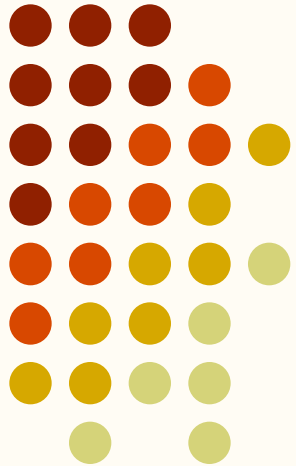
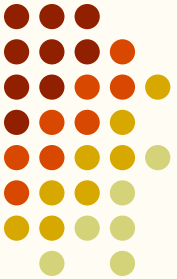
Basics

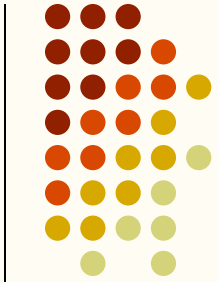


Outlines



- What is **R**? Why **R**?
- Basic concepts in **R**
- Data types and reading data
- Basic statistics and simple plots
- Customising plots
- Examples of advanced plots

Requirements



- **R** Knowledge: No previous R knowledge is required. Perhaps previous programming experience become handy.
- Statistical Knowledge: You are expected to know some basic statistical concepts and techniques such as mean, median, and variance

What is R

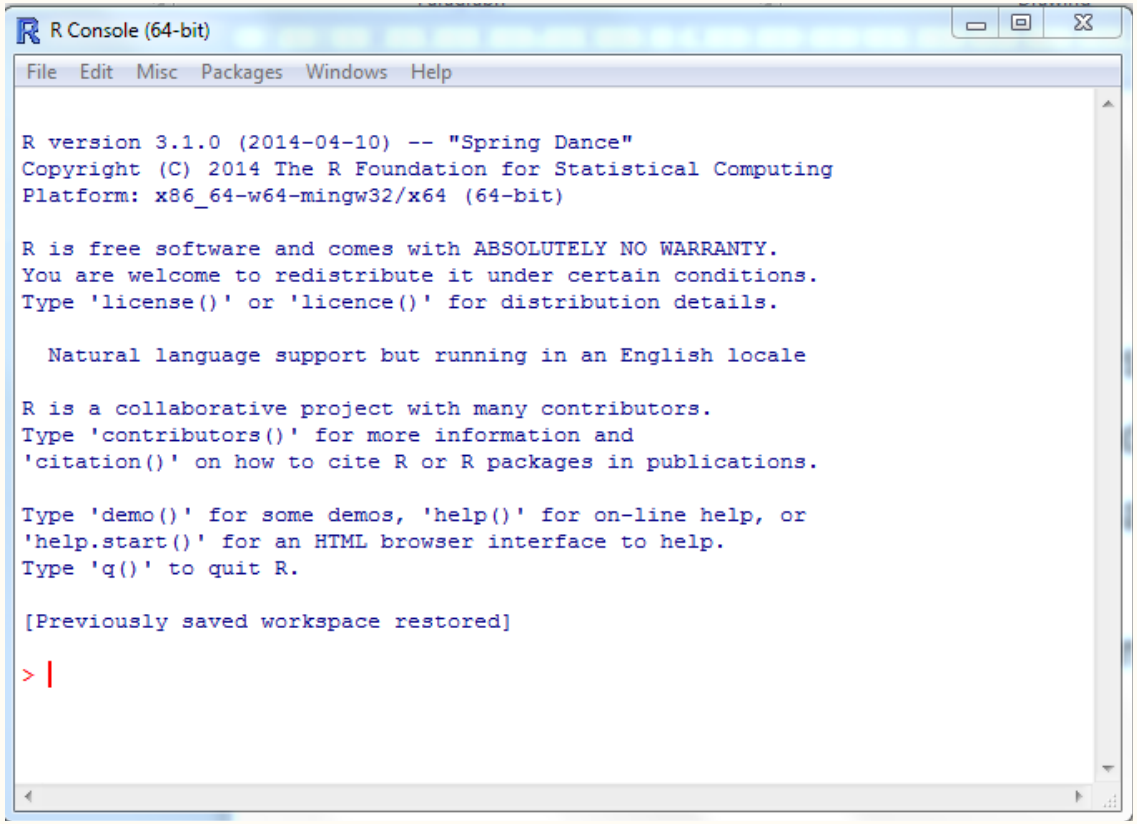
- **R** is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS.

http://www.r-project.org/

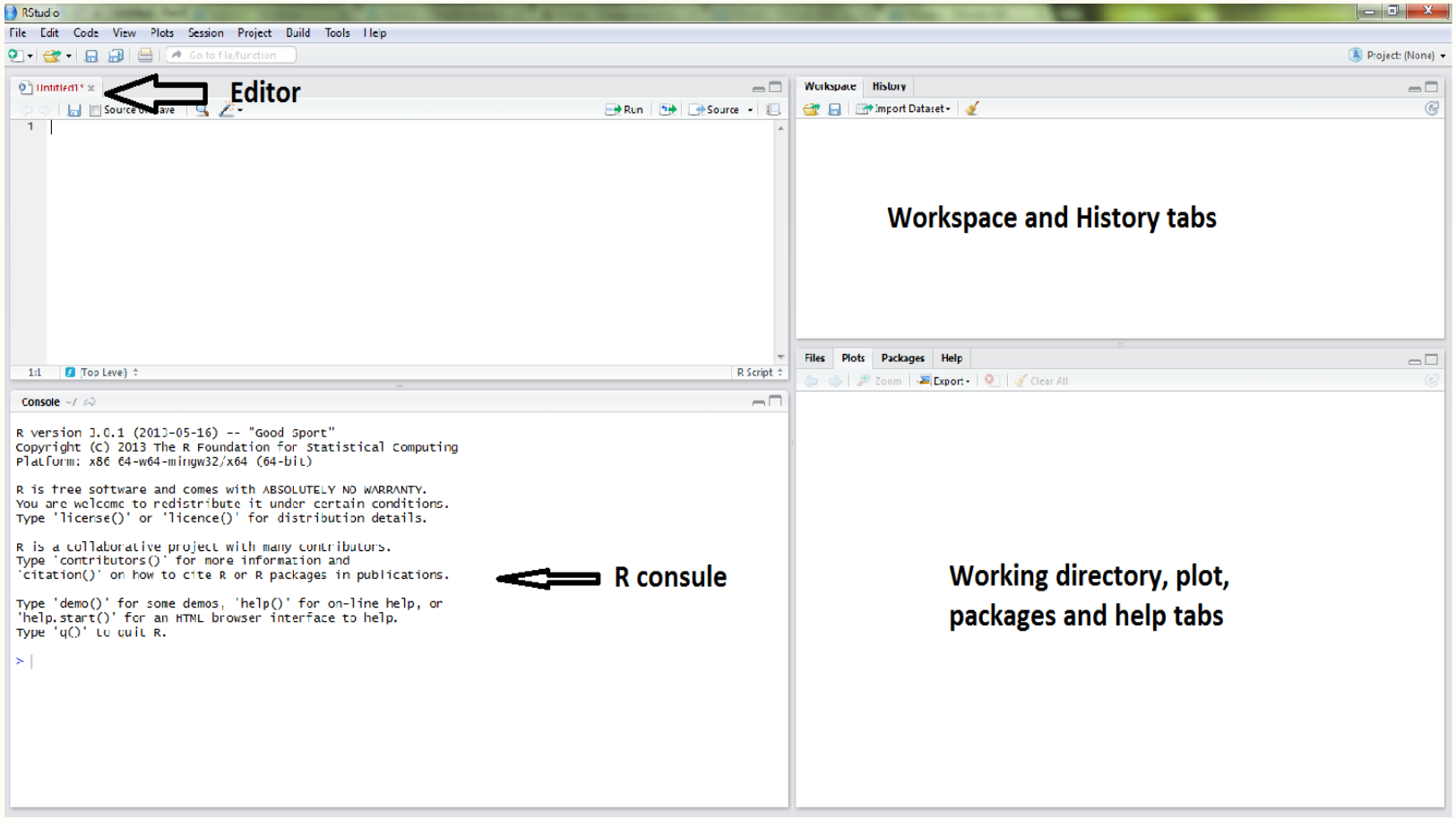
- RStudio is a set of integrated tools designed to help you be more productive with R.

http://www.rstudio.com

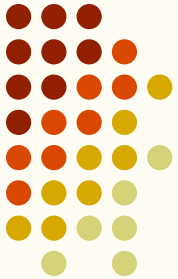
R environment



RStudio


RStudio
File Edit Code View Plots Session Project Build Tools Help
Editor
Workspace History
Import Dataset
Workspace and History tabs
Files Plots Packages Help
Export
Clear All
Console
R version 3.0.1 (2013-05-16) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
R consule
Working directory, plot,
packages and help tabs

First Things First



- Getting help from **R** built-in facility. You can call it by "?" as follows:

> ?function

Note that ">" is the prompt command, that is R is expecting you to input a command. You should not type it in **R**.

Do it yourself

```
> ?mean
```

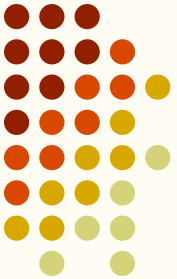
First Things First

- Do you need more information or can't you find what you are looking for? Use "??":

```
>??function
```

- Also you can obtain more details on features specified by special characters:

```
>?"[["
```



Do it yourself: try the following commands

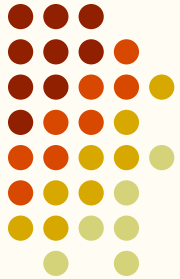
```
> ??mean↵
>?"%*%"↵
```

Let's try something new. Type the following command

```
> q() ↵
```

What does happen?

First Things First



NOTE: If a command is not complete at the end of a line, R will give a different prompt, by default is

+

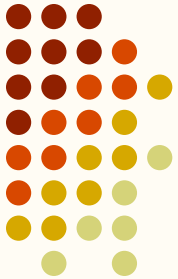
You will experience this a lot. So be careful with opening and closing your brackets, for example try

```
> mean(c(1,2,4,3)
```

or

- `q(`

First Things First



- Do you need example? Ok, use the example function:

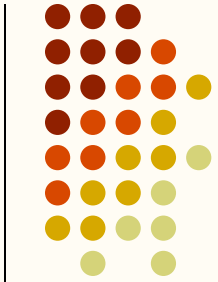
```
> example(topic)
```

Do it yourself: try the following commands

```
> example(mean) ↵

> example(sd) ↵
```

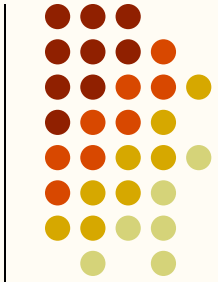
Expression



There are three types of expression in R

1. Numbers: 1,2,0.2,-5 , etc
2. Strings: alphabets or anything that is input by “” into R
3. Logical: TRUE/FALSE

Expression: examples



- Numbers:

> 2+3

- Strings:

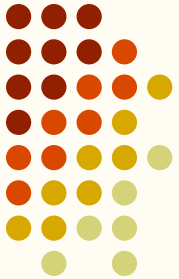
> “Hello”

- TRUE/FALSE

> 3<4

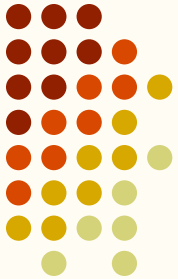
> 2+4 ==4

Arithmetic Operators



`x + y`	sum
`x - y`	subtract
`x * y`	multiply
`x / y`	divide
`x ^ y`	power

Arithmetic Operators

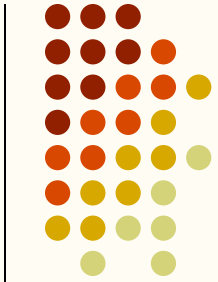


Do it yourself:

```
> 2+4↵

> 2^3↵
> 850/10↵
> 220-20↵
```

Storing Values



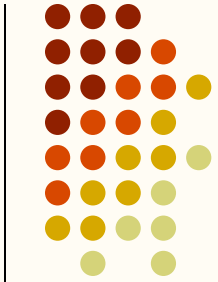
- In all programming languages we can store values in variables and access them later.
- This can be done in various ways using a selection of assignment operators. The most commonly used one is “<-”, see the example below:

```
> x<- 3

> y<- “Hey!”
```

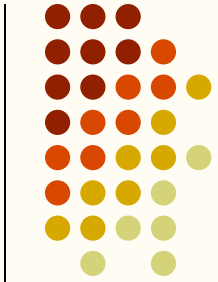
Note: In most contexts the ‘=’ operator can be used as an alternative.

Vectors



- A vector is simply a list of values. **R** relies on vectors for many of its operations, such as plots, basic statistics and statistical modelling.
- Values of vector can be numbers, strings, logical values or any other types, as long as they are all same type.

Vectors



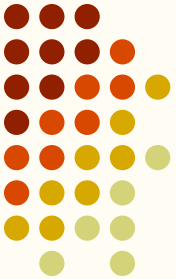
- Example: set up a vector named x, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

This is an *assignment* statement using the *function* `c()`.

In most contexts the '=' operator can be used as an alternative.

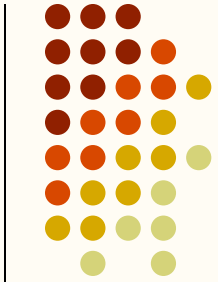
Vectors



Do it yourself:

```
> c(1,3,5) ↵
> c("H", "A", "B")↵
> c(TRUE, 2, "Sky") ↵
> y<- c(x, 0, x) ↵
> y↵
```

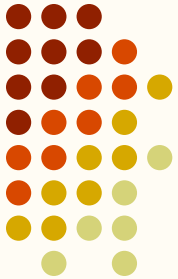
Vectors



- Vectors can be used in arithmetic expressions, in which case the operations are performed element by element.

```
> v <- 2*x + y + 1 ↵
> sum((x-mean(x))^2)/(length(x)-1) ↵
> sort(x) ↵
```

Matrix



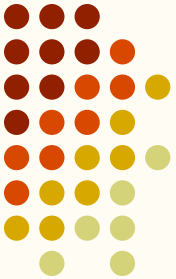
- Matrices are usually defined in **R** by function `matrix()`

```
> matrix(vector, nrow = n, ncol = m)
```

- You can define a diagonal matrix using the `diag()` function:

```
> diag(x, nrow= m, ncol=n)
```

Matrix



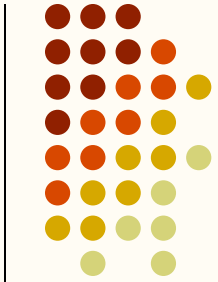
Do it yourself:

- Define a matrix of 3 rows and 2 columns with following vector

```
c(1,6,5,3,2,7)
```

- Define a diagonal matrix of 5 columns and 5 rows with the diagonal values of (3,6,9.1,-0.5,0.12)

Inputting Data

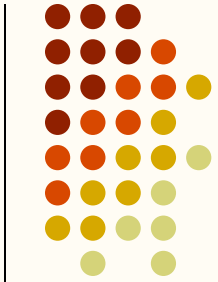


- **R** allows users to input data using a wide range methods.

  - ✓ Directly by typing the data into **R** (using scan())
  - ✓ Reading external files: txt, csv, SAS, SPSS, Excel.

I encourage you to learn different methods, but we will cover a common and robust use case: handling `csv` files.

Inputting Data : direct method



- You can directly input data points one by one using:
  - `scan()` function

Do it yourself

```
> x<- scan() ↵
13 2 1.2 3 18 6 ↵
> x ↵
```

This is called a *base function.*

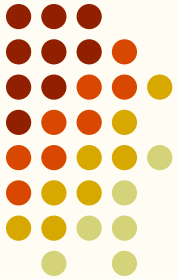
Inputting Data : external files

- External files come in various formats and a number of *convenience functions* are available:
  - `read.table()`
  - `read.csv()`
  - `read.delim()`
- Before we need to find out our working directory:

Do it yourself

```
> getwd()↵
```

Inputting Data : setting paths



- You can use `dir()` to find what is in each directory and `setwd()` to change to a new working directory.

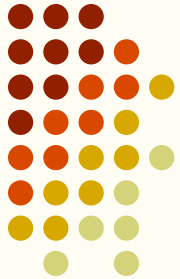
Do it yourself

```
> dir()↵
> setwd("C://Users/gsong/Desktop/
Data/")↵
```

Mac/Linux User?

```
> setwd('~/Desktop/Data/')↵
```

Inputting Data : example input



Do it yourself: Read the `simple.txt` data set and store it in a *data frame* called easy.

```
> easy <- read.table("simple.txt",
header = TRUE, sep = "\t")↵
```

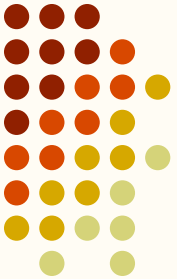
Let us look at the first 6 lines of the data:

```
> head(easy)↵
```

Now plot the data!

```
> plot(easy)↵
```

Inputting Data : comma separated



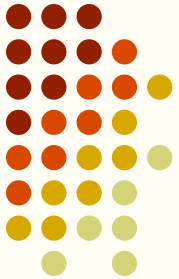
Do it yourself: Read the `smoking.csv` data set and store it in a *data frame* called smoking.

```
> smoking <- read.csv("smoking.csv",
header = TRUE)↵
```

Let us look at the first 6 lines of the data

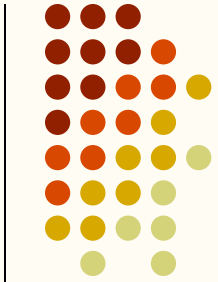
```
> head(smoking)↵
```

Data Frames



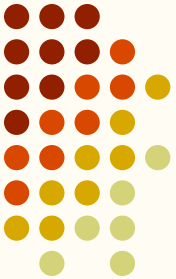
- A *data frame* is used for storing data tables. It is a list of vectors of equal length. For example both `easy` and `smoking` are data frames.
- The top line of the table, called the *header*, contains the column names.
- Each horizontal line afterward denotes a *data row*, which begins with the name of the row, and then followed by the actual data.

Built-in Data Frames



- We can also call built-in data frames in **R** for our tutorials.
- This can be done by using the `data()` command.
- For example, here is a built-in data frame in **R**, called `mtcars`.

Built-in Data Frames



Do it yourself: Call the R built-in data set `mtcars` as follows:

```
> data(mtcars) ↵
```

Let us look at the first 6 lines of the data

```
> head(mtcars) ↵
```

Find out more about it:

```
> ?mtcars ↵
```

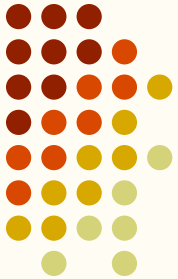
The Environment

Do it yourself: See all the objects and data in your environment:

```
> ls() ↵
```

Or you can see it in the top right corner of RStudio (Environment tab).

Plots

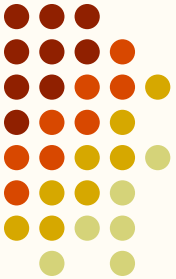


- Plots in R:
  - ✓ Plots of a single variable
  - ✓ Plots of two variables
  - ✓ Plots of three or more variables

We start by the plots of a single variable.

Plots: single variable

Scatter Plot



Do it yourself: Let us generate 100 random samples of a standard normal distribution, N(0,1) as follows:

```
> y <- rnorm(100, mean = 0, sd = 1) ↵
```

Produce a simple scatter plot:

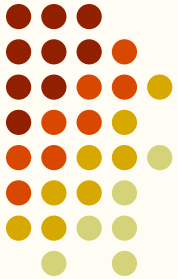
```
> plot(y) ↵
```

Find out more about the `plot` function:

```
> ?plot ↵
```

Plots: single variable

Scatter Plot



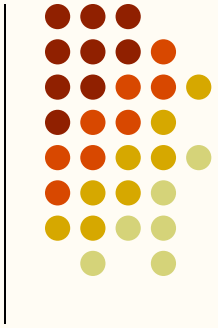
Do it yourself: Try to different plot types for y:

```
> plot(y, type = 'b') ↵
> plot(y, type = 'l') ↵
> plot(y, type = 'h') ↵
```

Explain how different each plot it.

Plots: single variable

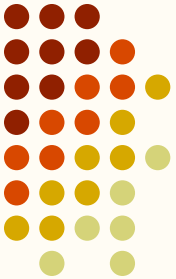
Scatter Plot



- Note: `type` is a plot option which allows you to choose between points “p”, bars “h”, line “l” and both points and line “b”.
- In general, options can be added in the **R** functions and are separated by “,”.
- Common options for the `plot` function and shared with other graphics functions are: `pch`, `xlab`, `ylab`, `xlim`, `ylim`, `cex`, `cex.lab`, `cex.axis`, `main` and `col`.
- You can set many of these globally with `par()`.

Plots: single variable

Histogram



Do it yourself: How about a histogram of y?

```
> hist(y, freq = FALSE) ↵
```

We can also add the curve of the empirical density of y to the histogram:

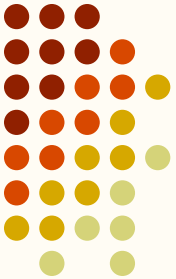
```
> lines(density(y)) ↵
```

Add the following options to the lines function:

```
col = "red"
```

Plots: single variable

Histogram



Do it yourself: You can also add the normal density to the histogram:

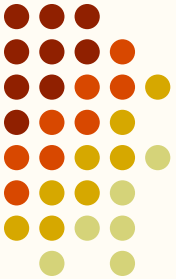
```
>  curve(dnorm(x), from = -4,to = 4,
add = TRUE, col = "purple")↵
```

Note: *Curve function* draws a curve corresponding to a function over the interval [from, to]. `curve` can plot also an expression in the variable xname, default x.

```
> curve(sin, -2*pi, 2*pi, xname = "t")
```

Plots: single variable

Bar charts



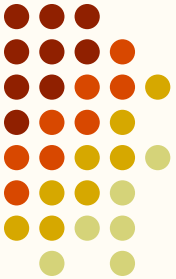
Note: Histogram and scatter plot are used for the continuous data. We can use `barplot` for plotting categorical variables.

Do it yourself: Construct a `barplot` for number of gears in `mtcars` data set:

```
> counts <- table(mtcars$gear) ↵
> counts↵
> barplot(counts, main="Car
Distribution”,xlab="Number of Gears”) ↵
```

Plots: two variables

Scatter plot



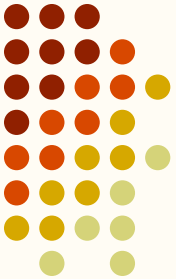
Do it yourself: Select the variables for “horse power” and “miles per gallon” from the “mtcars” data frame and plot them against each other:

```
> head(mtcars) ↵
> plot(mtcars$hp, mtcars$mpg) ↵
```

Note the $ sign. It selects a vector from the data frame `mtcars`.

Plots: two variables

Scatter plot of two groups



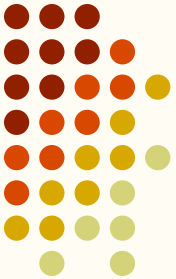
Do it yourself: We can plot horsepower grouped by weight and # of gears (at or above 3)

```
> plot(subset(mtcars, gear==3,
select=c(wt,hp)), ylim=c(0,250), col =
'blue') ↵

> points(subset(mtcars, gear>3,
select=c(wt,hp)), pch = 16, col =
'orange') ↵
```

Plots: two variables

Boxplot with one predictor



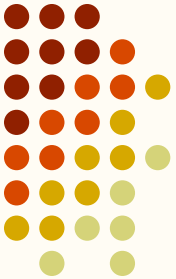
- You can use `boxplot()` to create a box and whiskers plot of a continuous variable affected by a predictor.

Do it yourself

```
> boxplot(mtcars$mpg ~ mtcars$am)↵
```

Plots: two variables

Saving your plot



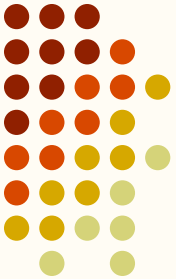
- You can use `pdf()` or `tiff()` to save your plot. Any changes you make to graphics will not be saved.

Do it yourself

```
> pdf(file="myplot.pdf", 7, 7) ↵
> par(col='red') ↵
> boxplot(mpg~am, data=mtcars) ↵
> dev.off() ↵
```

Plots: two variables

Boxplot with two predictors



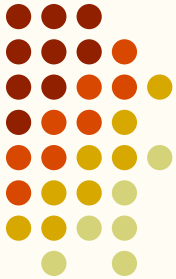
Do it yourself: We can plot by two categorical predictors using boxplot too.

```
> boxplot(mpg~vs*am, data=mtcars,
col=(c(”mistyrose",”lightblue")), main="Car
Engines", xlab="Config * Transmission",
ylab="Miles per gallon")↵
```

Try adding ‘notch=TRUE’ argument to above. Try saving your plot.

Plots: counts

Stacked Bar charts

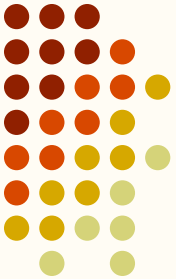


Do it yourself: Construct a `barplot` for number of `gears` in `mtcars` data set grouped by `vs`:

```
> counts <- table(mtcars$vs,mtcars$gear) ↵
> barplot(counts, main="Car Distribution
by Gears and VS", xlab="Number of Gears",
col=c("darkblue","red"),
legend=rownames(counts)) ↵
```

Plots: many variables

Contour, heat and 3D plots

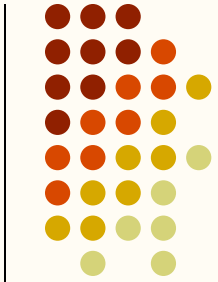


Do it yourself:

```
> library(MASS)
> x<- rnorm(100, 10,2)
> y<- -0.5 + 0.67 * x + rnorm(100,0,0.2)
> bivn.kde <-kde2d(x,y, n = 50)
> op <- par(mfrow = c(2,2))
> contour(bivn.kde)
> image(bivn.kde)
> persp(bivn.kde, phi = 10, theta =
30,col="grey")
> par(op)
```

Summary Statistics

summary()



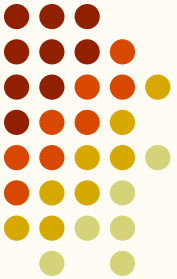
`summary` is a generic function used to produce result summaries of the results of various model fitting functions. Its general form is

```
summary(object, ...)
```

where object is an object for which a summary is desired. Object can be a data frame, matrix or a model.

Summary Statistics

summary()



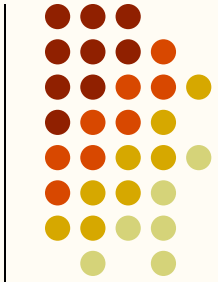
Do it yourself: Use the summary function for data set `mtcars`

```
> summary(mtcars)
```

`summary()` calculates min, 1st and 3rd quartiles, median, mean and max of each variable in the data frame.

Summary Statistics

sd() and var()



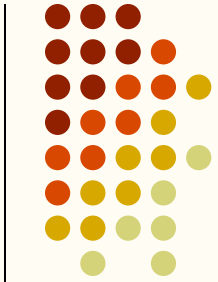
`sd()` and `var()` calculate standard deviation (SD) and the variance of a single vector. Variance-covariance matrix of a data frame can be calculated using the var() function.

Do it yourself: Calculate SD and variance of death rate from the `smoking` data set and number of seizures from the `epilepsy` data set.

```
> sd(mtcars$mpg/mtcars$wt)
> var(mtcars$mpg/mtcars$wt)
```

Tests of association

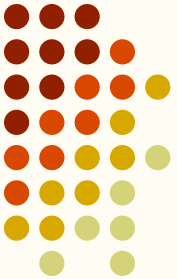
cor() and cor.test()



- `cor()` calculates correlation coefficient ("pearson", "kendall", "spearman") for a pair of variables and the correlation matrix for more than two variables.
- The significance of the linear relationship between two variables can be tested using `cor.test()`.

Summary Statistics

cor() and cor.test()



Do it yourself: Calculate correlation coefficient of `mpg` and `wt` from the `mtcars` data set. Is there a significant relationship between these two variables?

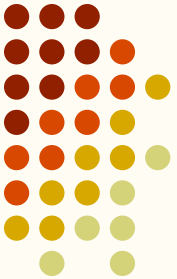
```
> cor(mtcars$mpg, mtcars$wt)
> cor.test(mtcars$mpg, mtcars$wt)
```

Do it yourself: Calculate the correlation matrix for the `mtcars` data set.

```
> cor(mtcars)
```

Summary Statistics

A bit advanced: aggregate()

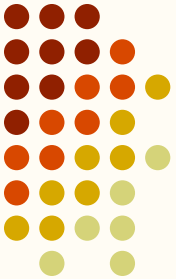


- Sometimes you need to obtain the summary statistics of a *data frame* x grouped by a *list* of grouping elements.
- For example death rate grouped by smokers and non-smokers.
- Use aggregate() for this purpose

```
aggregate(x, by, FUN )
```

Summary Statistics

aggregate()



Do it yourself: Calculate the average miles per gallons of cars grouped by number of cylinder ?

```
> aggregate(mtcars$mpg, by =
list(mtcars$cyl) , FUN = mean)
```