DSA2101

Essential Data Analytics Tools: Data Visualization

Yuting Huang

Week 2: Basics in `R` Programming II

Basics in R Programming

Week 1:

1. `R` objects
2. `R` syntax

Week 2:

3. `R` functions
4. Basic `R` plotting
5. Generate reports with `R` Markdown

Recap: R objects

1. What will be the output of the following R code?

```
m = matrix(0, nrow = 2, ncol = 3)
dim(m); length(m)
```

2. Consider the R output below. Which of the following was the correct command used to write this matrix?

```
##      [,1] [,2]
## [1,]    4    1
## [2,]    9    5
## [3,]   10    7
```

A. `matrix(c(4, 1, 9, 5, 10, 7), ncol = 3)`

B. `matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)`

C. `matrix(c(4, 9, 10, 1, 5, 7), nrow = 2)`

D. `matrix(c(4, 9, 10, 1, 5, 7), ncol = 2, byrow = TRUE)`

Conditional executions

The `if()` conditional statement executes one or more `R` command when the condition is met.

1. `if` statement

```
x = 8

if (x >= 10) {
  print("x is greater than or equal to 10.")
}
```

- The print statement will not appear in the console, because the condition is not met.

Conditional executions

2. `if... else` statement:

```
x = 8

if (x >= 10) {
  print("x is greater than or equal to 10.")
} else {
  print("x is less than 10.")
}
```

3. `if... else if... else` statement for multiple conditions:

```
x = 8

if (x >= 10) {
  print("x is greater than or equal to 10.")
} else if (x > 7) {
  print("x is greater than 7, but less than 10.")
} else {
  print("x is less than 7.")
}
```

Built-in `ifelse()` function

`R` also has a built-in `ifelse()` function.

- Basic syntax:

```
ifelse(test, action if TRUE, action if FALSE)
```

```r
x = 8
ifelse(x >= 10, "x is greater than or equal to 10.", "x is less than 10.")
```

- It is a shortcut for using `if()` and `else` in combination.

`for` loops

For loops iterate over items of a vector or a list.

```r
for (i in 1:5){
  print(i)
}
```

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

- The `i` here is a temporary variable.
- It is only important in that it matches how we call it later on in the statement.

A `for` loop with an `if-else` statement

```r
animals = c("cat", "dog", "dog", "pigeon")

for (i in animals) {
  if (i == "cat") {
    print("This is a cat.")
  } else {
    print("These are other animals.")
  }
}
```

```
## [1] "This is a cat."
## [1] "These are other animals."
## [1] "These are other animals."
## [1] "These are other animals."
```

`while` loops

A `while` loop is used when we want to perform a task indefinitely, until a particular **stopping condition** is met.

```r
i = 0
while (i <= 4) {
  i = i + 1
  print(i)
}
```

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

- The `i` here is the name of the variable we loop over.

A break in a `while` loop

Another way to accomplish this would be to use `break` to stop the iteration when certain condition is met.

```
i = 0
while (TRUE) {
  i = i + 1
  print(i)
  if (i > 4)
    break
}
```

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

`for()` versus `while()`

- `for()` is better when the number of times to iterate is clear in advance.
- `while()` is better when we know the stopping condition.
- `while()` is more general - every `for()` can be replaced with a `while()`, but not vice versa.
- `while()` loops can potentially result in infinite loops if not written properly. So we must use them with care.

The following code provides the same output:

```
i = 0
while (i < 4) {
  i = i + 1
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

```
for (i in 1:4) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

`R` functions

1. Built-in functions
2. The family of `apply` functions

R Functions

We have already encountered a few functions in R.

- As you have noticed, they can take arguments that modify their behavior.
- Not all arguments need to be specified.

R Functions

- For example, the `args()` function can be used to list the arguments of a function.

```
args(median)
```

```
## function (x, na.rm = FALSE, ...)
## NULL
```

- `x` does not have a default value. Its value must be supplied.
- `na.rm` has a default value, which is `FALSE`.

Built-in functions

The `seq()` command is used to generate regular sequences. It is more general than the `:` operator.

- Generate a sequence starting from a certain value up to another value and specify the increment of the sequence `by` a certain amount.

```
seq(1, 2, by = 0.2)
```

```
## [1] 1.0 1.2 1.4 1.6 1.8 2.0
```

- Generate a sequence starting from a certain value up to another value, with the final sequence having a particular `length`.

```
seq(1, 2, length = 3)
```

```
## [1] 1.0 1.5 2.0
```

Built-in functions

The `rep()` function is used to repeat a sequence.

- Repeat a sequence a specified number of `times`.

```
x = seq(1, 2, length = 3)
rep(x, times = 3)
```

```
## [1] 1.0 1.5 2.0 1.0 1.5 2.0 1.0 1.5 2.0
```

- Or repeat each element in a sequence a specified number of `times`.

```
rep(x, times = c(1, 2, 3))
```

```
## [1] 1.0 1.5 1.5 2.0 2.0 2.0
```

Built-in functions

`paste()` and `paste0()` can be used to concatenate vectors and convert them to character strings. Vector arguments will be recycled as needed.

- Concatenate strings with a space.

```
paste("A", 1:6)
```

```
## [1] "A 1" "A 2" "A 3" "A 4" "A 5" "A 6"
```

- Concatenate strings without spaces.

```
paste0("A", 1:6)
```

```
## [1] "A1" "A2" "A3" "A4" "A5" "A6"
```

Write R functions

At some point, we will have to write a function of our own.

We will need to decide

1. What argument it should take.
2. Whether these arguments should have default values, and if so, what those default values should be.
3. What output it should return.

The typical approach is to write a sequence of expressions that work, then package them into a function.

Example: A single game of dice

Let us suppose that we wish to write a function to simulate one game of dice between players A and B.

Code that simulates one single game:

```
set.seed(2101)
A = sample(1:6, size = 1); B = sample(1:6, size = 1)

if (A > B) {
  results = "A"
} else if (A == B) {
  results = "Draw"
} else {
  results = "B"
}

results
```

```
## [1] "B"
```

Function that simulates one single game:

```
single_game = function() {     # the function takes zero arguments

  A = sample(1:6, size = 1)
  B = sample(1:6, size = 1)

  if (A > B) {
    results1 = "A"
  } else if (A == B) {
    results1 = "Draw"
  } else {
    results1 = "B"
  }

  return(results1)                # return the vector named results1
}

set.seed(2101)                  # Set seed for reproducibility
single_game()                   # call the function
```

```
## [1] "B"
```

Clearer code for 1000 iterations

Rewrite the `for` loop that iterate the game 1000 times. It is much easier to read now.

```
set.seed(2101)                    # Set seed for reproducibility
results1 = rep(0, 1000)

for(i in 1:1000) {
  results1[i] = single_game()
}

table(results1)
```

```
## results1
##    A    B Draw
##  419  431  150
```

Roll more than one dice

Suppose that each player rolls more than 1 dice at each game and compares the **sum** of his/her dice to the component's.

```
single_game = function(n_dice = 1) {  # Takes one argument with default value 1

  A = sample(1:6, size = n_dice, replace = TRUE)
  B = sample(1:6, size = n_dice, replace = TRUE)

  if(sum(A) > sum(B)) {
    results1 = "A"
  } else if (sum(A) == sum(B)) {
    results1 = "Draw"
  } else {
    results1 = "B"
  }
  return(results1)
}

set.seed(2101)         # Set seed for reproducibility
single_game(2)         # Each player rolls 2 dice.
```

```
## [1] "B"
```

Default value of an argument

In the previous code, we declared the argument and its default value by `n_dice = 1`. When we call the function and leave the argument empty, R will take in the default value.

```
single_game()        # By default, each player roll one dice.
```

- However, we would get an error if we have declared the arguments by `n_dice` only. Without a default value, R would not know what value to use for `n_dice` in this case.

```
single_game() = function(n_dice) {

  # FUNCTION CODE HERE

}
single_game()
```

The family of `apply` functions

In data analysis, we often find ourselves having to repeat the same operation multiple times.

- For instance, we may need to split a data set by age group, and then compute the mean height for each age group.
- Or we may have a matrix of values, and we need to take the mean of each column or each row.

The family of `apply` functions

The `apply` functions are a family of functions in base `R` that allow us to repeatedly perform an action on data. We will cover:

- `apply()`: apply a function across rows or columns of a matrix or a data frame.
- `sapply()` apply a function across elements of a list, and return a vector or a matrix.
- `lapply()`: apply a function across elements of a list, and return a list.
- `tapply()`: apply a function on a subset of data frame broken down by factor levels.

apply()

If we have a matrix and want to apply a function to each row or column separately, then the `apply()` function is what we need.

Let us first generate a 10×3 matrix with values 1-30 in the entries.

```
M = matrix(1:30, nrow = 10, ncol = 3)
M
```

```
##       [,1] [,2] [,3]
##  [1,]    1   11   21
##  [2,]    2   12   22
##  [3,]    3   13   23
##  [4,]    4   14   24
##  [5,]    5   15   25
##  [6,]    6   16   26
##  [7,]    7   17   27
##  [8,]    8   18   28
##  [9,]    9   19   29
## [10,]   10   20   30
```

apply()

Basic syntax: `apply(X, MARGIN, FUN)`

- The argument `X` can a matrix or a data frame.
- We first apply the `mean` function to each column of the matrix (`MARGIN = 2`), thus computing the column means.

```
apply(M, 2, mean)
```

```
## [1]  5.5 15.5 25.5
```

- Next, we apply the function `mean` to each row of the matrix (`MARGIN = 1`), thus computing the row means.

```
apply(M, 1, mean)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

apply() with an anonymous function

Instead of using pre-defined functions (such as `mean()` and `sum()`) to apply to each row or column, we can define functions on-the-fly with an **anonymous function**.

- Let's say we want to count the proportion of values that are greater than 7 in each column:

```
apply(M, 2, function(x) sum(x > 7)/nrow(M))
```

```
## [1] 0.3 1.0 1.0
```

- The anonymous function takes one argument, which we have arbitrarily called `x`.
- In this case, `x` is a single column of the matrix `M` (indicated by `2`).

apply() with an anonymous function

What does the following code do?

```
apply(M, 2, function(x) mean(x[x > 7]))
```

```
## [1]  9.0 15.5 25.5
```

```
apply(M, 1:2, function(x) x + 3)
```

```
##       [,1] [,2] [,3]
##  [1,]    4   14   24
##  [2,]    5   15   25
##  [3,]    6   16   26
##  [4,]    7   17   27
##  [5,]    8   18   28
##  [6,]    9   19   29
##  [7,]   10   20   30
##  [8,]   11   21   31
##  [9,]   12   22   32
## [10,]   13   23   33
```

Example: `USArrests` data

Here is an example of `apply`ing anonymous functions on real data.

```
data(USArrests)
head(USArrests, 3)
```

```
##         Murder Assault UrbanPop Rape
## Alabama   13.2     236       58 21.2
## Alaska    10.0     263       48 44.5
## Arizona    8.1     294       80 31.0
```

```
apply(USArrests, 2, function(x) c(min(x), median(x), max(x), sd(x)))
```

```
##        Murder   Assault UrbanPop      Rape
## [1,]  0.80000  45.00000 32.00000  7.300000
## [2,]  7.25000 159.00000 66.00000 20.100000
## [3,] 17.40000 337.00000 91.00000 46.000000
## [4,]  4.35551  83.33766 14.47476  9.366385
```

sapply() and lapply()

If the object that we wish to iterate over is a list, we use `sapply()` and `lapply()`

- `sapply()` takes in a list and returns a vector or a matrix.
- `lapply()` takes in a list and returns a list.

Let us create a list and then apply the two functions to compare the outputs.

```
Y = list(a = 1:10,
         b = seq(1, 10, length = 5),
         c = c(TRUE, TRUE, FALSE))
```

- `sapply()` returns a numeric vector containing the output of a function.

```
sapply(Y, mean)
```

```
##         a         b         c
## 5.5000000 5.5000000 0.6666667
```

- While `lapply()` returns a list.

```
lapply(Y, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] 5.5
##
## $c
## [1] 0.6666667
```

tapply()

- `tapply()` applies an operation on a subset of data broken down by a given factor variable.

```
# Create synthetic data
set.seed(11)
df = data.frame(price = rnorm(100, sd = 5, mean = 20),
                city = sample(paste("City", 1:4), size = 100, replace = TRUE),
                region = sample(paste0("R", 1:4), size = 100, replace = TRUE))
df$city = factor(df$city)
df$region = factor(df$region)

# Use tapply() to compute group summaries
tapply(df$price, df$city, mean)
```

```
##   City 1   City 2   City 3   City 4
## 20.37629 17.94230 19.23478 19.36173
```

tapply()

We can also use `tapply()` on multiple factor variables, by passing them through a `list()` function.

- In the following, we compute the mean product price by city and region, and round the results to the second decimal place.

```r
# Use tapply() to compute average price by city and region
tapply(df$price, list(df$city, df$region), function(x) round(mean(x), 2))
```

```
##           R1    R2    R3    R4
## City 1 21.00 21.85 19.34 19.27
## City 2 14.77 19.24 14.91 19.68
## City 3 18.47 19.11 18.73 20.72
## City 4 19.79 22.80 18.61 17.72
```

Manipulation of important class types

Now we turn to introduce functions to manipulate some important types of `R` objects, including

1. String
2. Factor
3. Date

Strings in R

To manipulate strings, we shall use the `stringr` package, since it is more powerful than the base R functions.

- Install the package first, and then load the package:

```
# install.packages("stringr")
library(stringr)
```

Creating strings

In R, we can create a string using single or double quotes.

```
string1 = "This is a string"        # a string
string2 = c("one", "two", "three") # a vector of string
```

The `str_length()` function returns the number of characters in a string.

```
str_length(string1)
```

```
## [1] 16
```

```
str_length(string2)
```

```
## [1] 3 3 5
```

Combining strings

To combine strings, we can use the function `str_c()`.

- It is an alternative to `paste()` and `paste0()`.

```
str_c("x", "y", "z")
```

```
## [1] "xyz"
```

```
str_c("x", "y", "z", sep = ",")
```

```
## [1] "x,y,z"
```

Subsetting strings

To subset a string, we can use the function `str_sub()`.

```
x = c("apple", "banana", "orange")
str_sub(x, 1, 3)
```

```
## [1] "app" "ban" "ora"
```

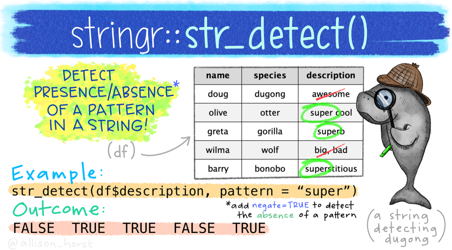
```
str_sub(x, -3, -1)  # count backwards using negative indices
```

```
## [1] "ple" "ana" "nge"
```

Parsing strings

Regular expressions are a language for expressing patterns in strings.

- They are used in many programming languages, not just `R`. So it is worth knowing a little about them.



Artwork by Allison Horst

Matching patterns

To detect presence/absence of a pattern, we use `str_detect()`.

- Basic syntax: `str_detect(string, pattern)`.

```
x = c("apple", "banana", "orange")
str_detect(x, "a")
```

```
## [1] TRUE TRUE TRUE
```

- To detect a string that begins with **a**:

```
str_detect(x, "^a")
```

```
## [1]  TRUE FALSE FALSE
```

Matching patterns

- To detect a string that ends with **a**:

```
str_detect(x, "a$")
```

```
## [1] FALSE  TRUE FALSE
```

- To detect a string that contains three characters with **a** in the middle:

```
str_detect(x, ".a.")
```

```
## [1] FALSE  TRUE  TRUE
```

Example: USArrests

- Let us try to identify the states that start with **M**:

```
data(USArrests)
USArrests$state = row.names(USArrests)      # add a column to store row names
id1 = which(str_detect(USArrests$state, "^M"))
USArrests[id1, ]
```

```
##               Murder Assault UrbanPop Rape         state
## Maine            2.1      83       51  7.8         Maine
## Maryland        11.3     300       67 27.8      Maryland
## Massachusetts    4.4     149       85 16.3 Massachusetts
## Michigan        12.1     255       74 35.1      Michigan
## Minnesota        2.7      72       66 14.9     Minnesota
## Mississippi     16.1     259       44 17.1   Mississippi
## Missouri         9.0     178       70 28.2      Missouri
## Montana          6.0     109       53 16.4       Montana
```

Factors in `R`

We use factors when we work with categorical variables in `R`.

- These are variables that have a fixed and known set of possible values.
  - Examples: gender, disease status, calendar month, ...
- We shall see that they are useful for dividing our data set into groups and performing analyses.

Creating factors

Suppose we have a vector containing month names:

```
x1 = c("Dec", "Apr", "Jan", "Mar", "Apr")
```

To convert the character vector `x1` to a factor:

```
factor(x1)
```

```
## [1] Dec Apr Jan Mar Apr
## Levels: Apr Dec Jan Mar
```

The levels of a factor are the possible values that the variable could take on.

- By default, they are arranged by **alphabetical** order, which may not correctly represent the natural ranking of the categories.
- If we plot a graph, April will come before January.

Levels of a factor

We can correct this using the following code.

- `R` also needs to be told about all the remaining months, even though they do not appear in the data set.

```
month_levels = c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
                 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
x2 = factor(x1, ordered = TRUE, levels = month_levels)
x2
```

```
## [1] Dec Apr Jan Mar Apr
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

- With the argument `ordered = TRUE`, we indicate that the factor is ordered.

Dates in R

R contains a Date class type to work easily with dates.

- Dates are stored internally as integers since 1st Jan 1970.
- This allow R to compute differences between dates, sequences of dates, and divide dates into convenient periods.

The easiest way to create a Date object is from character strings:

```
d1 = as.Date("2022/11/23", "%Y/%m/%d")
class(d1)
```

```
## [1] "Date"
```

```
d1
```

```
## [1] "2022-11-23"
```

Sequence of dates

The `seq()` function works just as well with `Date` objects.

- The following code creates a sequence of dates starting 100 days ago from today.
- The values in the sequence are 7 days apart.

```
today = Sys.Date()          # get the date of today
s1 = seq(today - 100, today, by = "1 week")
s1[1:3]
```

```
## [1] "2022-10-07" "2022-10-14" "2022-10-21"
```

Functions to manipulate dates

There are several convenient functions for extracting information that we typically need from `Date` objects.

```
weekdays(d1, abbreviate = FALSE)
```

```
## [1] "Wednesday"
```

```
months(d1, abbreviate = FALSE)
```

```
## [1] "November"
```

- Later on, we shall introduce functions from the `lubridate` package that provides even more convenient functions.

Base `R` plotting

1. Scatterplot
2. Bar plot

Scatterplot

The default `plot()` function takes arguments `x` and `y`

- `x` is considered as the horizontal axis
- `y` is considered the vertical axis
- They should be vectors of the same length

The axis, scales, titles, and plotting symbols are all chosen automatically. But these can be manually overridden.

`cars` data set

The `cars` data set (available in base `R`) contains two columns:

- `speed` in miles per hour.
- `distance` taken to stop, in feet.

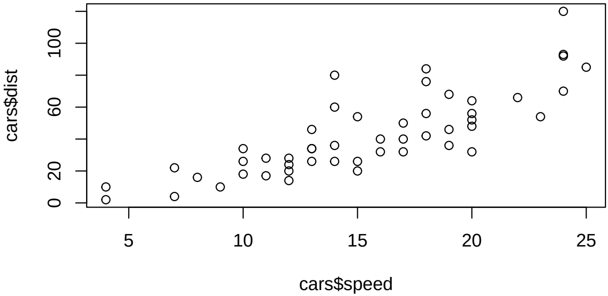
```
data(cars)
head(cars, n = 3)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

Scatterplot

The following command creates a basic plot

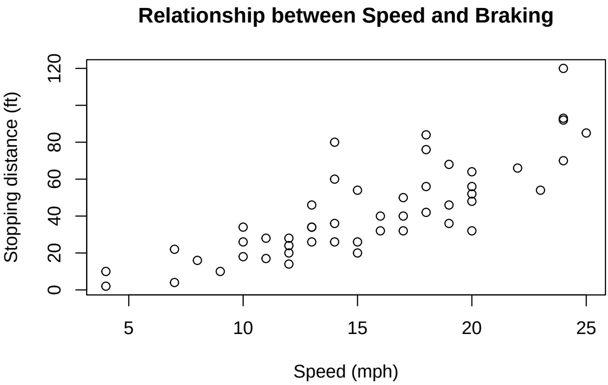
```
plot(cars$speed, cars$dist)
```



Adding axis label and title

We can overwrite some default arguments to create a more informative plot.

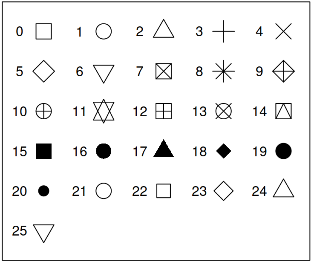
```
plot(cars$speed, cars$dist,
     xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     main = "Relationship between Speed and Braking")
```



Altering plotting symbols

In R, the plotting symbols are referred to as the plotting character (`pch` for short).

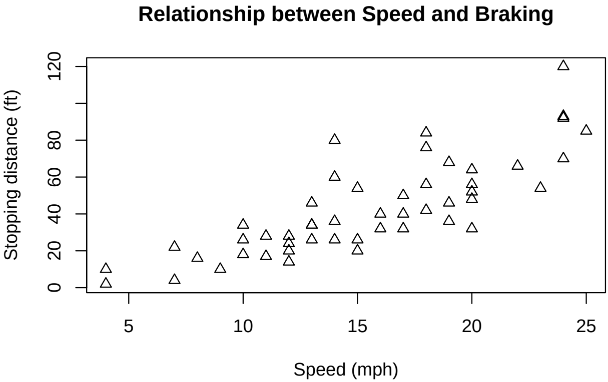
- Change the symbol by specifying the `pch` argument to `plot()`.
- The full list of symbols is displayed below.
- The default is `pch = 1`.



Altering plotting symbols

For instance, we can use unfilled triangles instead of unfilled circles.

```
plot(cars$speed, cars$dist, pch = 2,
     xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     main = "Relationship between Speed and Braking")
```



Altering plotting symbols

To change the size of the plotting character, we can modify the `cex` argument.

- `cex` stands for "character expansion", with a default value of 1.
- Larger values will make the symbol larger.
- This is an important abbreviation, because you will see a similar argument in a lot of help pages referring to other parameters:
  - `cex.axis` affects the font size of the axis
  - `cex.main` affects the font size of the title, and so on.

Colors

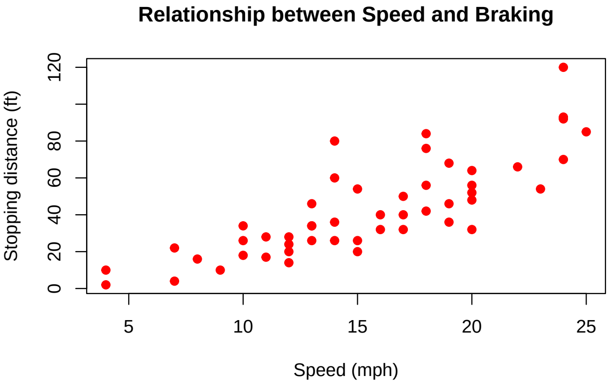
The color of the plotting characters can be changed using the `col` argument of the `plot()` function.

- The common colors can be accessed by their names, e,g, blue, green, red

```
plot(cars$speed, cars$dist, pch = 19, col = "red", cex = 1.5,
     xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     main = "Relationship between Speed and Braking")
```

- To see a list of all named colors in `R`, run the command `colors()`.
- A useful reference for colors: http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

Colors


Relationship between Speed and Braking
Stopping distance (ft)
0
20
40
60
80
120
5
10
15
20
25
Speed (mph)

Altering colors

When there are points that are very close to each other, it is useful to plot them using semi-transparent colors.

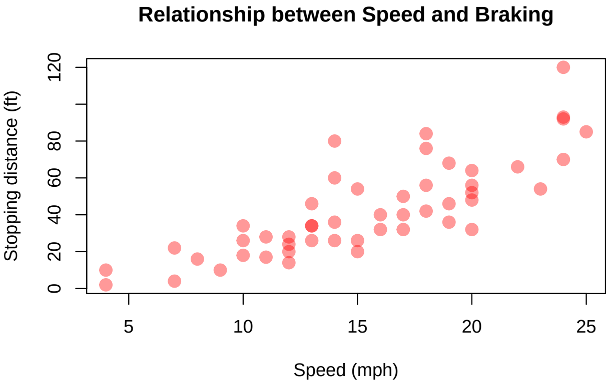
To use this feature with base R plotting, we have to create the color ourselves.

```
new_red = rgb(1, 0, 0, alpha = 0.4)            # RGB color specification

plot(cars$speed, cars$dist, col = new_red, pch = 19, cex = 1.5,
     xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     main = "Relationship between Speed and Braking")
```

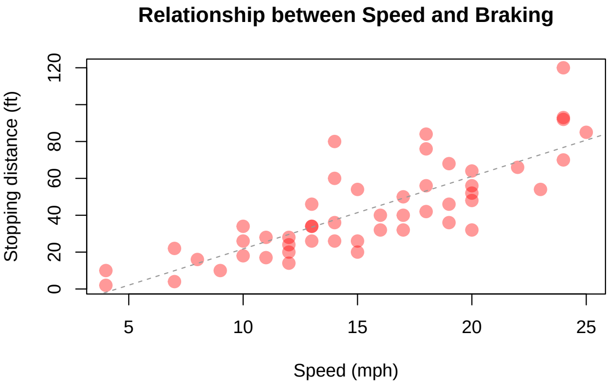
Read the help pages to understand what the arguments in `rgb()` do, or simply play around with them to see their effects.

Altering colors



Adding a trend line

```
new_red = rgb(1, 0, 0, alpha = 0.4)          # RGB color specification
plot(cars$speed, cars$dist, col = new_red, pch = 19, cex = 1.5,
     xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     main = "Relationship between Speed and Braking")
abline(reg = lm(dist ~ speed, data = cars), col = "gray60", lty = "dashed")
```



Bar charts

Bar charts represent a variable by drawing bars whose heights are proportional to the values of the variable.

Let us create a bar chart using a data frame that we set up early on.

Budget category	**Amount**
Manpower	\$519.4m
Asset	\$38.0m
Other	\$141.4m

Bar plot

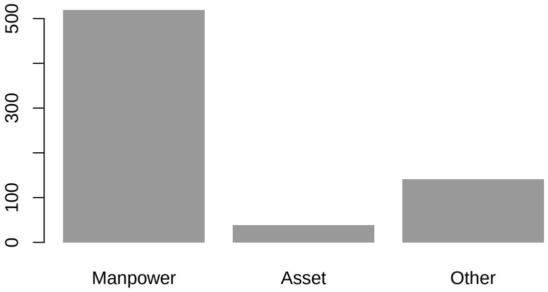
- Create the data frame:

```
budget_cat = c("Manpower", "Asset", "Other")
amount = c(519.4, 38.0, 141.4)
op_budget = data.frame(budget_cat, amount)
op_budget
```

```
##   budget_cat amount
## 1   Manpower  519.4
## 2      Asset   38.0
## 3      Other  141.4
```

Bar plot

```
barplot(op_budget$amount,
        border = NA, col = "gray60", names.arg = op_budget$budget_cat)
```



Summary

The base `R` plotting commands are quite powerful indeed. They give you full control over every element in a plot window.

- For us though, we are going to focus on a slightly different paradigm when plotting **- the grammar of graphics**.
- We will go into more details about this later. For now, this section was meant to give some knowledge on plotting with base `R`.
- As we proceed, we will see more examples of plotting with base `R` until we reach the grammar of graphics topic.

Generate reports with `R` Markdown

Introduction

- `R` Markdown is a language that allows you to combine code, its outputs, and your text into one text document.
- The text document can then be **knitted** into a range of output formats, including HTML and PDF.
- It is useful when you wish to
  - write a report based on your analysis (which is what you will be doing for the next few years in NUS).
  - share your work and findings with others. They will be able to easily reproduce your exact findings.
  - capture all your analyses on your data set.

`R` Markdown pre-requisites

Run the following code to install the required packages:

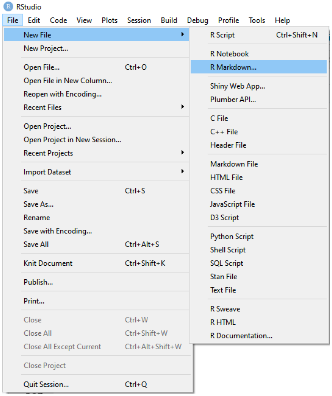
```
install.packages(c("knitr", "tinytex"))
library(knitr)
tinytex::install_tinytex()
```

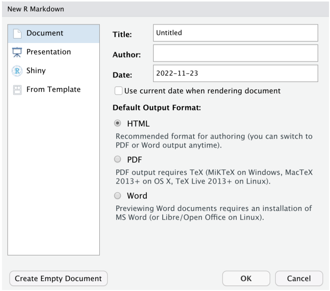
- `knitr` is a package that converts `R` markdown files to other formats such as HTML and Word.
- `tinytex` is a lightweight package that compiles `R` markdown document to a PDF file.

R Markdown basics

To create a new `R` Markdown file, click on `File -> New file -> R Markdown...`

- Select HTML as the default output format.





R Markdown basics

The first section of an `Rmd` file is usually a YAML header. It will look like this:

```
---
title: "Untitled"
date: "2022-11-23"
output: html_document
---
```

- YAML stands for Yet Another Markup Language.
- We usually do not have to write this ourselves. We can specify certain options and RStudio will write this part for us.

R Markdown basics

The rest of the Rmd file will consist of code chunks (R code) and text.

- **Chunks** of R code will be surrounded by tickmarks.
- **Text** will be simple text, formatted with #, *, and _.

````
## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF,
and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as
well as the output of any embedded R code chunks within the document. You can embed an R code
chunk like this:

```{r cars}
summary(cars)
```
````

Chunk options

- `eval = FALSE` tells `R` to print the code, but not run it.
- `echo = FALSE` tells `R` to run the code, but not print it.
- `include = FALSE` tells `R` to run the code, but not to include the output in the document.
- `message = FALSE, warnings = FALSE` suppress warning messages from appearing in your output document.
- `fig.asp = 0.618, out.width = "80%"` specify the figure aspect ratio and the size of the figure in your output document.

Chunk caching

Sometimes one or more of our code chunks is computationally expensive.

- In these cases, we do not want to run the chunk every time we knit the file.
- We would want to run it again only if some code in it has changed.
- To do this, we can put the `cache = TRUE` option in the code chunk option.

Learn more

`R` Markdown is a great tool for sharing your work.

- You no longer have to zip up source code, images, and PDF output to share with your team mates or colleagues.
- Just one `.Rmd` file, and they can do what you have done, exactly.
- It will take a short while to get used to the formatting. After that, it will become very easy to use.

The first chapter of the **Reporting with R Markdown** DataCamp assignment will help you get familiar with the Markdown syntax.