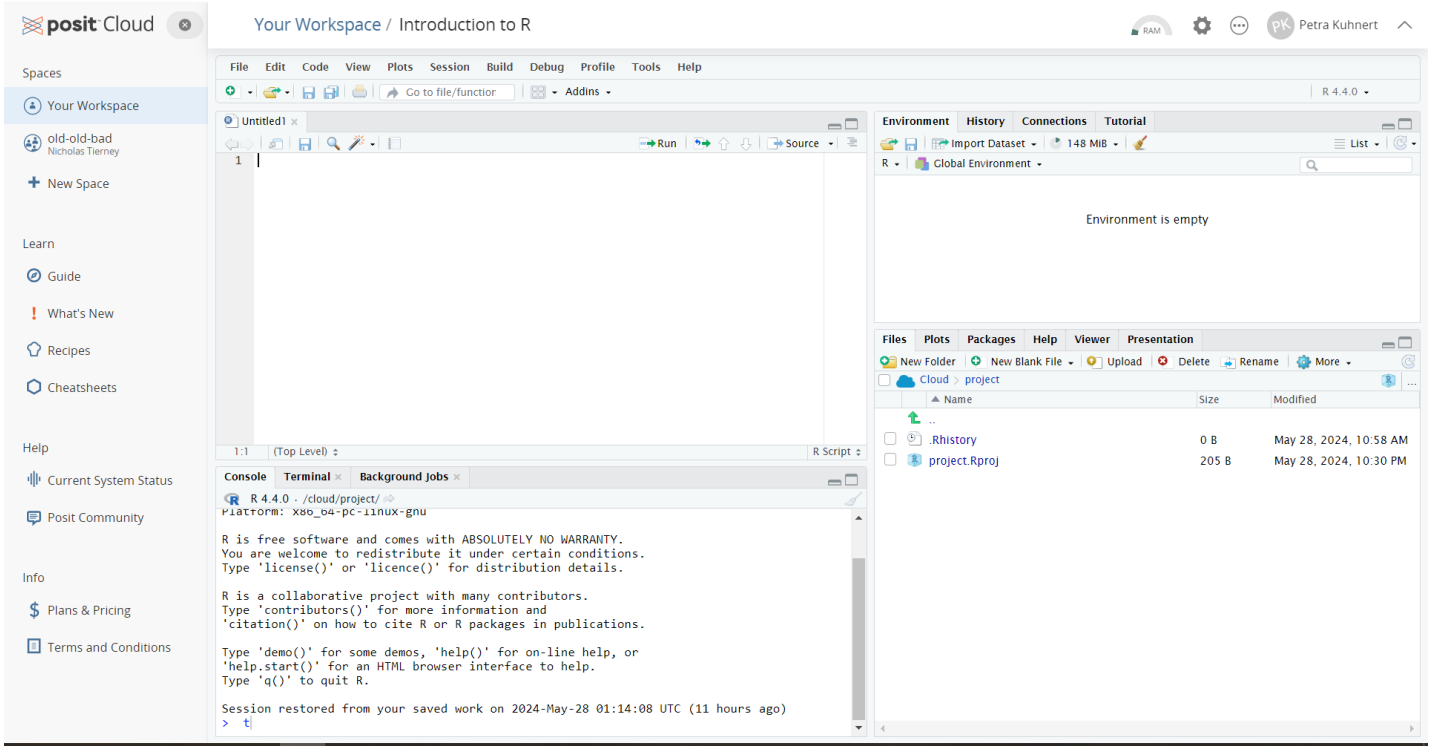


Modern day

- R is widely used in academia, research, and industry for statistical analysis, data visualization, and data science.
- The community actively contributes to the CRAN repository, adding new packages and tools.
- The R Core Team continues to manage the language's development, ensuring its stability and performance.

Running R in the browser

- Posit Cloud: https://posit.cloud/



Getting Help!

- Built-in help functions
  - `?`
  - `help()`
  - `help.search()`
- R Documentation
  - CRAN – Comrpehensive R Archive Network
  - `vignette()`
- Online help
  - Stack Overflow
  - Rstudio community
- Books and online tutorials
  - “R for Data Science” by Hadley Wickham

R Objects

Topics Covered in this Session

- Data objects
  - Vectors
    - creating vectors using c, scan, rep and seq
    - Numerical operations using vectors
  - Matrices
    - Creating matrices using dim, matrix, rbind and cbind
  - Data frames
    - Creating data frames using data.frame
  - Lists
    - Creating lists using the list function
    - Manipulating lists
- Indexing data objects

Data Objects in R

- Four most frequently used types of data objects
  - Vector
    - Set of elements of the same mode (logical, numeric: integer or double, complex, character (& lists)
  - Matrix
    - Set of elements appearing in rows and columns, where the elements are of the same mode (logical, numeric: integer or double, complex, character)
  - Data Frame
    - Similar to the Matrix object but columns can have different modes
  - List
    - Generalisation of a vector and represents a collection of data objects

Creating Vectors: c function

- To create a vector the simplest way is using the concatenation (c) function

```
> value.num <- c(3,4,2,6,20)
> value.char <- c("koala", "kangaroo", "echidna")
```

- For logical vectors TRUE and FALSE are logical values and T and F are variables with those values. So we can create a logical vector either by

```
> value.logical.1 <- c(F,F,T,T)
```

or

```
> value.logical.2 <- c(FALSE,FALSE,TRUE,TRUE)
```

Creating Vectors: `rep` and `seq` functions

- The rep function replicates elements of vectors

```
> value <- rep(5,6)
> value
[1]  5  5  5  5  5  5
```

- The seq function creates a regular sequence of values to form a vector

```
> seq(from=2,to=10,by=2)
[1]  2  4  6  8  10

> seq(from=2,to=10,length=5)
[1]  2  4  6  8  10
```

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

> seq(along=value)
[1]  1  2  3  4  5  6
```

Creating Vectors: `c`, `rep` and `seq`

- Vectors can also be created using a combination of these functions

```
> value <- c(1,3,4,rep(3,4), seq(from=1,to=6,by=2))
> value
[1]  1 3 4 3 3 3 3 1 3 5
```

- Note, elements of a vector are expected to be of the same mode. For example

```
> c(1:3,"a","b","c")
```

will produce an error message

Creating Vectors: `scan` function

- The scan function can be used to enter in data to vector format via the screen

```
> value <- scan()
1:  3  4  2  6  20
6:
> value
[1]  3  4  2  6  20
```

- Data can also be read in from files using this function but more about this later …

Basic Computation with Numerical Vectors

- Element-by-element operation
- Short vectors are ‘recycled’ to mach long ones

```
> x <- runif(10)
> x
 [1] 0.3565455 0.8021543 0.6338499 0.9511269
 [5] 0.9741948 0.1371202 0.2457823 0.7773790
 [9] 0.2524180 0.5636271
> y < 2*x + 1      # recycling short vectors
> y
[1] 1.713091 2.604309 2.267700 2.902254 2.948390
[6] 1.274240 1.491565 2.554758 1.504836 2.127254
```

Basic Computation with Numerical Vectors

- Some functions take vectors of values and produce results of the same length:
  - sin, cos, tan, asin, acos, atan, log, exp, …
- Some functions return a single value
  - sum, mean, max, min, prod, …

```
> z <- (x-mean(x))/sd(x)   # see also 'scale'
> z
[1] -0.69326707  0.75794573  0.20982940  1.24310440  1.31822981
[6] -1.40786896 -1.05398941  0.67726018 -1.03237897 -0.01886511
> mean(z)
[1] -1.488393e-16
> sd(z)
[1]  1
```

Creating Matrices: `dim` and `matrix` functions

- The dim function can be used to convert a vector to a matrix

```
> value <- rnorm(6)
> dim(value) <- c(2,3)
> value
           [,1]       [,2]       [,3]
[1,]  0.7093460 -0.8643547 -0.1093764
[2,] -0.3461981 -1.7348805  1.8176161
```

- This will fill the columns of the matrix. To convert back to a vector we simply use the dim function again.

```
> dim(value) <- NULL
```

Creating Matrices: `dim` and `matrix` functions

- Alternatively we can use the matrix function to convert a vector to a matrix

```
> matrix(value,2,3)
            [,1]       [,2]       [,3]
[1,]  0.7093460 -0.8643547 -0.1093764
[2,] -0.3461981 -1.7348805  1.8176161
```

- Or to fill by rows

```
> matrix(value,2,3,byrow=T)
             [,1]       [,2]       [,3]
[1,]  0.709346 -0.3461981 -0.8643547
[2,] -1.734881 -0.1093764  1.8176161
```

Creating Matrices: `rbind` and `cbind` functions

- To bind a row onto an already existing matrix, the rbind function can be used

```
> value <- matrix(rnorm(6),2,3,byrow=T)
> value2 <- rbind(value,c(1,1,2))
> value2
           [,1]       [,2]      [,3]
[1,]  0.5037181  0.2142138 0.3245778
[2,] -0.3206511 -0.4632307 0.2654400
[3,]  1.0000000  1.0000000 2.0000000
```

Creating Matrices: `rbind` and `cbind` functions

- To bind a column onto an already existing matrix, the cbind function can be used

```
> value3 <- cbind(value2,c(1,1,2))
                  [,1]       [,2]      [,3] [,4]
      [1,]  0.5037181  0.2142138 0.3245778    1
      [2,] -0.3206511 -0.4632307 0.2654400    1
      [3,]  1.0000000  1.0000000 2.0000000    2
```

Creating Dataframes: `data.frame` function

- The function data.frame converts a matrix or collection of vectors into a dataframe

```
> value3 <- data.frame(value3)
> value3
                  X1          X2        X3 X4
      1  0.5037181  0.2142138 0.3245778  1
      2 -0.3206511 -0.4632307 0.2654400  1
      3  1.0000000  1.0000000 2.0000000  2
```

- Another example

```
> value4 <- data.frame(rnorm(3),runif(3))
> value4
           rnorm.3.  runif.3.
      1 -0.6786953 0.8105632
      2 -1.4916136 0.6675202
      3  0.4686428 0.6593426
```

Specifying Row and Column Names

- The columns are automatically labelled

```
> names(value3)
[1] "X1" "X2" "X3" "X4"
```

- Alternative labels can be assigned

```
> names(value3) <- c("C1","C2","C3","C4")
```

- Row names are automatically assigned and are by default labelled "1", "2", "3", …

```
> row.names(value3)
[1] "1" "2" "3"
```

Specifying Row and Column Names

- These can also be renamed if desired

```
> row.names(value3) <- c("R1","R2","R3")
```

- Names can also be specified within the data.frame function

```
> data.frame(C1=rnorm(3),C2=runif(3),
      row.names=c("R1","R2","R3")
             C1        C2
R1 -0.2177390 0.8652764
R2  0.4142899 0.2224165
R3  1.8229383 0.5382999
```

Manipulating Data: An Example

- The iris dataset (`iris3`) is a 3-dimensional array (50x4x3), one dimension for each species: Setosa, Versicolor and Virginica
- Each species has the sepal lengths and widths and petal lengths and widths recorded
- To make this dataset more manageable, we can convert the 3-dimensional array into a 2-dimensional data frame

The Iris Data

```
> dimnames(iris3)
[[1]]
NULL

[[2]]
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."

[[3]]
[1] "Setosa"     "Versicolor" "Virginica"
```

The Iris Data

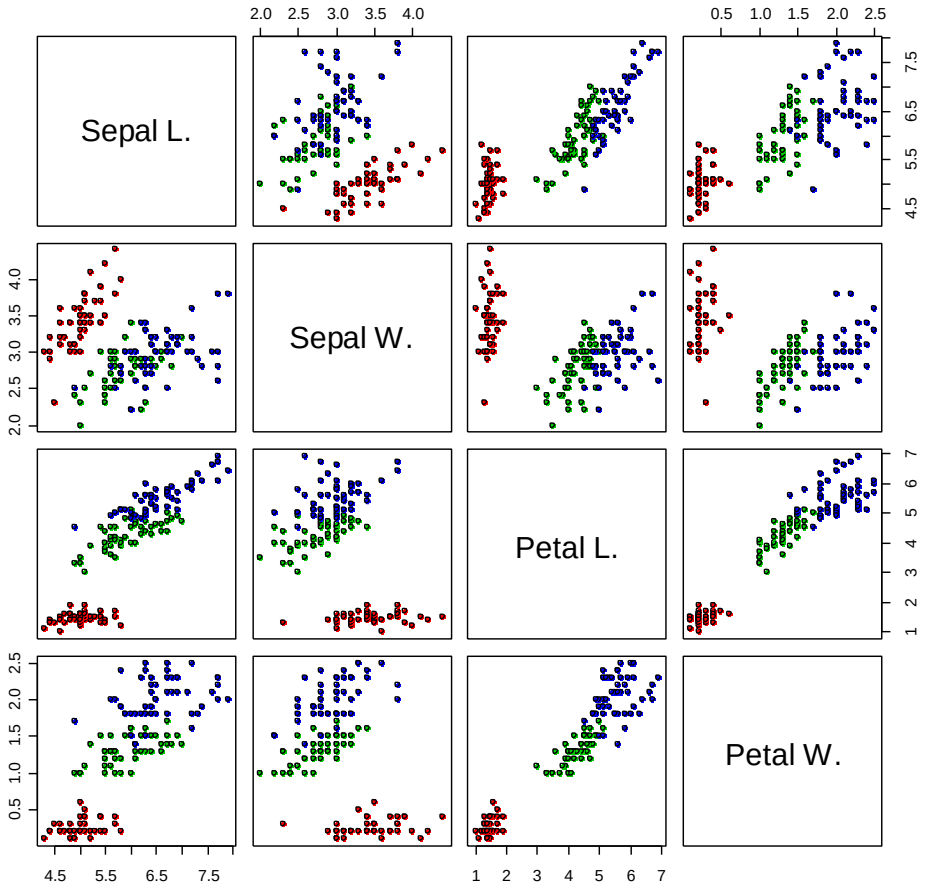
```
> Snames <- dimnames(iris3)[[3]]
# Convert into a 150 x 3 matrix
> iris.df <- rbind(iris3[,,1],iris3[,,2],iris3[,,3])
# Coerce the matrix into a data frame and check the
variable names
> iris.df <- as.data.frame(iris.df)
> names(iris.df)
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."
# Make a species factor and bind it to the columns of this
matrix
> iris.df$Species <- factor(rep(Snames,rep(50,3)))
```

- Lets have a look at the first 5 rows of the data frame

```
> iris.df[1:5,]
  Sepal L. Sepal W. Petal L. Petal W. 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
```

- Now produce a scatterplot of the data

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


Anderson's Iris Data
Sepal L.
Sepal W.
Petal L.
Petal W.
2.0 2.5 3.0 3.5 4.0
0.5 1.0 1.5 2.0 2.5
4.5 5.5 6.5 7.5
1 2 3 4 5 6 7

Accessing Elements of a Vector or Matrix

- Indexing may be done by
  - A vector of positive integers – to indicate inclusion
  - A vector of negative integers – to indicate exclusion
  - A vector of logical values – to indicate which are in and which are out
  - A vector of names – if the object has a names attribute
    - If a zero index occurs on the right, no element is selected; if a zero index occurs on the left, no assignment is made
    - An empty index position stands for “the lot”

Indexing Vectors

```
> x <- sample(1:5, 20, rep=T)
> x
 [1] 3 4 1 1 2 1 4 2 1 1 5 3 1 1 1 2 4 5 5 3
> x == 1
 [1] FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE TRUE
[11] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
> ones <- (x == 1)     # parentheses unnecessary
> x[ones] <- 0
> x
 [1] 3 4 0 0 2 0 4 2 0 0 5 3 0 0 0 2 4 5 5 3
```

Indexing Vectors

```
> others <- (x > 1)    # parentheses unnecessary
> y <- x[others]
> y
 [1] 3 4 2 4 2 5 3 2 4 5 5 3
> which(x > 1)
 [1]  1  2  5  7  8 11 12 16 17 18 19 20
```

Indexing Matrices and Dataframes: Indexing by Columns

```
> value3
           C1          C2        C3 C4
R1  0.5037181  0.2142138 0.3245778  1
R2 -0.3206511 -0.4632307 0.2654400  1
R3  1.0000000  1.0000000 2.0000000  2
> value3[, "C1"] <- 0
> value3
   C1          C2        C3 C4
R1  0  0.2142138 0.3245778  1
R2  0 -0.4632307 0.2654400  1
R3  0  1.0000000 2.0000000  2
```

Indexing Matrices and Dataframes: Indexing by Rows

```
> value3["R1", ] <- 0
> value3
   C1          C2         C3 C4
R1  0  0.0000000 0.0000000  0
R2  0 -0.4632307 0.2654400  1
R3  0  1.0000000 2.0000000  2
> value3[] <- 1:12
> value3
  C1 C2 C3  C4
R1 1  4  7  10
R2 2  5  8  11
R3 3  6  9  12
```

- Accessing the first two rows of the matrix/dataframe

```
> value3[1:2,]
   C1 C2 C3 C4
R1  1  4  7 10
R2  2  5  8 11
```

- Accessing the first two columns of the matrix/dataframe

```
> value3[,1:2]
   C1 C2
R1  1  4
R2  2  5
R3  3  6
```

- Obtaining any element with a value greater than 5

```
> as.vector(value3[value3>5])
[1]  6  7  8  9 10 11 12
```

Creating Lists: `list` function

- Lists can be created using the list function

```
> L1 <- list(x = sample(1:5, 20, rep=T),
           y = rep(letters[1:5], 4), z = rpois(20, 1))
> L1
$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
$y
 [1]  "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b"
 [13] "c" "d" "e" "a" "b" "c" "d" "e"
$z
 [1] 1 3 0 0 3 1 3 1 0 1 2 2 0 3 1 1 0 1 2 0
```

- Three equivalent ways of accessing the first component

```
> L1[["x"]]
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1[[1]]
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1

A sublist consisting of the first component only
> L1[1]
$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

Working with Lists

- The length of a list is equal to the number of components

```
> length(L1)
[1]  3
```

- The names of the list can be accessed using the names function and like dataframes can be altered

```
> names(L1) <- c("Item1","Item2","Item3")
```

- Indexing lists can be achieved in a similar way to how data frames are indexed

```
>  L1$Item1[L1$Item1>2]
 [1]  4  3  4  5  3  3  3  5  3  3  5
```

• Joining two lists can be done using the concatenation function, c

```
> L2 <- list(x = c(1,5,6,7), y =
c("apple","orange","melon","grapes"))
> c(L1,L2)
$Item1
 [1] 2 4 3 4 1 5 3 1 1 2 3 3 5 2 1 3 2 3 5 1
$Item2
 [1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b" [13]"c"
"d" "e" "a" "b" "c" "d" "e"
$Item3
 [1] 0 0 2 1 1 0 2 0 0 1 1 1 0 0 1 1 1 3 0 2
$x
[1] 1 5 6 7
$y
[1] "apple"  "orange" "melon"  "grapes"
```

- or the append function

```
> append(L1,L2,after=2)
$Item1
 [1] 2 4 3 4 1 5 3 1 1 2 3 3 5 2 1 3 2 3 5 1
$Item2
 [1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" [12]"b"
"c" "d" "e" "a" "b" "c" "d" "e"
$x
[1] 1 5 6 7
$y
[1] "apple"  "orange" "melon"  "grapes"
$Item3
 [1] 0 0 2 1 1 0 2 0 0 1 1 1 0 0 1 1 1 3 0 2
```

- Adding elements to a list can be achieved by

```
> L1$Item4 <- c("apple","orange","melon","grapes")
```

or by

```
> L1[[4]] <- c("apple","orange","melon", "grapes")
> names(L1)[4] <- c("Item4")
```

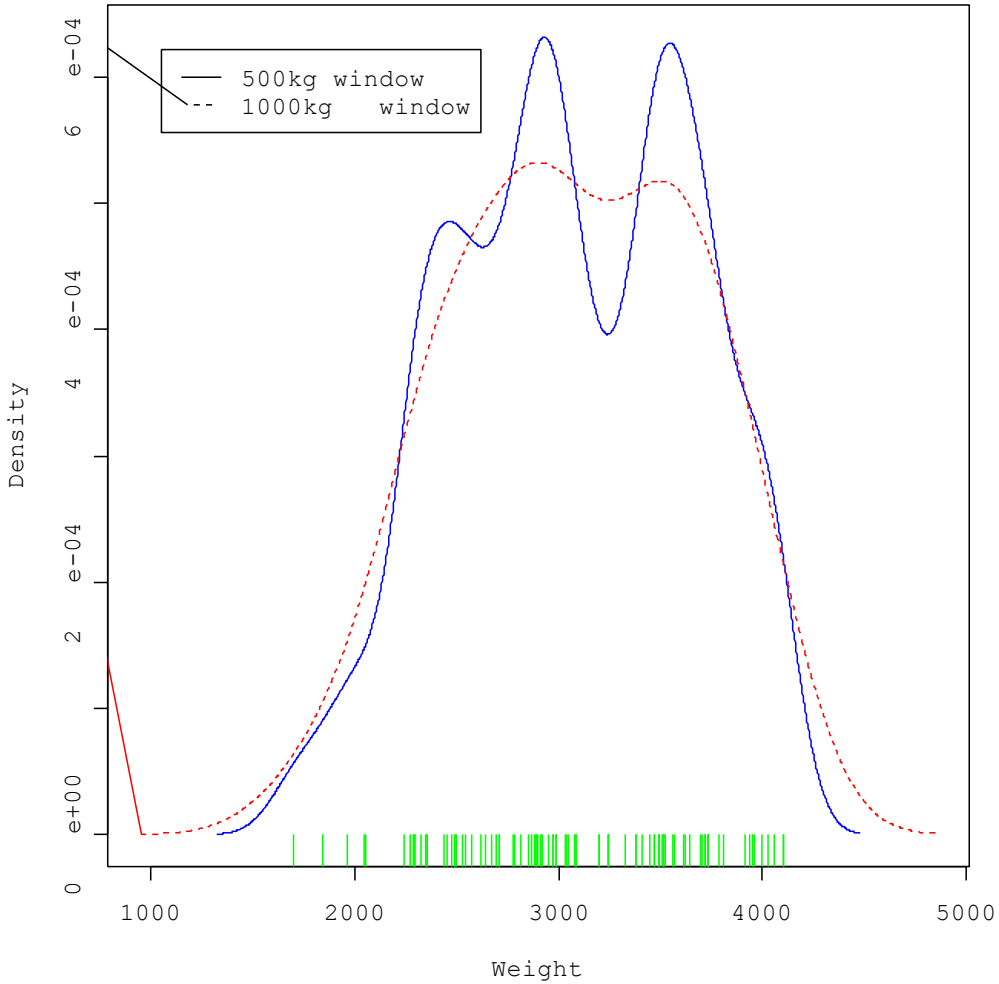
or by

```
> L1[["Item4"]] <- c("apple","orange","melon", "grapes")
```

- Other functions within R may produce a list as an output e.g. spline(), density(), and locator()

Comparison between 500kg and 1000kg Density Estimates for Weight: Cars93 Data

```
> attach(Cars93)
> dw5 <- spline(density(Weight, width=500))  # list
> dw10 <- spline(density(Weight,width=1000)) # list
> rx <- range(dw5$x,dw10$x)
> ry <- range(dw5$y,dw10$y)
> plot(dw5,type="n",xlim=rx,ylim=ry,cex=1.5,
      xlab="Weight",ylab="Density")
> lines(dw5,lty=1,col="blue")
> lines(dw10,lty=2,col="red")
> segments(Weight,pu[1],Weight,0,col="green")
> legend(locator(1),c("500kg window", "1000kg window"),lty=1:2)
> detach("Cars93")
```


500kg window
1000kg window
Density
0
e+00
2
e-04
4
e-04
6
e-04
1000
2000
3000
4000
5000
Weight

Grammar of graphics

Introduction to Grammar of Graphics

- Wickham (2016) ggplot2: elegant graphics for data analysis, Springer.
- The Grammar of Graphics is a systematic way to describe and build graphics.
- Implemented in ggplot2

- **Core Concepts:**
  - Data
  - Aesthetics
  - Geometries
  - Scales
  - Facets
  - Themes

Anatomy of ggplot

- **ggplot2 Basics:**
  - Starting a new plot with **ggplot()**.
  - Adding layers with **+**.
- **Example:**

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
color = Species)) +   geom_point()
```

- **Comparison with base R:**
  - Base R: **`plot(iris$Sepal.Length, iris$Sepal.Width)`**
  - ggplot2: More customizable and intuitive.

Displaying univariate data

- Histogram
  - Base R: `hist(iris$Sepal.Length)`
  - **ggplot2**

```
ggplot(iris, aes(x = Sepal.Length)) +
geom_histogram(binwidth = 0.5)
```

- Boxplots
  - `Base R: boxplot(iris$Sepal.Length)`
  - **ggplot2**
  - `ggplot(iris, aes(x = Species, y = Sepal.Length)) + geom_boxplot()`

Comparison of groups

- Comparative plots
  - Boxplots, histograms, density plots
  - Examples, comparing sepal.length across species

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) +  geom_boxplot()
```

Time series data

- Time series plots:
  - Example with hypothetical data
  - Base R: plot.ts(data)

```
ggplot(time_series_data, aes(x = Date, y =
Value)) +  geom_line()
```

Displaying Bivariate Data

- Scatterplots:
  - Base R: **plot(iris$Sepal.Length, iris$Sepal.Width)**
  - **Ggplot2:**

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +  geom_point()
```

  - **Adding Points and text:**
    - **Example adding text labels.**

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() +   geom_text(aes(label = Species), hjust = 1.5, vjust = 1.5)
```

Advanced plotting techniques

- Overlaying Figures
  - Adding multiple layers
  - **Example**

```
ggplot(data = volcano_data, aes(x = lon, y = lat)) +   geom_tile(aes(fill = elevation))
+   geom_contour(aes(z = elevation))
```

  - **Faceting**
    - **Splitting data into multiple panels**
    - **Example:**

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species)
```

Customising plots

- Themes and labels
  - Customising plot appearance)
  - **Example**

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() + labs(title = "Sepal Dimensions by
Species",  x = "Sepal Length", y = "Sepal Width") +
  theme_minimal()
```

Manipulating Objects in R (base)

Topics Covered in this Session

- Sorting
  - Sorting and ordering elements: **`sort`**, **`rank`** and **`order`**
- Dates and times
  - Manipulation of dates and conversion
- Summarising data
  - Tabulating and separating data: **`table`** and **`split`**
- Vectorised calculations
  - The **`apply`** family of functions

Sorting: `order` function

- Usually best done indirectly:
  - Find an index vector that achieves the sort operation and
  - Use it for all vectors that need to remain together
- '**`order`**' is a function that allows sorting with tie-breaking:
- Find an index vector that:
  - arranges the first of its arguments in increasing order,
  - ties are broken by the second argument,
  - any remaining ties are broken by a third argument

Sorting: An example using the `order` function

```
> x <- sample(1:5, 20, rep=T)
> y <- sample(1:5, 20, rep=T)
> z <- sample(1:5, 20, rep=T)
> xyz <- rbind(x, y, z)
> dimnames(xyz)[[2]] <- letters[1:20]
> xyz
  a b c d e f g h i j k l m n o p q r s t
x 4 4 2 4 3 4 4 1 2 2 5 3 1 5 5 3 4 5 3 4
y 5 5 2 5 2 3 5 4 4 2 4 2 1 4 3 4 4 2 2 2
z 4 5 3 2 4 2 4 5 5 2 4 2 4 5 3 4 3 4 4 3
```

```
> o <- order(x, y, z)
> xyz[, o]
  m h j c i l e s p t f q d a g b r o k n
x 1 1 2 2 2 3 3 3 3 4 4 4 4 4 4 4 5 5 5 5
y 1 4 2 2 4 2 2 2 4 2 3 4 5 5 5 5 2 3 4 4
z 4 5 2 3 5 2 4 4 4 3 2 3 2 4 4 5 4 3 4 5

> xyz          # reminder
  a b c d e f g h i j k l m n o p q r s t
x 4 4 2 4 3 4 4 1 2 2 5 3 1 5 5 3 4 5 3 4
y 5 5 2 5 2 3 5 4 4 2 4 2 1 4 3 4 4 2 2 2
z 4 5 3 2 4 2 4 5 5 2 4 2 4 5 3 4 3 4 4 3
```

Sorting: `sort` function

- The **`sort`** function can also be used to sort a vector or a list respectively into ascending or descending order

```
> sort(x)
[1] 1 1 2 2 2 3 3 3 3 4 4 4 4 4 4 4 5 5 5 5
> sort(x,decreasing=T)
[1] 5 5 5 5 4 4 4 4 4 4 4 3 3 3 3 2 2 2 1 1
```

- To sort a vector partially, use the partial argument

```
> sort(x,partial=c(3,4))
[1] 1 1 2 2 3 4 4 4 2 4 5 3 4 5 5 3 4 5 3 4
```

Sorting: `rank` function

- To rank the values in a vector, the rank function can be used. Ties result in ranks being averaged by default but other options are available: taking the first occurrence, randomly selecting a value or selecting the maximum or minimum value

```
> rank(x)
[1]  13.0 13.0 4.0 13.0  7.5 13.0 13.0  1.5 4.0  4.0 [11]
18.5  7.5 1.5 18.5 18.5  7.5 13.0 18.5 7.5 13.0
> rank(x, ties="first")          # first occurrence wins
[1] 4 4 2 4 3 4 4 1 2 2 5 3 1 5 5 3 4 5 3 4
> rank(x, ties="random")         # ties broken at random
[1] 16 15 5 14 9 12 11 1 3 4 19 6 2 17 18 8 13 20
[19] 7 10
> rank(x, ties="min")            # typical sports ranking
[1] 10 10 3 10 6 10 10 1 3 3 17 6 1 17 17 6 10 17
[19] 6 10
```

Dates and Times

- R has several mechanisms available for the representation of dates and times. The 'standard' one, however is the **`POSIXct/POSIXlt`** suite of functions and object possibilities
- objects of (old) class "**`POSIXct`**" are numeric vectors with each component the number of seconds since the start of 1970. Such objects are suitable for inclusion in data frames, for example.
- objects of (old) class "**`POSIXlt`**" are lists with the separate parts of the date/time held as separate components, plus a few.

Conversion from one form to another

- **`as.POSIXlt(obj)`** converts from **`POSIXct`** to **`POSIXlt`**
- **`as.POSIXct(obj)`** converts from **`POSIXlt`** to **`POSIXct`**
- **`strptime(char,form)`** generates **`POSIXlt`** objects from suitable character string vectors. You must specify the format used in the input character strings
- **`format(obj,form)`** generates character string vectors from **`POSIXlt`** or **`POSIXct`** objects. You must specify the format to be used in the output character strings
  `as.character(obj)` also generates character string vectors like format(,), but only to the ISO standard time/date format.
- For formatting details see, for example **`?strptime`**

On what day of the week were you born and for how many seconds have you lived?

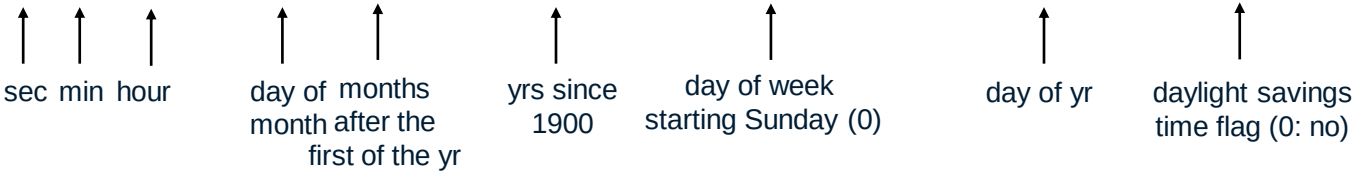
```
> myBday <- strptime("18-Apr-1973", "%d-%b-%Y")
> class(myBday)
[1] "POSIXt"  "POSIXlt"
> myBday
[1] "1973-04-18"
> weekdays(myBday)
[1] "Wednesday"

> Sys.time()
[1] "2005-01-19 12:08:12 E. Australia Standard Time"
> Sys.time() - myBday
Time difference of 11599.51 days
```

Arithmetic on `POSIXt` objects

- Some arithmetic operations are allowed on date/time objects (POSIXlt or POSIXct). These are
  - **`obj + number`**
  - **`obj – number`**
  - **`obj1 <lop> obj2`**
  - **`obj1 – obj2`**
- In the first two cases, '**`number`**' is a number of seconds and each date is augmented by this number of seconds. If you wish to augment by days you need to work with multiples of **`60*60*24`**.
- In the second case '**`<lop>`**' is a logical operator and the result is a logical vector
- In the third case the result is a '**`difftime`**' object, represented as a number of seconds 'time difference'.

Birthday example continued …

```
> as.numeric(Sys.time())
[1] 1106100492
> as.numeric(myBday)
[1]   0  0   0    18    3      73         3          107        0
```




```
> as.numeric(as.POSIXct(myBday))
[1] 103903200
> as.numeric(Sys.time()) - as.numeric(as.POSIXct(myBday))
[1] 1002197292
```

The `table` function

- The **`table`** function can be used to tabulate factors or find frequencies of objects.
- The quine dataset consists of 146 rows describing children's ethnicity (**`Eth`**), age (**`Age`**), sex (**`Sex`**), days absent from school (**`Days`**) and their learning ability (**`Lrn`**). Eth, Sex, Age and Lrn are factors. Days is a numeric vector.
- If we want to find out the age classes in the **`quine`** dataset

```
> attach(quine)
> table(Age)
Age
F0 F1 F2 F3   # F0: primary, F1-F3: forms 1-3
27 46 40 33
```

- If we need to know the breakdown of ages according to sex

```
>  table(Sex,Age)
   Age
Sex F0 F1 F2 F3
      F 10 32 19 19
      M 17 14 21 14
```

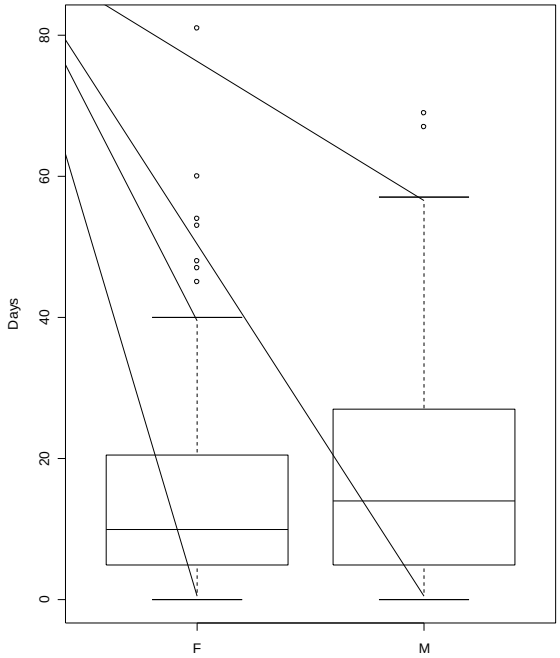
The `split` function

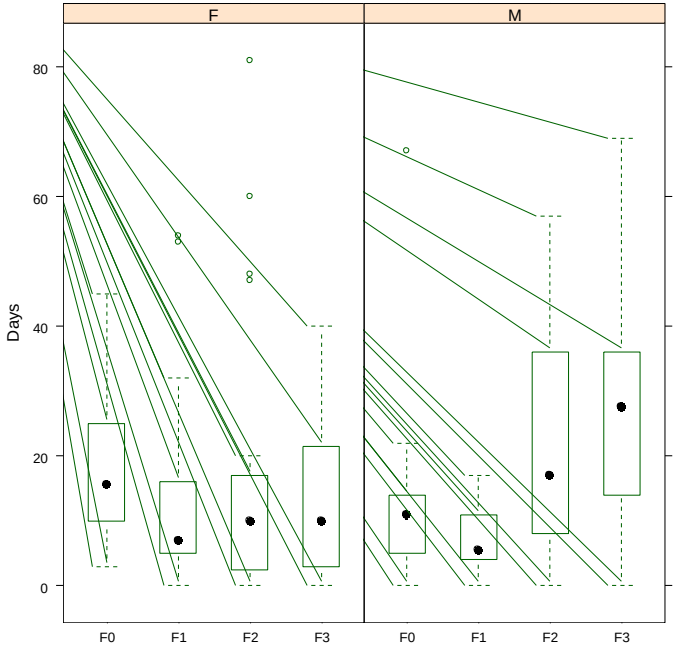
- The split function divides the data specified by vector **`x`** into the groups defined by factor **`f`**
- This function can be useful for graphical displays of data
- If we want to obtain a summary of **`Days`** split by **`Sex`**

```
> split(Days,Sex)
$F
 [1]  3  5 11 24 45  5  6  6  9 13 23 25 32 53 54  5  5 11 17
[20] 19  8 13 14 20 47 48 60 81  2  0  2  3  5 10 14 21 36 40
[39] 25 10 11 20 33  5  7  0  1  5  5  5  5  7 11 15  5 14  6
[58]  6  7 28  0  5 14  2  2  3  8 10 12  1  1  9 22  3  3  5
[77] 15 18 22 37

$M
 [1]  2 11 14  5  5 13 20 22  6  6 15  7 14  6 32 53 57 14 16
[20] 16 17 40 43 46  8 23 23 28 34 36 38  6 17 67  0  0  2  7
[39] 11 12  0  0  5  5  5 11 17  3  4 22 30 36  8  0  1  5  7
[58] 16 27  0 30 10 14 27 41 69
```

or for some nice graphical displays

```
> boxplot(split(Days,Sex),ylab="Days Absent")
> library(lattice)                          # trellis graphics
> trellis.par.set(col.whitebg())
> bwplot(Days ~ Age | Sex)  # implicit split
```





with, subset and transform Functions

- These functions operate on an object or elements within an object
- No attachment necessary
  - with: evaluates expressions constructed from the data

```
> with(Cars93,plot(Weight,100/MPG.highway))
```

  - subset: returns subsets of vectors or data frames that meet specific requirements

```
> Vans <- subset(Cars93,Type=="Van")
```

  - transform: transforms elements of an object

```
> Cars93T <- transform(Cars93,
                       WeightT=Weight/1000)
```

Vectorised Calculations

- Those from a programming background may be used to operating on individual elements of a vector. However, R has the ability to perform vectorised calculations. This allows you to perform calculations on an entire vector/matrix/dataframe/list instead of the individual elements.
- Not all problems can be vectorised but most can once you know how.

The ‘apply’ family

- Four members: **lapply**, **sapply**, **tapply**, **apply**
  - **lapply**: takes any structure, gives a list of results
  - **sapply**: like lapply, but ‘simplifies’ the result if possible
  - **apply**: only used for arrays
  - **tapply**: used for ‘ragged arrays’: vectors with an indexing specified by one or more factors.
- Used for

  a)efficiency relative to explicit loops and

  b)convenience

The `apply` function

- This function allows functions to operate on successive sections of an array
  - Compute the mean of each column of the iris data

```
> iris[1:4,]
  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
> apply(iris[,-5],2,mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

The `tapply` function

- Ragged arrays represent a combination of a vector and a labelling factor or factors, where the group sizes are irregular
- To apply functions to ragged arrays the **`tapply`** function is used, where the object, list of factors and function is supplied
- We will illustrate this function using the quine dataset again

```
> quine[1:5,]
  Eth Sex Age Lrn Days
1   A   M  F0  SL    2
2   A   M  F0  SL   11
3   A   M  F0  SL   14
4   A   M  F0  AL    5
5   A   M  F0  AL    5
```

The `tapply` function continued …

- To calculate the average number of days absent for each age class we can use the tapply function

```
> tapply(Days,Age,mean)
       F0        F1        F2        F3
14.85185 11.15217 21.05000 19.60606
```

- To perform this calculation for each gender we need to specify using the **`list`** function two factors: Sex and Age

```
> tapply(Days,list(Sex,Age),mean)
         F0        F1        F2        F3
F 18.70000 12.96875 18.42105 14.00000
M 12.58824  7.00000 23.42857 27.21429
```

Operating along structures: `lapply` and `sapply` functions

- **`lapply`** and sapply operate on components of a list or vector
- **`lapply`** will always return a list
- **`sapply`** is a more user friendly version of lappy and will attempt to simplify the result into a vector or array

```
# Example 1: lapply
> l <- list(Sex=Sex,Eth=Eth)
> lapply(l,table)
$Sex
F  M
80 66

$Eth
A  N
69 77
```

```
# Example 2: sapply
> l <- list(Sex=Sex,Eth=Eth)
> sapply(l,table)
   Sex  Eth
F   80   69
M   66   77
```

The 'apply' functions versus conventional programming

- Programming methods will be discussed in more detail in a later session but … how do the family of **apply** functions compare with conventional programming methods
- We can test the performance of these functions using the **system.time** function
- Lets compare …
  - PROBLEM: We wish to subtract the mean from each element in a 25000 x 4 matrix

```
> mat <- matrix(rnorm(100000),ncol=4)
```

Program 1: The programmer's approach

5 `for` loops

```
program1 <- function(mat){
  col.scale <- matrix(NA,nrow(mat),ncol(mat))# create an empty matrix
  m <- rep(0,ncol(mat))                      # create a vector, m and
                                             #   set it to zero
  for(j in 1:ncol(mat)){
      for(i in 1:nrow(mat)) {
          m[j] <- m[j] + mat[i,j]            # compute the sum of
                                             #   elements in each
                                             #   column
      }
  }
  for(j in 1:ncol(mat))
      m[j] <- m[j]/nrow(mat)                 # compute the mean
  for(i in 1:nrow(mat)){
      for(j in 1:ncol(mat)){
          col.scale[i,j] <- mat[i,j]-m[j]    # centre each column by
                                             #  the mean
      }
   }
  col.scale                                  # print the scaled matrix
}
```

Program 2: The programmer's approach (using a built-in function)

3 `for` loops

```
program2 <- function(mat){
  col.scale <- matrix(NA,nrow(mat),ncol(mat))# create an empty matrix
  m <- NULL                                                  # initialise
   the vector,m
  for(j in 1:ncol(mat))
      m[j] <- mean(mat[,j])                        # compute the mean of
                                                             #    each
   column
  for(i in 1:nrow(mat)){
      for(j in 1:ncol(mat)){
          col.scale[i,j] <- mat[i,j]-m[j]      # centre columns
      }
  }
  col.scale                                                  # print the
   scaled matrix
}
```

Program 3: R programming approach

No `for` loops

```
program3 <- function(mat){
    apply(mat,2,scale,scale=F)   # apply
  to the matrix (mat)
                                 #    the
  function scale
                                 #
  specifying that
                                 #
  centring only be
                                 #
  performed.
                                 # print the
  scaled matrix
}
```

How Does Each Program Perform?

- **system.time** function -> (1) user CPU, (2) system CPU, (3) Elapsed time, (4) subprocessor 1 time and (5) subprocessor 2 time
- Only the first three are of real interest

```
> system.time(v1 <- program1(mat)) # Slowest
[1]  2.33 0.04 2.37
> system.time(v2 <- program2(mat)) # 3.5x increase in CPU
[1]  0.68 0.00 0.72
> system.time(v3 <- program3(mat))  # 78x increase in CPU
[1]  0.03 0.00 0.03
> system.time(v4 <- scale(mat,scale=F)) # 233x increase in CPU
[1]  0.01 0.00 0.01
> check <- function(v1,v2) abs(diff(range(v1-v2))) # check
> check(v1,v2) + check(v2,v3) + check(v3,v4)
[1] 4.440892e-16
```

Which approach would you use?

R packages for data science

R packages for data science

- Install.packages("tidyverse")
- All packages take on the philosophy of the "grammar of graphics"
  - **ggplot2**: creating graphics
  - **dplyr:** data manipulation
  - **forcats**: manipulating factors
  - **tibble**: data frame structure that has better properties
  - **readr**: fast and friendly way to read in data
  - **stringr**: string manipulation
  - **tidyr**: creating tidy data
  - **purrr**: enhancing functional programming

Overview of `dplyr`

- Key functions
  - `filter()`: for subsetting rows based on conditions
  - `arrange()`: for reordering rows
  - `select()`: for selecting columns
  - `mutate()`: for creating new columns or modifying existing ones
  - `summarise()`: for summarising multiple values to a single value
- Working with pipes `%>%`

```
library(tidyverse)

data <- starwars %>%
        select(name, height, mass, gender) %>%
        filter(!is.na(mass), mass > 100) %>%
        arrange(desc(mass))
```

Summarising data

- Use of `group_by()` and `summarise()` to compute summaries

```
summary <- data %>%
           group_by(gender) %>%
           summarise(
             average_height = mean(height, na.rm = TRUE),
             average_mass = mean(mass, na.rm = TRUE)
           )
```

Manipulating dates and times with `dplyr`

- `lubridate` is a package that is part of tidyverse
- Aimed at making working with dates/times much easier than base.
- Key functions:
  - `ymd()`,`mdy()`,`dmy()`: functions for creating date objects from character strings.
  - `year()`,`month()`,`day()`,`hour()`,`minute()`,`second()`: functions to extract components from date-time objects

Manipulating dates: an example

```
library(lubridate); library(dplyr)

# Example data frame
data <- tibble(
  timestamp = c("2021-06-01 12:01:00", "2021-06-02 15:30:00", "2021-06-03
08:45:00"),
  value = c(10, 15, 20)
)

# Convert character to POSIXct date-time object and extract parts
data <- data %>%
  mutate(
    date_time = ymd_hms(timestamp),
    date = date(date_time),
    time = format(date_time, "%H:%M:%S"),
    month = month(date_time, label = TRUE),
    day = day(date_time)
  ) %>%
  select(-timestamp, -date_time)  # Clean up by removing the original timestamp
columns
```

Manipulating dates: an example

```
# Summarize data by month
monthly_summary <- data %>%
  group_by(month) %>%
  summarise(
    average_value = mean(value),
    total_value = sum(value),
    .groups = 'drop'  # Drop grouping structure after summarisation
  )
```

- The **`ymd_hms()`** function converts strings to POSIXct date-time objects
- **`mutate()`** is used to extract the date, time, month and day
- dataset is grouped by month using **`group_by()`** and summarised with **`summarise()`** to find average and total values.

Working with tibbles in R

- A different, more efficient way to work with data frames.
- Designed to print better at the console than traditional data frames
- Useful where datasets are complex and/or large.

```
library(tibble)

# Creating a simple tibble
data <- tibble(
  x = 1:5,
  y = c("one", "two", "three", "four", "five"),
  z = c(TRUE, TRUE, FALSE, TRUE, FALSE)
)
data
```

Tidying data with `tidyr`

- Aims to create “tidy” data where each column is a variable and each row is an observation.
- Some key functions:
  - **`pivot_wider()`, `pivot_longer()`:**
  - **`separate()`:** splits a single column into multiple columns based on a delimiter
  - **`unite()`:** joins multiple columns
  - **`fill()`; `replace_na()`:** fills in missing values
  - **`expand()`; `crossing()`; `complete()`:** creating new rows and columns of data

```
data %>%
  pivot_wider(names_from = y, values_from = z)
```

Simplifying factor management with `forcats`

- A tidverse package designed to handle factor variables more efficiently.

```
library(forcats)
library(dplyr)

# Example data
data <- tibble(
  category = factor(c("apple", "banana", "apple", "banana", "cherry"))
)

# Reorder factors by frequency
data %>%
  mutate(category = fct_infreq(category))
```

Functional programming with **purrr**

- More efficient and streamlined way of working with functions and vectors
- Key functions:
  - **map(), reduce(), pluck(), chuck(), keep(), discard()**
  - **safely(), quietly(), possibly(), at_depth()**

```
library(purrr)

# Example of using map to apply a
function
data <- tibble(x = 1:5)
data %>%
  mutate(squared = map_dbl(x, ~ .x^2))
```

Functional programming with `stringr`

- Simplifies the process of working with strings
- Key functions:
  - `str_detect()`, `str_replace()`, `str_length()`, `str_c()`
  - `str_split()`, `str_sub()`, `str_trim`, `str_squish()`

```
library(stringr)

# Example string manipulation
text <- "Hello, World!"
str_to_upper(text)
```

Getting stuff into and out of R

- Getting stuff in
  - Low level functions: **`scan`**
  - Functions for small-midsized datasets: **`read.table`**, **`read.csv`**
  - **`tidyverse: readr: read_csv`**
- Getting stuff out
  - Low level functions: **`cat`**, **`print`** and **`sink`**
  - Functions for small-midsized datasets: **`write.table`**, **`write`**
  - Exporting large data files: **`write.matrix`**
  - **`tidyverse" readr: write_csv`**

Getting stuff into and out of R using

- Getting stuff in

```
library(readr)

# Reading a CSV file
data <-
read_csv("path/to/your/data.csv")
```

- Getting stuff out

```
library(readr)

# Assuming 'data' is a data frame
write_csv(data,
"path/to/your/file.csv")
```

Where to find out more

- " R for Data Science" by Hadley Wickham and Garrett Grolemund
- " ggplot2: Elegant graphics for data analysis " by Hadley Wickham
- CRAN: Comprehensive R Network
- R bloggers: https://www.r-bloggers.com/

- **Swirl**: An R package that teaches R programming and data science interactively within the R console itself. It's a great way to learn by doing.

```
install.packages("swirl")
library(swirl)
swirl()
```

Some Homework

Exploring the mtcars dataset

- `mtcars` is a preloaded dataset in the R base package. You should be able to access the dataset by typing it into the console.
  - Find out about the dataset by getting some help up on the screen.
  - How many columns and rows are represented by the data?
  - What is the average weight of cars in this dataset?
  - Weights in this dataset are recorded in lbs. If you divide the weight in pounds by 453.592 you can get the equivalent reading in grams. Create a new R object that takes on the `mtcars` data and add a column that represents the weight in grams.

Exploring the mtcars dataset

- Produce a histogram of the weight column using the hist base plotting command. Then find an equivalent way to do this using the `ggplot2` package. (Hint: look up `geom_histogram` for more information)
- How might you explore whether there is a relationship between the displacement of a vehicle and its relationship to miles per gallon?
- Which vehicles have 8 cylinders? Can you produce a graphic that summarises the vehicles mpg by number of cylinders? (Hint: explore `geom_boxplot`)
- Explore the help on `geom_boxplot` and see how you might be able to make a more visually appealing figure. Try out different options.