Data Handling in R

```
# Rename variables
str_replace("cEV1_GSV", "EVT")
str_replace("cAge_1", "Age")
str_replace_all(";", ";times;")
# Italicize variable names, except for blank ones
str_replace("<em></em>", "")
# Complete the subscript
paste_after(times) %>% subscript %>%
paste_after(emphasize("$gamma;"))
gammas
```

Bernd Klaus, EMBL
Basic data handling: subsetting and extraction

- Density determined from underwater weighing
- Percent body fat from Siri’s (1956) equation
- Age (years)
- Weight (lbs)
- Height (inches)
- Neck circumference (cm)
- Chest circumference (cm)
- Abdomen 2 circumference (cm)
- Hip circumference (cm)
- Thigh circumference (cm)
- Knee circumference (cm)
- Ankle circumference (cm)
- Biceps (extended) circumference (cm)
- Forearm circumference (cm)
- Wrist circumference (cm)
Reminder: boolean operators

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>(a \mid \neg b)</th>
<th>(a &amp; b)</th>
<th>xor(a, b)</th>
</tr>
</thead>
</table>

\[
x > 1 \\
x \geq 1 \\
x < 1 \\
x \leq 1 \\
x \neq 1 \\
x == 1 \\
x \%in\% ("a", "b")
\]
### Extracting and Subsetting

Extract lines from data tables using boolean operators

```r
## all samples with age between 40 and 60 and height
## between 50 and 65
bodyfat[, bodyfat$age > 40 & bodyfat$age < 60 & bodyfat$height > 50 & bodyfat$height < 65,

#> density percent.fat age weight height neck.circum chest.circum abdomen.circum hip.circum
#> 74 1.068 13.5 55 125 64 33.2 87.7 76 88.6
#> 216 0.995 47.5 51 219 64 41.2 119.8 122 112.8

#> thigh.circum knee.circum ankle.circum bicep.circum forearm.circum wrist.circum
#> 74 50.9 35.4 19.1 29.3 25.7 16.9
#> 216 62.5 36.9 23.6 34.7 29.1 18.4

## get the corresponding indices
which(bodyfat$age > 40 & bodyfat$age < 60 & bodyfat$height > 50 & bodyfat$height < 65)

#> [1]  74  216

## and samples
bodyfat[, which(bodyfat$age > 40 & bodyfat$age < 60 & bodyfat$height > 50 & bodyfat$height < 65),

#> density percent.fat age weight height neck.circum chest.circum abdomen.circum hip.circum
#> 74 1.068 13.5 55 125 64 33.2 87.7 76 88.6
```
Caveat: selections via "which"

- Often you only want to data with special characteristics, e.g. all men older than 40
- using a logical expression such as `variable == value`, e.g. `(var1 < 5) & (var2 > 10)` you can get the respective samples with: `X[Condition]` or `subset(X, Condition)`
- The `which(Condition)` command returns the respective indices
- Alternatively, you can access the respective data entries with `X[which(Condition)]`
- However, if the condition is always evaluated as FALSE, `X[!which(Condition)]` is not the same as `X[!Condition]` since `which(Condition)` is an empty vector in this case ...
- => Use `X[!Condition]` rather than `X[!which(Condition)]`!
Another example: Irises

Iris virginica

Iris setosa

Iris versicolor
which and subset examples

the following commands give all iris having a sepal.width greater than 4

\[
\text{iris[which(iris$Sepal.Width > 4), ], or subset(iris, Sepal.Width > 4) or iris[Sepal.Width > 4,]}
\]

the output is:

<table>
<thead>
<tr>
<th>Sepal.Length</th>
<th>Sepal.Width</th>
<th>Petal.Length</th>
<th>Petal.Width</th>
<th>Species</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5.7</td>
<td>4.4</td>
<td>1.5</td>
<td>0.4</td>
</tr>
<tr>
<td>33</td>
<td>5.2</td>
<td>4.1</td>
<td>1.5</td>
<td>0.1</td>
</tr>
<tr>
<td>34</td>
<td>5.5</td>
<td>4.2</td>
<td>1.4</td>
<td>0.2</td>
</tr>
</tbody>
</table>
### Reading Data

Easy with RStudio: Tools -> Import

Example file: **Patients.csv** (NA is a marker for a quantity that is not available)

<table>
<thead>
<tr>
<th></th>
<th>Height,</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1,</td>
<td>1.65,</td>
<td>80.0</td>
</tr>
<tr>
<td>P2,</td>
<td>1.30,</td>
<td>NA</td>
</tr>
<tr>
<td>P3,</td>
<td>1.20,</td>
<td>50.0</td>
</tr>
</tbody>
</table>

R-Code: `patients <- read.csv(file = 
/path/to/filename/Patients.csv, header = TRUE)`

Excel read/write via the package **xlsx**, `read.xlsx` / `write.xlsx`
More complex data structures

When the data to be stored is more complex, special objects are created to store and handle it.

Higher order objects are constructed using simple data types we have seen so far as building blocks.

As an example, we look at how Microarray data is handled in Bioconductor.
More complex data structures

When the data to be stored is more complex, special objects are created to store and handle it. Higher order objects are constructed using simple data types we have seen so far as building blocks. As an example, we look at how Microarray data is handled in Bioconductor.

```r
> data(sample.ExpressionSet)
> sample.ExpressionSet
ExpressionSet (storageMode: lockedEnvironment)
  assayData: 500 features, 26 samples
    element names: exprs, se.exprs
  protocolData: none
  phenoData
    sampleNames: A B ... Z (26 total)
    varLabels: sex type score
    varMetadata: labelDescription
  featureData: none
  experimentData: use 'experimentData(object)'
Annotation: hgu95av2
```
Microarray data

• Objects are stored in different environments = unordered sets of objects
• Specialized functions are available to extract the data, e.g. `exprs(sample.ExpressionSet)` to get the expression measurements
• Available slots of the object can be retrieved by `slotNames(sample.ExpressionSet)`
• These include `assayData` (expression values) `phenoData` (sample discriptions, e.g. experimental conditions and `featureData` (Probeset / gene annotations e.g. ENSEMBL gene IDs)
Microarray data

- Objects are stored in different environments as unordered sets of objects.
- Specialized functions are available to extract the data, e.g., `exprs(sample.ExpressionSet)` to get the expression measurements.
- Available slots of the object can be retrieved by `slotNames(sample.ExpressionSet)`.
- These include `assayData` (expression values), `phenoData` (sample descriptions, e.g., experimental conditions), and `featureData` (Probeset / gene annotations, e.g., ENSEMBL gene IDs).
The `dplyr` verbs

The package `dplyr` provides a “grammar” of data manipulation

It includes “verbs” that provide basic functionality

structure:

- first argument is a data frame
- return value is a data frame
- nothing is modified in place
The *dplyr* verbs

- first argument is a data frame
- return value is a data frame
- nothing is modified in place

- Select rows with `filter()`
- Arrange rows with `arrange()`
- Select columns with `select()`
- Add columns with `mutate()`

useful helper functions for viewing data frames:

`glimpse()` nice summary,
`tbl_df()` compact viewing,
`sample_n()` select random sample of rows
Example data frame

def <- data.frame(
    color = c("blue", "black", "blue", "blue", "black"),
    value = 1:5)

filter(df, color == "blue")
```r
filter(df, value %in% c(1, 4))
```
The function `arrange()` is used to sort the `df` DataFrame by the `color` column.

**Before Sorting:**
```
<table>
<thead>
<tr>
<th>color</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
```

**After Sorting:**
```
<table>
<thead>
<tr>
<th>color</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>
```
arrange(df, desc(color))
```python
select(df, color)
```
mutate() df

<table>
<thead>
<tr>
<th>color</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
</tr>
<tr>
<td>black</td>
<td>2</td>
</tr>
<tr>
<td>blue</td>
<td>3</td>
</tr>
<tr>
<td>blue</td>
<td>4</td>
</tr>
<tr>
<td>black</td>
<td>5</td>
</tr>
</tbody>
</table>

mutate(df, double = 2 * value)

<table>
<thead>
<tr>
<th>color</th>
<th>value</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>black</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>blue</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>blue</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>black</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
mutate() - 2

**df**

<table>
<thead>
<tr>
<th>color</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
</tr>
<tr>
<td>black</td>
<td>2</td>
</tr>
<tr>
<td>blue</td>
<td>3</td>
</tr>
<tr>
<td>blue</td>
<td>4</td>
</tr>
<tr>
<td>black</td>
<td>5</td>
</tr>
</tbody>
</table>

→

<table>
<thead>
<tr>
<th>color</th>
<th>value</th>
<th>double</th>
<th>quadruple</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>black</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>blue</td>
<td>3</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>blue</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>black</td>
<td>5</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

**mutate(df, double = 2 * value, quadruple = 2 * double)**
Data viewing subsetting with dplyr

using filter and select

```r
select(filter(iris, Sepal.Width > 4), -Species)
```

```
Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
1   5.7          4.4          1.5          0.4
2   5.2          4.1          1.5          0.1
3   5.5          4.2          1.4          0.2
```

Looking at the data with `glimpse`

```r
> glimpse(iris)
```

Variables:

- `Sepal.Length` (dbl) 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4,...
- `Sepal.Width` (dbl) 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7,...
- `Petal.Length` (dbl) 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5,...
- `Petal.Width` (dbl) 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2,...
- `Species` (fctr) setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa,
The split-apply-combine paradigm

The ideas behind apply functions can be extended to a split-apply-combine paradigm

In data analysis problems you very often need to ...

**split up** a big data structure into homogeneous pieces

**apply** a function to each piece and then

**combine** all the results back together

this saves A LOT of for loops and typing and is implemented in the R package *plyr*, and in its successor *dplyr*

**See:** The Split-Apply-Combine Strategy for Data Analysis, Wickham, Journal of Statistical Software, 2010
Example with toy data

<table>
<thead>
<tr>
<th>color</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
</tr>
<tr>
<td>black</td>
<td>2</td>
</tr>
<tr>
<td>blue</td>
<td>3</td>
</tr>
<tr>
<td>blue</td>
<td>4</td>
</tr>
<tr>
<td>black</td>
<td>5</td>
</tr>
</tbody>
</table>

by_color <- group_by(df, color)
summarise(by_color, total = sum(value))
Example of the strategy with the Iris data

looking again at the iris data, a question of interest might be the computation of species wise statistics i.e. the following workflow:

1.) split/group data by species

2.) “apply” summary (e.g. mean)

3.) combine results in a new data frame
The pipe operator `%>%%

dplyr offers a chaining operator (read: “in”)

\[ x \%>\% f(y) = = f(x,y) \]

This allows nicely readable code

\[
\text{iris} \%>\%
\text{group\_by}(\text{Species}) \%>\%
\text{summarize}(\text{mean.s}l = \text{mean}(\text{Sepal.Length}) )
\]

Result:

<table>
<thead>
<tr>
<th>Species</th>
<th>mean.s</th>
</tr>
</thead>
<tbody>
<tr>
<td>setosa</td>
<td>5.006</td>
</tr>
<tr>
<td>versicolor</td>
<td>5.936</td>
</tr>
<tr>
<td>virginica</td>
<td>6.588</td>
</tr>
</tbody>
</table>
Simple chaining example

create two vectors and calculate Euclidian distance between them

```r
x1 <- 1:5; x2 <- 2:6

usual way
sqrt(sum((x1-x2)^2))
#> [1] 2.24

chaining method
(x1-x2)^2 %>%
sum() %>%
sqrt()
#> [1] 2.24
```
split - apply - combine

```r
iris %>%
group_by(Species) %>% grouping step
summarize( mean.sl = mean(Sepal.Length) )
   applying and summarizing

grouping step: groups the data frame iris by the column species
apply step: creates a new data frame with mean.sl contain the sepal length per species
combine step: summarize computes transformations of columns and turns the result into a new data frame
```
Tidy data

A lot of analysis time is spent on the process of cleaning and preparing the data.

Data preparation is not just a first step, but must be repeated many over the course of analysis.

data tidying: structuring datasets to facilitate analysis.
Tidy datasets are all alike but every messy dataset is messy in its own way.

Tidy datasets provide a standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning).

**Definitions:**

A dataset is a collection of values,

Values are organized in two ways: Every value belongs to a variable and an observation.

A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units.

An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.
Example from the tidyverse pkg

Untidy data

```r
> preg
name      treatmenta treatmentb
John Smith  NA         182
Jane Doe    4          13
Mary Johnson 6          7
```

variable value appears in column names
this leads to collapsing of two observations in only one row
Some code leads to a tidy version of the data, with each column representing a variable and each row an observation.

```r
library(tidyrr)
library(dplyr)
preg2 <- preg %>%
  gather(treatment, n, treatmenta:treatmentb) %>%
  mutate(treatment = gsub("treatment", ", treatment)) %>%
  arrange(name, treatment)
preg2
#>           name treatment  n
#> 1     Jane Doe         a  4
#> 2     Jane Doe         b  1
#> 3   John Smith         a NA
#> 4   John Smith         b 18
#> 5 Mary Johnson         a  6
#> 6 Mary Johnson         b  7
```
Elementary reshaping: gathering and spreading

gather() takes multiple columns, and gathers them into key–value pairs: it makes “wide” data longer.

spread() takes two columns (key & value) and spreads into multiple columns, it makes “long” data wider.
Example TSS coverage

```r
sample_n(covs,10)[,1:5]
#>             geneID -3000 -2999 -2998 -2997
#> 419  ENSMUSG00000029561 1     1
#> 138  ENSMUSG00000074896 0     0
#> 267  ENSMUSG00000050908 0     0
#> 406  ENSMUSG00000034616 2     2
#> 125  ENSMUSG00000041112 1     1
#> 217  ENSMUSG0000004085 0     0
#
```

the table summarizing the coverage around 150 TSS experiment with 3 conditions
apart from the conditions saved in the time column the data contains the geneIDs
this is a “wide” format

the data is courtesy of Elisabeth Zielonka (Hentze)
Gathering the data

dataGathered <- covs %>%
gather(key = "posRelToTSS",
       value = "coverage", -geneID, -time)

Gathering turns wide formats into long formats

sample_n(dataGathered, 10)
#>                       geneID time posRelToTSS coverage
#> 1  1790122 ENSMUSG000000089901  5h          4032     3
#> 2  2652590 ENSMUSG000000035692  WT          5975     1

Spreading does the opposite