Week 2: R Basics

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

19 September 2022

Module website: tinyurl.com/POP77001

Overview

- Backstory
- R operators and objects
- Data structures and types
- Indexing and subsetting
- Attributes

R background

Source: University of Auckland, R Project

- S (for **s**tatistics) is a programming language for statistical analysis developed in 1976 in AT&T Bell Labs
- Original S language and its extention S-PLUS were closed source
- In 1991 **R**oss Ihaka and **R**obert Gentleman began developing R, an open-source alternative to S

R Release Names

- `4.2.1` - "Funny-Looking Kid"
- `4.2.0` - "Vigorous Calisthenics"
- `4.1.3` - "One Push-Up"

Source: GoComics
Extra: More on historical R release names

R basics

- R is an *interpreted* language (like Python and Stata)
- It is geared towards statistical analysis
- R is often used for interactive data analysis (one command at a time)
- But it also permits to execute entire scripts in *batch* mode

R basics

- R is an *interpreted* language (like Python and Stata)
- It is geared towards statistical analysis
- R is often used for interactive data analysis (one command at a time)
- But it also permits to execute entire scripts in *batch* mode

In [2]:
```
print("Hello World!")
```

```
[1] "Hello World!"
```

Operators

Key *operators* (*infix* functions) in R are:

- Arithmetic (`+`, `-`, `*`, `^`, `/`, `%/%`, `%%`, `%*%`)
- Boolean (`&`, `&&`, `|`, `||`, `!`)
- Relational (`==`, `!=`, `>`, `>=`, `<`, `<=`)
- Assignment (`<-`, `<<-`, `=`)
- Membership (`%in%`)

Basic mathematical operations in R

Basic mathematical operations in R

In [3]:
```
1 + 1
```

```
[1] 2
```

Basic mathematical operations in R

In [3]:
```
1 + 1
```

```
[1] 2
```

In [4]:
```
5 - 3
```

```
[1] 2
```

Basic mathematical operations in R

In [3]:
```
1 + 1
```

```
[1] 2
```

In [4]:
```
5 - 3
```

```
[1] 2
```

In [5]:
```
6 / 2
```

```
[1] 3
```

Basic mathematical operations in R

In [3]:
```
1 + 1
```

```
[1] 2
```

In [4]:
```
5 - 3
```

```
[1] 2
```

In [5]:
```
6 / 2
```

```
[1] 3
```

In [6]:
```
4 * 4
```

```
[1] 16
```

Basic mathematical operations in R

In [3]:
```
1 + 1
```

```
[1] 2
```

In [4]:
```
5 - 3
```

```
[1] 2
```

In [5]:
```
6 / 2
```

```
[1] 3
```

In [6]:
```
4 * 4
```

```
[1] 16
```

In [7]:
```
## Exponentiation, note that 2 ** 4 also works, but is not recommended
2 ^ 4
```

```
[1] 16
```

Advanced mathematical operations in R

Advanced mathematical operations in R

In [8]:
```r
# Integer division, equivalent to Python's `//`
7 %/% 3
```

```
[1] 2
```

Advanced mathematical operations in R

In [8]:
```
# Integer division, equivalent to Python's `//`
7 %/% 3
```

```
[1] 2
```

In [9]:
```
# Modulo operation (remainder of division), equivalent to Python's `%`
7 %% 3
```

```
[1] 1
```

Basic logical operations in R

Basic logical operations in R

In [10]:
```
3 != 1 # Not equal
```

```
[1] TRUE
```

Basic logical operations in R

In [10]:
```
3 != 1 # Not equal
```

```
[1] TRUE
```

In [11]:
```
3 > 3 # Greater than
```

```
[1] FALSE
```

Basic logical operations in R

In [10]:
```
3 != 1 # Not equal
```

```
[1] TRUE
```

In [11]:
```
3 > 3 # Greater than
```

```
[1] FALSE
```

In [12]:
```
FALSE | TRUE # True if either first or second operand is True, False ot
```

```
[1] TRUE
```

Basic logical operations in R

In [10]:
```
3 != 1 # Not equal
```

```
[1] TRUE
```

In [11]:
```
3 > 3 # Greater than
```

```
[1] FALSE
```

In [12]:
```
FALSE | TRUE # True if either first or second operand is True, False ot
```

```
[1] TRUE
```

In [13]:
```
F | T # R also treats F and T as Boolean, but it is not recommended due
```

```
[1] TRUE
```

Basic logical operations in R

In [10]:
```
3 != 1 # Not equal
```

```
[1] TRUE
```

In [11]:
```
3 > 3 # Greater than
```

```
[1] FALSE
```

In [12]:
```
FALSE | TRUE # True if either first or second operand is True, False ot
```

```
[1] TRUE
```

In [13]:
```
F | T # R also treats F and T as Boolean, but it is not recommended due
```

```
[1] TRUE
```

In [14]:
```
3 > 3 | 3 >= 3 # Combining 3 Boolean expressions
```

```
[1] TRUE
```

R objects

> *Everything is an object.*
>
> *John Chambers*

- Fundamentally, everything you are dealing with in R is an *object*
- That includes individual variables, datasets, functions and many other classes of objects
- The key reference to an object is its *name*
- Typically, the reference is established through assignment operation

Assignment operations

- `<-` is the standard assignment operator in R
- While `=` is also supported it is not recommended
- As it hides the difference between `<-` and `<<-` (deep assignment)

Extra: R Documentation on assignment

Assignment operations

- `<-` is the standard assignment operator in R
- While `=` is also supported it is not recommended
- As it hides the difference between `<-` and `<<-` (deep assignment)

Extra: R Documentation on assignment

In [15]:
```
x <- 3
x
```

```
[1] 3
```

Assignment operations

- `<-` is the standard assignment operator in R
- While `=` is also supported it is not recommended
- As it hides the difference between `<-` and `<<-` (deep assignment)

Extra: R Documentation on assignment

In [15]:
```
x <- 3
x
```

```
[1] 3
```

In [16]:
```
x <- 3
f <- function() {
    x <<- 1 # Modifies the existing variable in parent namespace (or cr
}
f()
x
```

```
[1] 1
```

Membership operations

Operator `%in%` returns `TRUE` if an object of the left side is in a sequence on the right.

Membership operations

Operator `%in%` returns `TRUE` if an object of the left side is in a sequence on the right.

In [17]:
```r
"a" %in% "abc" # Note that R strings are not sequences
```

```
[1] FALSE
```

Membership operations

Operator `%in%` returns `TRUE` if an object of the left side is in a sequence on the right.

In [17]:
```r
"a" %in% "abc" # Note that R strings are not sequences
```

```
[1] FALSE
```

In [18]:
```r
3 %in% c(1, 2, 3) # c(1, 2, 3) is a vector
```

```
[1] TRUE
```

Membership operations

Operator `%in%` returns `TRUE` if an object of the left side is in a sequence on the right.

In [17]:
```r
"a" %in% "abc" # Note that R strings are not sequences
```

```
[1] FALSE
```

In [18]:
```r
3 %in% c(1, 2, 3) # c(1, 2, 3) is a vector
```

```
[1] TRUE
```

In [19]:
```r
!(3 %in% c(1, 2, 3))
```

```
[1] FALSE
```

Data structures

Base R data structures can be classified along their *dimensionality* and *homogeneity*

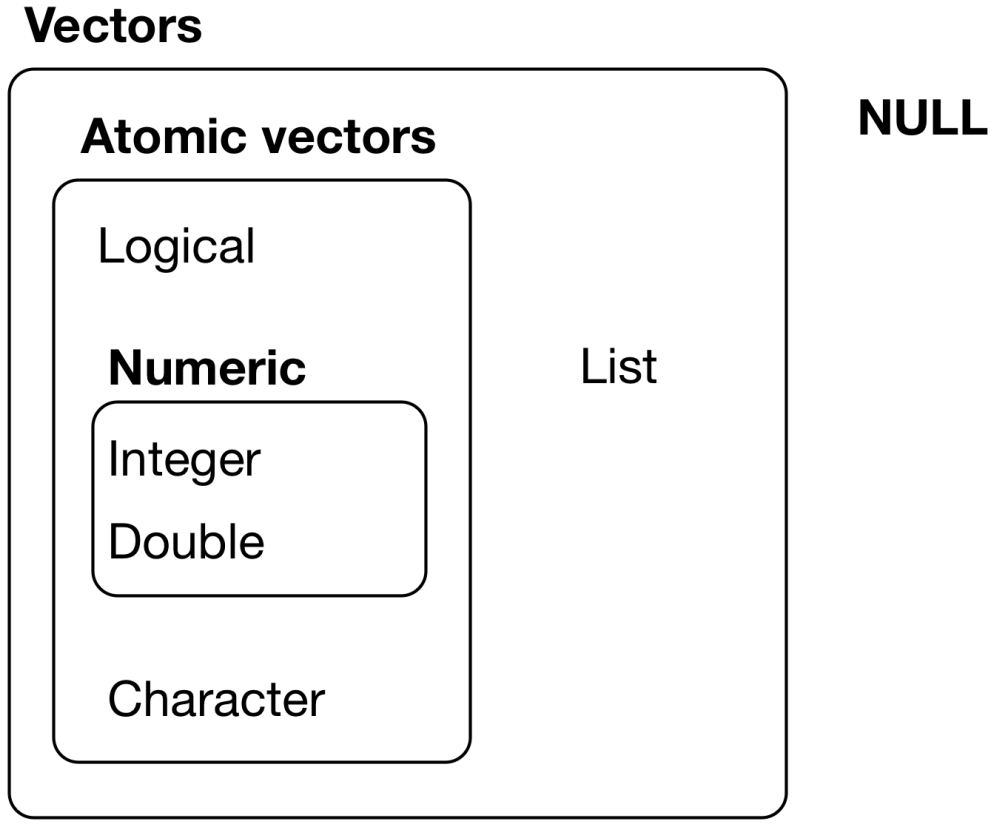
5 main built-in data structures in R:

- Atomic vector (`vector`)
- Matrix (`matrix`)
- Array (`array`)
- List (`list`)
- Data frame (`data.frame`)

Summary of data structures in R

Structure	Description	Dimensionality	Data Type
`vector`	Atomic vector (scalar)	1d	homogenous
`matrix`	Matrix	2d	homogenous
`array`	One-, two or n-dimensional array	1d/2d/nd	homogenous
`list`	List	1d	heterogeneous
`data.frame`	Rectangular data	2d	heterogeneous

Vectors in R



Source: R for Data Science

Atomic vectors

- *Vector* is the core building block of R
- R has no scalars (they are just vectors of length 1)
- Vectors can be created with `c()` function (short for **c**ombine)

Atomic vectors

- *Vector* is the core building block of R
- R has no scalars (they are just vectors of length 1)
- Vectors can be created with `c()` function (short for **c**ombine)

In [20]:
```
v <- c(8, 10, 12)
v
```

```
[1]  8 10 12
```

Atomic vectors

- *Vector* is the core building block of R
- R has no scalars (they are just vectors of length 1)
- Vectors can be created with `c()` function (short for **c**ombine)

In [20]:
```
v <- c(8, 10, 12)
v
```

```
[1]  8 10 12
```

In [21]:
```
v <- c(v, 14) # Vectors are always flattened (even when nested)
v
```

```
[1]  8 10 12 14
```

Data types

4 common data types that are contained in R structures:

- Character (`character`)
- Integer (`integer`)
- Double/numeric (`double` / `numeric`)
- Logical/boolean (`logical`)

Character vector

Character vector

In [22]:
```
char_vec <- c("apple", "banana", "watermelon")
```

Character vector

In [22]:
```
char_vec <- c("apple", "banana", "watermelon")
```

In [23]:
```
char_vec
```

```
[1] "apple"      "banana"     "watermelon"
```

Character vector

In [22]:
```
char_vec <- c("apple", "banana", "watermelon")
```

In [23]:
```
char_vec
```

```
[1] "apple"      "banana"     "watermelon"
```

In [24]:
```
# length() function gives the length of an R object (analogous to Pytho
length(char_vec)
```

```
[1] 3
```

Character vector

In [22]:
```
char_vec <- c("apple", "banana", "watermelon")
```

In [23]:
```
char_vec
```

```
[1] "apple"      "banana"     "watermelon"
```

In [24]:
```
# length() function gives the length of an R object (analogous to Pytho
length(char_vec)
```

```
[1] 3
```

In [25]:
```
is.character(char_vec)
```

```
[1] TRUE
```

Integer vector

Integer vector

In [26]:
```
# Note the 'L' suffix to make sure you get an integer rather than doubl
int_vec <- c(300L, 200L, 4L)
```

Integer vector

In [26]:
```r
# Note the 'L' suffix to make sure you get an integer rather than doubl
int_vec <- c(300L, 200L, 4L)
```

In [27]:
```r
int_vec
```

```
[1] 300 200   4
```

Integer vector

In [26]:
```
# Note the 'L' suffix to make sure you get an integer rather than doubl
int_vec <- c(300L, 200L, 4L)
```

In [27]:
```
int_vec
```

```
[1] 300 200   4
```

In [28]:
```
# typeof() function returns the type of an R object (analogous to Pytho
typeof(int_vec)
```

```
[1] "integer"
```

Integer vector

In [26]:
```
# Note the 'L' suffix to make sure you get an integer rather than doubl
int_vec <- c(300L, 200L, 4L)
```

In [27]:
```
int_vec
```

```
[1] 300 200   4
```

In [28]:
```
# typeof() function returns the type of an R object (analogous to Pytho
typeof(int_vec)
```

```
[1] "integer"
```

In [29]:
```
# is.integer() tests whether R object (vector/array/matrix) contains el
is.integer(int_vec)
```

```
[1] TRUE
```

Double vector

Double vector

In [30]:
```r
# Note that even without decimal part R treats these numbers as double
dbl_vec <- c(300, 200, 4)
```

Double vector

In [30]:
```r
# Note that even without decimal part R treats these numbers as double
dbl_vec <- c(300, 200, 4)
```

In [31]:
```r
dbl_vec
```

```
[1] 300 200   4
```

Double vector

In [30]:
```
# Note that even without decimal part R treats these numbers as double
dbl_vec <- c(300, 200, 4)
```

In [31]:
```
dbl_vec
```

```
[1] 300 200   4
```

In [32]:
```
typeof(dbl_vec)
```

```
[1] "double"
```

Double vector

In [30]:
```r
# Note that even without decimal part R treats these numbers as double
dbl_vec <- c(300, 200, 4)
```

In [31]:
```r
dbl_vec
```

```
[1] 300 200   4
```

In [32]:
```r
typeof(dbl_vec)
```

```
[1] "double"
```

In [33]:
```r
is.double(dbl_vec)
```

```
[1] TRUE
```

Double vector

In [30]:
```
# Note that even without decimal part R treats these numbers as double
dbl_vec <- c(300, 200, 4)
```

In [31]:
```
dbl_vec
```

```
[1] 300 200   4
```

In [32]:
```
typeof(dbl_vec)
```

```
[1] "double"
```

In [33]:
```
is.double(dbl_vec)
```

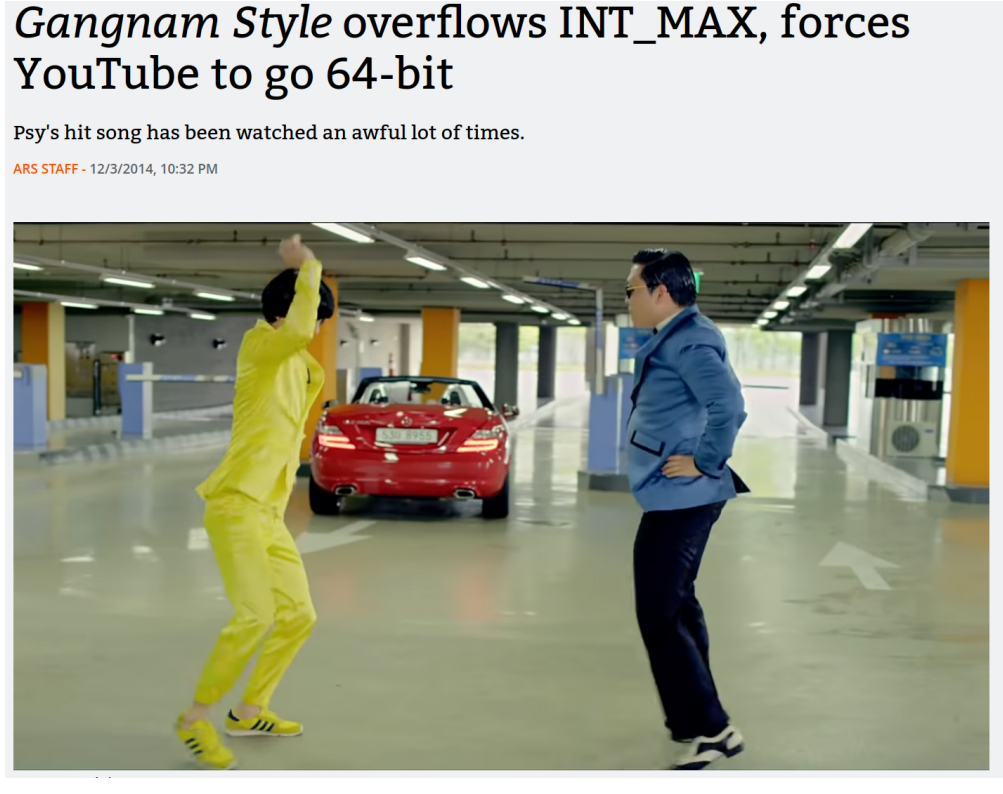
```
[1] TRUE
```

In [34]:
```
# Note that is.numeric() function is a generic way of testing whether \
# integers or double
is.numeric(int_vec)
```

```
[1] TRUE
```

Integer vs double

- Integers are used to store whole numbers (e.g. counts)
- 32-bit integer: $2^{32} = 4,294,967,296$
- Signed 32-bit integer: $[-2,147,483,648 \; .. \; 2,147,483,648]$



Gangnam Style overflows INT_MAX, forces YouTube to go 64-bit

Psy's hit song has been watched an awful lot of times.

ARS STAFF - 12/3/2014, 10:32 PM

Extra: More on integer overflow on YouTube

Logical vector

Logical vector

In [35]:
```
log_vec <- c(FALSE, FALSE, TRUE)
log_vec
```

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

Logical vector

In [35]:
```
log_vec <- c(FALSE, FALSE, TRUE)
log_vec
```

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

In [36]:
```
# While more concise, using T/F instead of TRUE/FALSE can be confusing
log_vec2 <- c(F, F, T)
log_vec2
```

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

Logical vector

In [35]:
```
log_vec <- c(FALSE, FALSE, TRUE)
log_vec
```

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

In [36]:
```
# While more concise, using T/F instead of TRUE/FALSE can be confusing
log_vec2 <- c(F, F, T)
log_vec2
```

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

In [37]:
```
typeof(log_vec)
```

```
[1] "logical"
```

Type coercion in vectors

- All elements of a vector must be of the same type
- If you try to combine vectors of different types, their elements will be *coerced* to the most flexible type

Type coercion in vectors

- All elements of a vector must be of the same type
- If you try to combine vectors of different types, their elements will be *coerced* to the most flexible type

In [38]:
```
# Note that logical vector get coerced to 0/1 for FALSE/TRUE
c(dbl_vec, log_vec)
```

```
[1] 300 200   4   0   0   1
```

Type coercion in vectors

- All elements of a vector must be of the same type
- If you try to combine vectors of different types, their elements will be *coerced* to the most flexible type

In [38]:
```
# Note that logical vector get coerced to 0/1 for FALSE/TRUE
c(dbl_vec, log_vec)
```

```
[1] 300 200   4   0   0   1
```

In [39]:
```
c(char_vec, int_vec)
```

```
[1] "apple"      "banana"     "watermelon" "300"        "200"
[6] "4"
```

Type coercion in vectors

- All elements of a vector must be of the same type
- If you try to combine vectors of different types, their elements will be *coerced* to the most flexible type

In [38]:
```
# Note that logical vector get coerced to 0/1 for FALSE/TRUE
c(dbl_vec, log_vec)
```

```
[1] 300 200   4   0   0   1
```

In [39]:
```
c(char_vec, int_vec)
```

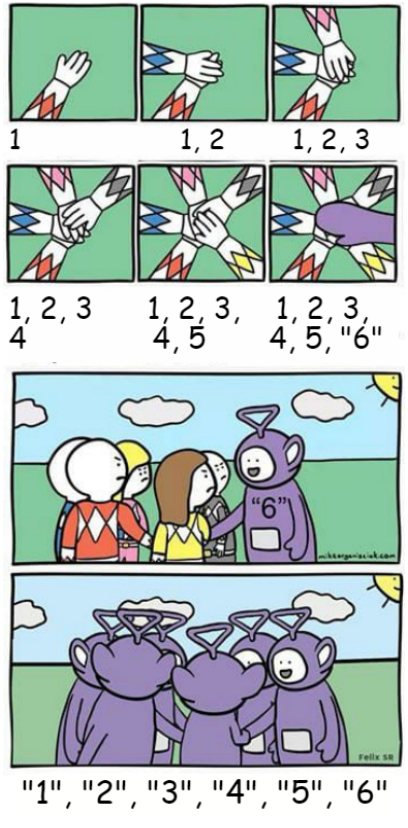
```
[1] "apple"      "banana"     "watermelon" "300"        "200"
[6] "4"
```

In [40]:
```
# If no natural way of type conversion exists, NAs are introduced
as.numeric(char_vec)
```

```
Warning message in eval(expr, envir, enclos):
“NAs introduced by coercion”
```

```
[1] NA NA NA
```

Implicit type coercion



Source: Twitter

NA and NULL values

- In Python we encountered `None` value
- R makes a distinction between:
    - `NA` - value exists, but is unknown (e.g. survey non-response)
    - `NULL` - object does not exist
- `NA`'s are defined for each data type (integer, character, numeric, etc.)

Extra: R Documentation on NA

NA and NULL example

NA and NULL example

In [41]:
```
na <- c(NA, NA, NA)
na
```

```
[1] NA NA NA
```

NA and NULL example

In [41]:
```
na <- c(NA, NA, NA)
na
```

```
[1] NA NA NA
```

In [42]:
```
length(na)
```

```
[1] 3
```

NA and NULL example

In [41]:
```
na <- c(NA, NA, NA)
na
```

```
[1] NA NA NA
```

In [42]:
```
length(na)
```

```
[1] 3
```

In [43]:
```
null <- c(NULL, NULL, NULL)
null
```

```
NULL
```

NA and NULL example

In [41]:
```
na <- c(NA, NA, NA)
na
```

```
[1] NA NA NA
```

In [42]:
```
length(na)
```

```
[1] 3
```

In [43]:
```
null <- c(NULL, NULL, NULL)
null
```

```
NULL
```

In [44]:
```
length(null)
```

```
[1] 0
```

Working with NAs

Working with NAs

In [45]:
```r
# Presence of NAs can lead to unexpected results
v_na <- c(1, 2, 3, NA, 5)
mean(v_na)
```

```
[1] NA
```

Working with NAs

In [45]:
```r
# Presence of NAs can lead to unexpected results
v_na <- c(1, 2, 3, NA, 5)
mean(v_na)
```

```
[1] NA
```

In [46]:
```r
# NAs should be treated specially
mean(v_na, na.rm = TRUE)
```

```
[1] 2.75
```

Working with NAs

In [45]:
```r
# Presence of NAs can lead to unexpected results
v_na <- c(1, 2, 3, NA, 5)
mean(v_na)
```

```
[1] NA
```

In [46]:
```r
# NAs should be treated specially
mean(v_na, na.rm = TRUE)
```

```
[1] 2.75
```

In [47]:
```r
# Remember NAs are missing values
# Thus result of comparing them is unknown
NA == NA
```

```
[1] NA
```

Working with NAs

In [45]:
```r
# Presence of NAs can lead to unexpected results
v_na <- c(1, 2, 3, NA, 5)
mean(v_na)
```

```
[1] NA
```

In [46]:
```r
# NAs should be treated specially
mean(v_na, na.rm = TRUE)
```

```
[1] 2.75
```

In [47]:
```r
# Remember NAs are missing values
# Thus result of comparing them is unknown
NA == NA
```

```
[1] NA
```

In [48]:
```r
# is.na() is a special function that checks whether value is missing (N
is.na(v_na)
```

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

Working with NAs

In [45]:
```r
# Presence of NAs can lead to unexpected results
v_na <- c(1, 2, 3, NA, 5)
mean(v_na)
```

```
[1] NA
```

In [46]:
```r
# NAs should be treated specially
mean(v_na, na.rm = TRUE)
```

```
[1] 2.75
```

In [47]:
```r
# Remember NAs are missing values
# Thus result of comparing them is unknown
NA == NA
```

```
[1] NA
```

In [48]:
```r
# is.na() is a special function that checks whether value is missing (N
is.na(v_na)
```

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

In [49]:
```r
# We can use such logical vectors for subsetting (more below)
v_na[!is.na(v_na)]
```

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

Vector indexing and subsetting

- Indexing in R starts from **1** (as opposed to 0 in Python)
- To subset a vector, use `[ ]` to index the elements you would like to select:

```
vector[index]
```

Vector indexing and subsetting

- Indexing in R starts from **1** (as opposed to 0 in Python)
- To subset a vector, use `[]` to index the elements you would like to select:

```
vector[index]
```

In [50]:
```
dbl_vec[1]
```

```
[1] 300
```

Vector indexing and subsetting

- Indexing in R starts from **1** (as opposed to 0 in Python)
- To subset a vector, use `[]` to index the elements you would like to select:

```
vector[index]
```

In [50]:
```
dbl_vec[1]
```

```
[1] 300
```

In [51]:
```
dbl_vec[c(1,3)]
```

```
[1] 300   4
```

Summary of vector subsetting

Value	Example	Description
Positive integers	`v[c(3, 1)]`	Returns elements at specified positions
Negative integers	`v[-c(3, 1)]`	Omits elements at specified positions
Logical vectors	`v[c(FALSE, TRUE)]`	Returns elements where corresponding logical value is `TRUE`
Character vector	`v[c("c", "a")]`	Returns elements with matching names (only for named vectors)
Nothing	`v[]`	Returns the original vector
0 (Zero)	`v[0]`	Returns a zero-length vector

Generating sequences for subsetting

- You can use `:` operator to generate vectors of indices for subsetting
- `seq()` function provides a generalization of `:` for generating arithemtic progressions

Generating sequences for subsetting

- You can use `:` operator to generate vectors of indices for subsetting
- `seq()` function provides a generalization of `:` for generating arithemtic progressions

In [52]:
```
2:4
```

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

Generating sequences for subsetting

- You can use `:` operator to generate vectors of indices for subsetting
- `seq()` function provides a generalization of `:` for generating arithemtic progressions

In [52]:
```
2:4
```

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

In [53]:
```
# It is similar to Python's object[start:stop:step] syntax
seq(from = 1, to = 4, by = 2)
```

```
[1] 1 3
```

Vector subsetting examples

Vector subsetting examples

In [54]:
```
v
```

```
[1]  8 10 12 14
```

Vector subsetting examples

In [54]:
```
v
```

```
[1]  8 10 12 14
```

In [55]:
```
v[2:4]
```

```
[1] 10 12 14
```

Vector subsetting examples

In [54]:
```
v
```

```
[1]  8 10 12 14
```

In [55]:
```
v[2:4]
```

```
[1] 10 12 14
```

In [56]:
```
# Argument names can be omitted for matching by position
v[seq(1,4,2)]
```

```
[1]  8 12
```

Vector subsetting examples

In [54]:
```
v
```

```
 [1]  8 10 12 14
```

In [55]:
```
v[2:4]
```

```
[1] 10 12 14
```

In [56]:
```
# Argument names can be omitted for matching by position
v[seq(1,4,2)]
```

```
 [1]  8 12
```

In [57]:
```
# All but the last element
v[-length(v)]
```

```
 [1]  8 10 12
```

Vector subsetting examples

In [54]:
```
v
```

```
 [1]  8 10 12 14
```

In [55]:
```
v[2:4]
```

```
[1] 10 12 14
```

In [56]:
```
# Argument names can be omitted for matching by position
v[seq(1,4,2)]
```

```
 [1]  8 12
```

In [57]:
```
# All but the last element
v[-length(v)]
```

```
 [1]  8 10 12
```

In [58]:
```
# Reverse order
v[seq(length(v),1,-1)]
```

```
 [1] 14 12 10  8
```

Vector recycling

For operations that require vectors to be of the same length R recycles (reuses) the shorter one

Vector recycling

For operations that require vectors to be of the same length R recycles (reuses) the shorter one

In [59]:
```
c(0, 1) + c(1, 2, 3, 4)
```

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

Vector recycling

For operations that require vectors to be of the same length R recycles (reuses) the shorter one

In [59]:
```
c(0, 1) + c(1, 2, 3, 4)
```

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

In [60]:
```
5 * c(1, 2, 3, 4)
```

```
[1]  5 10 15 20
```

Vector recycling

For operations that require vectors to be of the same length R recycles (reuses) the shorter one

In [59]:
```
c(0, 1) + c(1, 2, 3, 4)
```

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

In [60]:
```
5 * c(1, 2, 3, 4)
```

```
[1]  5 10 15 20
```

In [61]:
```
c(1, 2, 3, 4)[c(TRUE, FALSE)]
```

```
[1] 1 3
```

`which()` function

Returns indices of TRUE elements in a vector

`which()` function

Returns indices of TRUE elements in a vector

In [62]:
```
char_vec
```

```
[1] "apple"      "banana"     "watermelon"
```

`which()` function

Returns indices of TRUE elements in a vector

In [62]:
```
char_vec
```

```
[1] "apple"      "banana"     "watermelon"
```

In [63]:
```
char_vec == "watermelon"
```

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

`which()` function

Returns indices of TRUE elements in a vector

In [62]:
```
char_vec
```

```
[1] "apple"      "banana"     "watermelon"
```

In [63]:
```
char_vec == "watermelon"
```

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

In [64]:
```
which(char_vec == "watermelon")
```

```
[1] 3
```

`which()` function

Returns indices of TRUE elements in a vector

In [62]:
```
char_vec
```

```
[1] "apple"      "banana"     "watermelon"
```

In [63]:
```
char_vec == "watermelon"
```

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

In [64]:
```
which(char_vec == "watermelon")
```

```
[1] 3
```

In [65]:
```
dbl_vec[char_vec == "watermelon"]
```

```
[1] 4
```

`which()` function

Returns indices of TRUE elements in a vector

In [62]:
```
char_vec
```

```
[1] "apple"      "banana"     "watermelon"
```

In [63]:
```
char_vec == "watermelon"
```

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

In [64]:
```
which(char_vec == "watermelon")
```

```
[1] 3
```

In [65]:
```
dbl_vec[char_vec == "watermelon"]
```

```
[1] 4
```

In [66]:
```
dbl_vec[which(char_vec == "watermelon")]
```

```
[1] 4
```

Lists

- As opposed to vectors, *lists* can contain elements of any type
- List can also have nested lists within it
- Lists are constructed using `list()` function in R

Lists

- As opposed to vectors, *lists* can contain elements of any type
- List can also have nested lists within it
- Lists are constructed using `list()` function in R

In [67]:
```r
# We can combine different data types in a list and, optionally, name e
l <- list(2:4, "a", B = c(TRUE, FALSE, FALSE), list("x", 1L))
l
```

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

[[2]]
[1] "a"

$B
[1]  TRUE FALSE FALSE

[[4]]
[[4]][[1]]
[1] "x"

[[4]][[2]]
[1] 1
```

R object structure

- `str()` - one of the most useful functions in R
- It shows the **str**ucture of an arbitrary R object

R object structure

- `str()` - one of the most useful functions in R
- It shows the **str**ucture of an arbitrary R object

In [68]:
```
str(l)
```

```
List of 4
 $  : int [1:3] 2 3 4
 $  : chr "a"
 $ B: logi [1:3] TRUE FALSE FALSE
 $  :List of 2
  ..$ : chr "x"
  ..$ : int 1
```

List subsetting

- As with vectors you can use `[]` to subset lists
- This will return a list of length one
- Components of the list can be individually extracted using `[[` and `$` operators

```
list[index]
```

```
list[[index]]
```

```
list$name
```

List subsetting examples

List subsetting examples

In [69]:
```
l[3]
```

```
$B
[1]  TRUE FALSE FALSE
```

List subsetting examples

In [69]:
```
l[3]
```

```
$B
[1]  TRUE FALSE FALSE
```

In [70]:
```
str(l[3])
```

```
List of 1
 $ B: logi [1:3] TRUE FALSE FALSE
```

List subsetting examples

In [69]:
```
l[3]
```

```
$B
[1]  TRUE FALSE FALSE
```

In [70]:
```
str(l[3])
```

```
List of 1
 $ B: logi [1:3] TRUE FALSE FALSE
```

In [71]:
```
l[[3]]
```

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

List subsetting examples

In [69]:
```
l[3]
```

```
$B
[1]  TRUE FALSE FALSE
```

In [70]:
```
str(l[3])
```

```
List of 1
 $ B: logi [1:3] TRUE FALSE FALSE
```

In [71]:
```
l[[3]]
```

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

In [72]:
```
# Only works with named elements
l$B
```

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

Attributes

- All R objects can have attributes that contain metadata about them
- Attributes can be thought of as named lists
- Names, dimensions and class are common examples of attributes
- They (and some other) have special functions for getting and setting them
- More generally, attributes can be accessed and modified individually with `attr()` function

Attributes examples

Attributes examples

In [73]:
```
v
```

```
[1]  8 10 12 14
```

Attributes examples

In [73]:
```
v
```

```
[1]  8 10 12 14
```

In [74]:
```
attr(v, "example_attribute") <- "This is a vector"
```

Attributes examples

In [73]:
```
v
```

```
[1]  8 10 12 14
```

In [74]:
```
attr(v, "example_attribute") <- "This is a vector"
```

In [75]:
```
attr(v, "example_attribute")
```

```
[1] "This is a vector"
```

Attributes examples

In [73]:
```
v
```

```
 [1]  8 10 12 14
```

In [74]:
```
attr(v, "example_attribute") <- "This is a vector"
```

In [75]:
```
attr(v, "example_attribute")
```

```
 [1] "This is a vector"
```

In [76]:
```
# To set names for vector elements we can use names() function
names(v) <- c("a", "b", "c", "d")
v
```

```
 a  b  c  d
 8 10 12 14
attr(,"example_attribute")
[1] "This is a vector"
```

Attributes examples

In [73]:
```
v
```

```
 [1]  8 10 12 14
```

In [74]:
```
attr(v, "example_attribute") <- "This is a vector"
```

In [75]:
```
attr(v, "example_attribute")
```

```
 [1] "This is a vector"
```

In [76]:
```
# To set names for vector elements we can use names() function
names(v) <- c("a", "b", "c", "d")
v
```

```
  a  b  c  d
  8 10 12 14
attr(,"example_attribute")
[1] "This is a vector"
```

In [77]:
```
# Names of vector elements can be used for subsetting
v["b"]
```

```
  b
 10
```

Factors

- Factors form the basis of categorical data analysis in R
- Values of nominal (categorical) variables represent categories rather than numeric data
- Examples are abundant in social sciences (gender, party, region, etc.)
- Internally, in R factor variables are represented by integer vectors
- With 2 additional attributes:
    - `class()` attribute which is set to `factor`
    - `levels()` attribute which defines allowed values

Factors example

Factors example

In [78]:
```
cities <- c("Dublin", "Cork", "Cork", "Limerick", "Galway")
cities
```

```
[1] "Dublin"   "Cork"     "Cork"     "Limerick" "Galway"
```

Factors example

In [78]:
```
cities <- c("Dublin", "Cork", "Cork", "Limerick", "Galway")
cities
```

```
[1] "Dublin"   "Cork"     "Cork"     "Limerick" "Galway"
```

In [79]:
```
typeof(cities)
```

```
[1] "character"
```

Factors example

In [78]:
```
cities <- c("Dublin", "Cork", "Cork", "Limerick", "Galway")
cities
```

```
[1] "Dublin"   "Cork"     "Cork"     "Limerick" "Galway"
```

In [79]:
```
typeof(cities)
```

```
[1] "character"
```

In [80]:
```
# We use factor() function to convert character vector into factor
# Only unique elements of character vector are considered as a level
cities <- factor(cities)
cities
```

```
[1] Dublin   Cork     Cork     Limerick Galway
Levels: Cork Dublin Galway Limerick
```

Factors example

In [78]:
```
cities <- c("Dublin", "Cork", "Cork", "Limerick", "Galway")
cities
```

```
[1] "Dublin"   "Cork"     "Cork"     "Limerick" "Galway"
```

In [79]:
```
typeof(cities)
```

```
[1] "character"
```

In [80]:
```
# We use factor() function to convert character vector into factor
# Only unique elements of character vector are considered as a level
cities <- factor(cities)
cities
```

```
[1] Dublin   Cork     Cork     Limerick Galway
Levels: Cork Dublin Galway Limerick
```

In [81]:
```
class(cities)
```

```
[1] "factor"
```

Factors example

In [78]:
```
cities <- c("Dublin", "Cork", "Cork", "Limerick", "Galway")
cities
```

```
[1] "Dublin"   "Cork"     "Cork"     "Limerick" "Galway"
```

In [79]:
```
typeof(cities)
```

```
[1] "character"
```

In [80]:
```
# We use factor() function to convert character vector into factor
# Only unique elements of character vector are considered as a level
cities <- factor(cities)
cities
```

```
[1] Dublin   Cork     Cork     Limerick Galway
Levels: Cork Dublin Galway Limerick
```

In [81]:
```
class(cities)
```

```
[1] "factor"
```

In [82]:
```
# Note that the data type of this vector is integer (and not character)
typeof(cities)
```

```
[1] "integer"
```

Factors example continued

Factors example continued

In [83]:
```
# Note that R automatically sorted the categories alphabetically
levels(cities)
```

```
[1] "Cork"     "Dublin"   "Galway"   "Limerick"
```

Factors example continued

In [83]:
```r
# Note that R automatically sorted the categories alphabetically
levels(cities)
```

```
[1] "Cork"    "Dublin"   "Galway"   "Limerick"
```

In [84]:
```r
# You can change the reference category using relevel() function
cities <- relevel(cities, ref = "Dublin")
levels(cities)
```

```
[1] "Dublin"   "Cork"     "Galway"   "Limerick"
```

Factors example continued

In [83]:
```
# Note that R automatically sorted the categories alphabetically
levels(cities)
```

```
[1] "Cork"     "Dublin"   "Galway"   "Limerick"
```

In [84]:
```
# You can change the reference category using relevel() function
cities <- relevel(cities, ref = "Dublin")
levels(cities)
```

```
[1] "Dublin"   "Cork"     "Galway"   "Limerick"
```

In [85]:
```
# Or define an arbitrary ordering of levels using levels argument in fa
cities <- factor(cities, levels = c("Limerick", "Galway", "Dublin", "Co
levels(cities)
```

```
[1] "Limerick" "Galway"   "Dublin"   "Cork"
```

Factors example continued

In [83]:
```
# Note that R automatically sorted the categories alphabetically
levels(cities)
```

```
[1] "Cork"    "Dublin"   "Galway"   "Limerick"
```

In [84]:
```
# You can change the reference category using relevel() function
cities <- relevel(cities, ref = "Dublin")
levels(cities)
```

```
[1] "Dublin"   "Cork"     "Galway"   "Limerick"
```

In [85]:
```
# Or define an arbitrary ordering of levels using levels argument in fa
cities <- factor(cities, levels = c("Limerick", "Galway", "Dublin", "Co
levels(cities)
```

```
[1] "Limerick" "Galway"   "Dublin"   "Cork"
```

In [86]:
```
# Under the hood factors continue to be integer vectors
as.integer(cities)
```

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

Tabulation

- `table()` function is very useful for describing discrete data.
- It can be used for:
    - tabulating a single variable
    - creating contingency tables (crosstabs).
- Implicitly, R treats tabulated variables as factors.

Tabulation

- `table()` function is very useful for describing discrete data.
- It can be used for:
    - tabulating a single variable
    - creating contingency tables (crosstabs).
- Implicitly, R treats tabulated variables as factors.

In [87]:
```r
var_1 <- sample(c("a", "b", "c"), size = 50, replace = TRUE)
var_2 <- sample(c(1, 2, 3), size = 50, replace = TRUE)
```

Tabulation

- `table()` function is very useful for describing discrete data.
- It can be used for:
    - tabulating a single variable
    - creating contingency tables (crosstabs).
- Implicitly, R treats tabulated variables as factors.

In [87]:
```
var_1 <- sample(c("a", "b", "c"), size = 50, replace = TRUE)
var_2 <- sample(c(1, 2, 3), size = 50, replace = TRUE)
```

In [88]:
```
table(var_1, var_2)
```

```
     var_2
var_1 1 2 3
    a 7 4 5
    b 6 6 5
    c 7 2 8
```

Factors in crosstabs

Factors in crosstabs

In [89]:
```
var_2 <- factor(var_2, levels = c(3, 1, 2))
```

Factors in crosstabs

In [89]:
```
var_2 <- factor(var_2, levels = c(3, 1, 2))
```

In [90]:
```
table(var_2)
```

```
var_2
 3  1  2
18 20 12
```

Factors in crosstabs

In [89]:
```
var_2 <- factor(var_2, levels = c(3, 1, 2))
```

In [90]:
```
table(var_2)
```

```
var_2
 3  1  2
18 20 12
```

In [91]:
```
var_2 <- factor(var_2, levels = c(3, 1, 2), labels = c("Three", "One",
```

Factors in crosstabs

In [89]:
```
var_2 <- factor(var_2, levels = c(3, 1, 2))
```

In [90]:
```
table(var_2)
```

```
var_2
 3  1  2
18 20 12
```

In [91]:
```
var_2 <- factor(var_2, levels = c(3, 1, 2), labels = c("Three", "One",
```

In [92]:
```
table(var_1, var_2)
```

```
     var_2
var_1 Three One Two
    a     5   7   4
    b     5   6   6
    c     8   7   2
```

Arrays and matrices

- Arrays are vectors with an added class and dimensionality attribute
- These attributes can be accessed using `class()` and `dim()` functions
- Arrays can have an arbitrary number of dimensions
- Matrices are special cases of arrays that have just two dimensions
- Arrays and matrices can be created using `array()` and `matrix()` functions
- Or by adding dimension attribute with `dim()` function

Array example

Array example

In [93]:
```r
# : operator can be used generate vectors of sequential numbers
a <- 1:12
a
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Array example

In [93]:
```
# : operator can be used generate vectors of sequential numbers
a <- 1:12
a
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

In [94]:
```
class(a)
```

```
[1] "integer"
```

Array example

In [93]:
```
# : operator can be used generate vectors of sequential numbers
a <- 1:12
a
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

In [94]:
```
class(a)
```

```
[1] "integer"
```

In [95]:
```
dim(a) <- c(3, 2, 2)
a
```

```
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    7   10
```

```
     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

Array example

In [93]:
```
# : operator can be used generate vectors of sequential numbers
a <- 1:12
a
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

In [94]:
```
class(a)
```

```
[1] "integer"
```

In [95]:
```
dim(a) <- c(3, 2, 2)
a
```

```
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    7   10
```

```
[2,]    8   11
[3,]    9   12
```

In [96]:

```
class(a)
```

```
[1] "array"
```

Matrix example

Matrix example

In [97]:
```
m <- 1:12
```

Matrix example

In [97]:
```
m <- 1:12
```

In [98]:
```
dim(m) <- c(3, 4)
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

Matrix example

In [97]:
```
m <- 1:12
```

In [98]:
```
dim(m) <- c(3, 4)
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

In [99]:
```
# Alternatively, we could use matrix() function
m <- matrix(1:12, nrow = 3, ncol = 4)
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

Matrix example

In [97]:
```
m <- 1:12
```

In [98]:
```
dim(m) <- c(3, 4)
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

In [99]:
```
# Alternatively, we could use matrix() function
m <- matrix(1:12, nrow = 3, ncol = 4)
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

In [100]:
```
# Note that length() function displays the length of underlying vector
length(m)
```

```
[1] 12
```

Array and matrix subsetting

- Subsetting higher-dimensional (> 1) structures is a generalisation of vector subsetting
- But, since they are built upon vectors there is a nuance (albeit uncommon)
- They are usually subset in 2 ways:
    - with multiple vectors, where each vector is a sequence of elements in that dimension
    - with 1 vector, in which case subsetting happens from the underlying vector

```
array[vector_1, vector_2, ..., vector_n]
```

```
array[vector]
```

Array subsetting example

Array subsetting example

In [101]:
```
a
```

```
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

Array subsetting example

In [101]:
```
a
```

```
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

In [102]:
```
# Most common way
a[1, 2, 2]
```

```
[1] 10
```

Array subsetting example

In [101]:
```
a
```

```
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

In [102]:
```
# Most common way
a[1, 2, 2]
```

```
[1] 10
```

In [103]:
```
# Specifying drop = FALSE after indices retains the original dimensiona
a[1, 2, 2, drop = FALSE]
```

```
, , 1
```

```
     [,1]
[1,]   10
```

Array subsetting example

In [101]:
```
a
```

```
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

In [102]:
```
# Most common way
a[1, 2, 2]
```

```
[1] 10
```

In [103]:
```
# Specifying drop = FALSE after indices retains the original dimensiona
a[1, 2, 2, drop = FALSE]
```

```
, , 1
```

```
     [,1]
[1,]   10
```

In [104]:

```r
# Here elements are subset from underlying vector (with repetition)
a[c(1, 2, 2)]
```

```
[1] 1 2 2
```

Matrix subsetting example

Matrix subsetting example

In [105]:
```
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

Matrix subsetting example

In [105]:
```
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

In [106]:
```
# As with arrays drop = FALSE prevents from this object being collapsed
m[, 1, drop = FALSE]
```

```
     [,1]
[1,] 1
[2,] 2
[3,] 3
```

Matrix subsetting example

In [105]:
```
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

In [106]:
```
# As with arrays drop = FALSE prevents from this object being collapsed
m[, 1, drop = FALSE]
```

```
     [,1]
[1,] 1
[2,] 2
[3,] 3
```

In [107]:
```
# Subset all rows, first two columns
m[1:nrow(m), 1:2]
```

```
     [,1] [,2]
[1,] 1    4
[2,] 2    5
[3,] 3    6
```

Matrix subsetting example

In [105]:
```
m
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
```

In [106]:
```
# As with arrays drop = FALSE prevents from this object being collapsed
m[, 1, drop = FALSE]
```

```
     [,1]
[1,] 1
[2,] 2
[3,] 3
```

In [107]:
```
# Subset all rows, first two columns
m[1:nrow(m), 1:2]
```

```
     [,1] [,2]
[1,] 1    4
[2,] 2    5
[3,] 3    6
```

In [108]:
```
# Note that vector recycling also applies here
m[c(TRUE, FALSE), -3]
```

```
     [,1] [,2] [,3]
[1,] 1    4    10
[2,] 3    6    12
```

R packages

- R's flexibility comes from its rich package ecosystem
- Comprehensive R Archive Network (CRAN) is the official repository of R packages
- At the moment it contains > 18K external packages
- Use `install.packages(<package_name>)` function to install packages that were released on CRAN
- Check `devtools` package if you need to install a package from other sources (e.g. GitHub, Bitbucket, etc.)
- Type `library(<package_name>)` to load installed packages

Help!

R has an inbuilt help facility which provides more information about any function:

Help!

R has an inbuilt help facility which provides more information about any function:

In [109]:
```
?length
```

Help!

R has an inbuilt help facility which provides more information about any function:

In [109]:
```
?length
```

In [110]:
```
help(dim)
```

Help!

R has an inbuilt help facility which provides more information about any function:

In [109]:
```
?length
```

In [110]:
```
help(dim)
```

- The quality of documentation varies a lot across packages.
- Stackoverflow is a good resource for many standard tasks.
- For custom packages it is often helpful to check the issues page on the GitHub.
- E.g. for `ggplot2`: https://github.com/tidyverse/ggplot2/issues
- Or, indeed, any search engine #LMDDGTFY

Next

- Tutorial: R objects, attributes and subsetting
- Next week: Control flow and functions in R