Week 2: R Basics

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

Overview

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

Introduction to R

R Background

University of Auckland

The R Project for Statistical Computing

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

R Overview

- 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
# Generate 5 random numbers from a standard normal distribution
rnorm(5)
```

```
[1]  0.5935632  0.7220500 -2.1026584 -0.6814161  0.8405273
```

Operations

Operators

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

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

Mathematical Operations

Arithmetic operations in R:

```
1 + 1
```

```
[1] 2
```

```
5 - 3
```

```
[1] 2
```

```
6 / 2
```

```
[1] 3
```

```
4 * 4
```

```
[1] 16
```

```
## Exponentiation, note that 2 ** 4 also works, but is not recommended
2 ^ 4
```

```
[1] 16
```

Advanced mathematical operations:

```
# Integer division
7 %/% 3
```

```
[1] 2
```

```
# Modulo operation (remainder of division)
7 %% 3
```

```
[1] 1
```

Logical Operations

```r
3 != 1 # Not equal
```

```
[1] TRUE
```

```r
3 > 3 # Greater than
```

```
[1] FALSE
```

```r
# OR - TRUE if either first or second operand is TRUE, FALSE otherwise
FALSE | TRUE
```

```
[1] TRUE
```

```r
# R also treats F and T as Boolean, but
# it is not recommended due to poor legibility
F | T
```

```
[1] TRUE
```

```r
# Longer form '&&' (AND) and '||' (OR) evaluate
# to a single value and are preferable in programming control flow
FALSE && TRUE
```

```
[1] FALSE
```

```r
# Shorter form '|' ('&') performs element-wise comparison
# and is often used in vectorised operations
c(FALSE, TRUE) | c(FALSE, FALSE)
```

```
[1] FALSE  TRUE
```

```r
# It is the only form appropriate for vector comparison
# (starting from R version 4.3.0)
c(FALSE, TRUE) || c(FALSE, FALSE)
```

```
Error in c(FALSE, TRUE) || c(FALSE, FALSE): 'length = 2' in coercion to 'logical(1)'
```

Operator Precedence

Order	Operator	Description
1	:: :::	access variables in a namespace
2	$ @	component / slot extraction
3	[[[	indexing
4	^	exponentiation (right to left)
5	- +	unary minus and plus
6	:	sequence operator
7	%any% |>	special operators (including %% and %/%)
8	* /	multiply, divide
9	+ -	(binary) add, subtract
10	< > <= >= == !=	ordering and comparison
11	!	negation
12	& &&	and
13	| ||	or
14	~	as in formulae
15	-> ->>	rightwards assignment
16	<- <<-	assignment (right to left)
17	=	assignment (right to left)
18	?	help (unary and binary)

Extra

R Documentation on Operator Syntax and Precedence

Operator Precedence: Examples

```
# Effectively, 5 > (3 * 2)
5 > 3 * 2
```

```
[1] FALSE
```

```
(5 > 3) * 2
```

```
[1] 2
```

```
# Effectively, 3 + (2 ^ 1)/2
3 + 2 ^ 1/2
```

```
[1] 4
```

```
3 + 2 ^ (1/2)
```

```
[1] 4.414214
```

Assignment

R Objects

> Everything that exists in R 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 Operation

- **Assignment** is the most important operation in R.
- It is used to *bind* an object to a name.
- `<-` is the standard assignment operator in R.
- While `=` is also supported, it is not recommended.

```
1 x <- 3
```

```
1 x
```

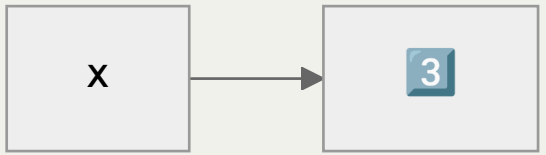
```
[1] 3
```

> **Extra**
>
> R Documentation on Assignment

Objects and Names

```
1 x <- 3
```

- The way to read this is
  - “create an object (numeric vector of length 1) with element 3 and bind it to the name x”.
- Thus, assignment operation does 2 things:
  - Creates an object.
  - Binds it to a name.



Memory Address

Aliases

- Creating (binding) another name (**alias**) to an object does not create a copy of the object.

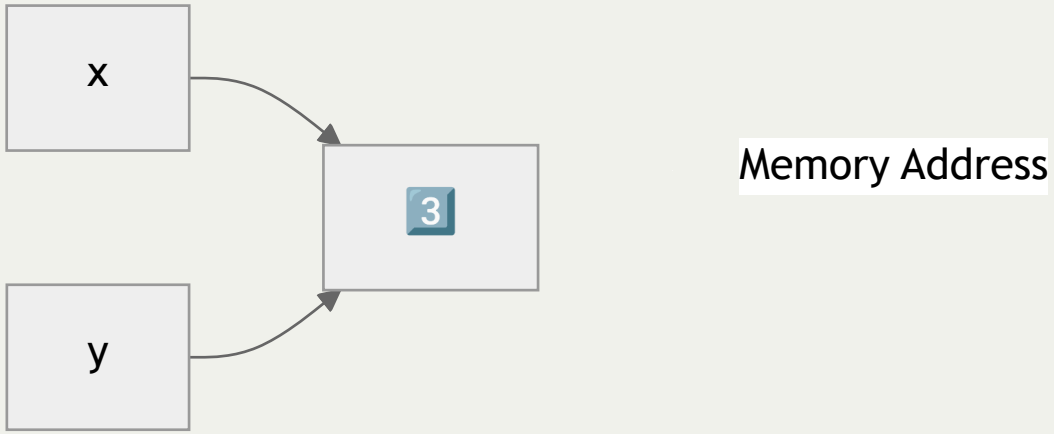
```
1 # We can use tracemem() function to trace the memory address of an object
2 tracemem(x)
```

```
[1] "<0x57602b2098f8>"
```

```
1 y <- x
```

```
1 tracemem(y)
```

```
[1] "<0x57602b2098f8>"
```



Copies

- R will create a copy of an object if the original object is modified.
- This is also known as **copy-on-modify** semantics.
- While it might seem innocuous, it can have implications for large objects.

```
1 x <- 5
```

```
1 # Pointing to the original object
2 tracemem(y)
```

```
[1] "<0x57602b2098f8>"
```

```
1 # Pointing to a new object
2 tracemem(x)
```

```
[1] "<0x57602dadf4f0>"
```

Vector

Dimensionality & Homogeneity

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

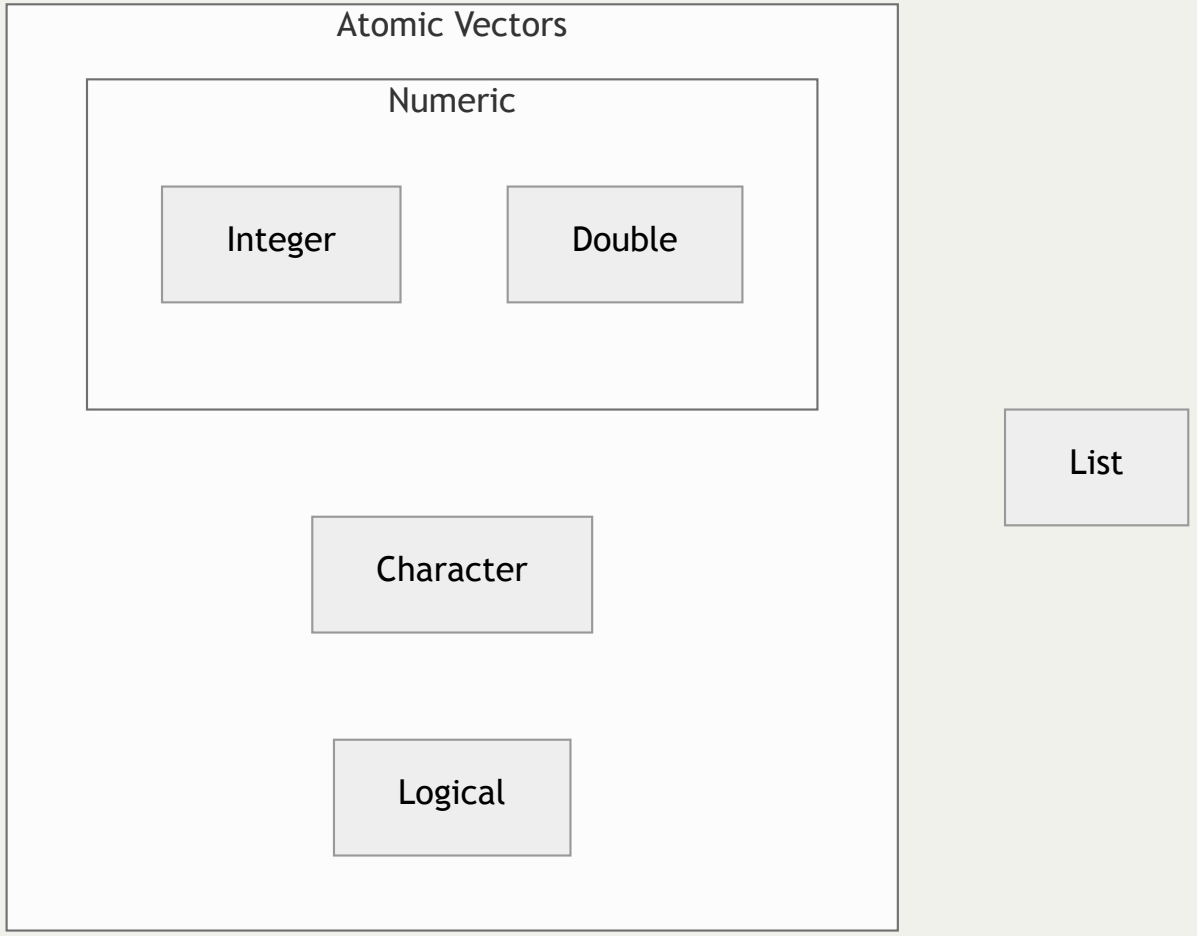
5 main built-in data structures in R:

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

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


Atomic Vectors
Numeric
Integer
Double
Character
Logical
List

Atomic Vectors

- **Vector** is the core building block of R
- Vectors can be created with `c()` function (short for **c**ombine)

```r
v <- c(1, 2, 3)
v
```

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

```r
v <- c(v, 4) # Vectors are always flattened (even when nested)
v
```

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

Scalars

- R has no scalars.
- Single values are just vectors of length 1.

```
1[1]
```

```
[1] 1
```

Data Types

Main Data Types

4 common data types that are contained in R structures:

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

Character Vector

```
1 char_vec <- c("apple", "banana", "watermelon")
```

```
1 char_vec
```

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

```
1 # length() function gives the length of an R object
2 length(char_vec)
```

```
[1] 3
```

```
1 # typeof() function returns the type of an R object
2 typeof(char_vec)
```

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

```
1 # is.character() tests whether R object (vector/array/matrix)
2 # contains elements of type 'character'
3 is.character(char_vec)
```

```
[1] TRUE
```

Double Vector

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

```r
dbl_vec
```

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

```r
typeof(dbl_vec)
```

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

```r
is.double(dbl_vec)
```

```
[1] TRUE
```

Integer Vector

```
# Note the 'L' suffix to make sure you get an integer rather than double
int_vec <- c(300L, 200L, 4L)
```

```
int_vec
```

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

```
typeof(int_vec)
```

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

```
is.integer(int_vec)
```

```
[1] TRUE
```

```
# Note that is.numeric() function is a generic way of testing
# whether vector contains numbers - either integers or double
is.numeric(int_vec)
```

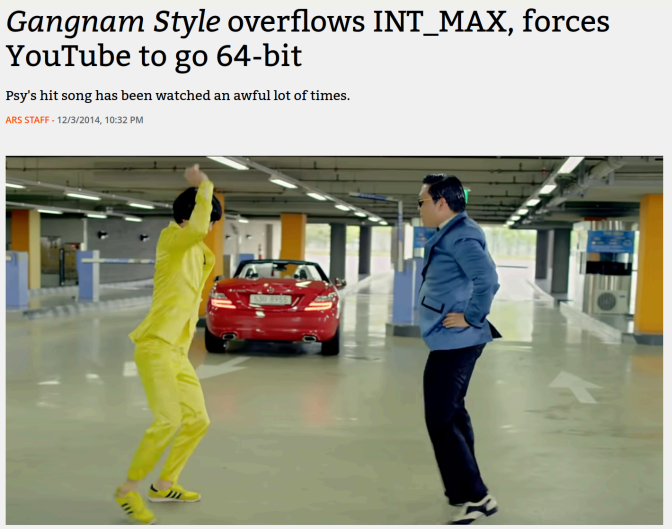
```
[1] TRUE
```

```
is.numeric(dbl_vec)
```

```
[1] TRUE
```

Integer vs Double

- Integers are used to store whole numbers (e.g. counts)
- Unsigned 32-bit integer: $2^{32} - 1 = 4,294,967,295$
  - Signed 32-bit integer: $[-2,147,483,648 \; .. \; 2,147,483,647]$
- Unsigned 64-bit integer: $2^{64} - 1 = 18,446,744,073,709,551,615$



Extra

More on integer overflow on YouTube

Logical Vector

```r
log_vec <- c(FALSE, FALSE, TRUE)
log_vec
```

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

```r
# 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
```

```r
typeof(log_vec)
```

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

```r
is.logical(log_vec)
```

```
[1] TRUE
```

Type Coercion

- 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.

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

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

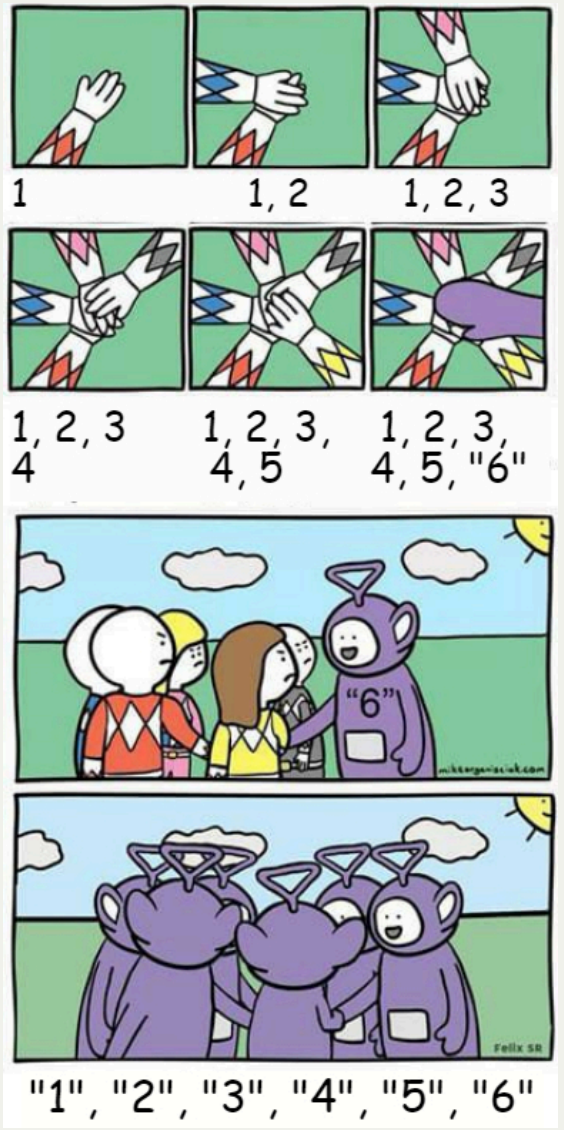
```
c(char_vec, int_vec)
```

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

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

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

Implicit Type Coercion



Twitter (X)

NA and NULL

- R makes a distinction between:
  - `NA` - value exists, but is unknown (e.g. survey non-response)
  - `NULL` - object does not exist
- `NA`'s are used in data sets (missing data).
- `NULL`'s are used in function calls (optional arguments).

```
1 NA
```

```
[1] NA
```

```
1 NULL
```

```
NULL
```

Extra

R Documentation on NA

NA and NULL: Example

```
1 na <- c(NA, NA, NA)
2 na
```

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

```
1 length(na)
```

```
[1] 3
```

```
1 null <- c(NULL, NULL, NULL)
2 null
```

```
NULL
```

```
1 length(null)
```

```
[1] 0
```

Working with NAs

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

```
[1] NA
```

```r
# NAs should be treated specially
mean(v_na, na.rm = TRUE)
```

```
[1] 2.75
```

```r
# Remember NAs are missing values
# Thus result of comparing them is unknown
NA == NA
```

```
[1] NA
```

```r
# is.na() is a special function that checks whether value is missing (NA)
is.na(v_na)
```

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

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

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

Subsetting

Vector Indexing and Subsetting

- Indexing in R starts from **1**.
- To subset a vector, use `[]` to index the elements you would like to select.
- If you would like to select only a single element you can also use `[[]]`.

```
vector[index]
```

```
vector[[index]]
```

```
dbl_vec[[1]]
```

```
[1] 300
```

```
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

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

```
1 2:4
```

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

```
1 seq(from = 1, to = 4, by = 2)
```

```
[1] 1 3
```

Vector Subsetting: Examples

```r
v
```

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

```r
v[2:4]
```

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

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

```
[1] 1 3
```

```r
# All but the last element
v[-length(v)]
```

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

```r
# Reverse order
v[seq(length(v), 1, -1)]
```

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

Vector Recycling

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

```
c(0, 1) + c(1, 2, 3, 4)
```

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

```
5 * c(1, 2, 3, 4)
```

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

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

```
[1] 1 3
```

`which()` function

Returns indices of TRUE elements in a vector.

```
1 char_vec
```

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

```
1 char_vec == "watermelon"
```

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

```
1 which(char_vec == "watermelon")
```

```
[1] 3
```

```
1 dbl_vec[char_vec == "watermelon"]
```

```
[1] 4
```

```
1 dbl_vec[which(char_vec == "watermelon")]
```

```
[1] 4
```

Membership Operation

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

```
"a" %in% "abc" # Note that R strings are not sequences
```

```
[1] FALSE
```

```
3 %in% c(1, 2, 3) # c(1, 2, 3) is a vector
```

```
[1] TRUE
```

```
!(3 %in% c(1, 2, 3))
```

```
[1] FALSE
```

Lists

Lists

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

```r
# We can combine different data types in a list and, optionally, name elements.
l <- list(
  "linear regression",
  model = "y ~ x",
  coefs = c(1.2, 0.5),
  list(
    x = 1:10,
    y = 1.2 + 1:10 * 0.5
  )
)
l
```

```
[[1]]
[1] "linear regression"

$model
[1] "y ~ x"

$coefs
[1] 1.2 0.5
```

```
[[4]]
[[4]]$x
 [1]  1  2  3  4  5  6  7  8  9 10

[[4]]$y
 [1] 1.7 2.2 2.7 3.2 3.7 4.2 4.7 5.2 5.7 6.2
```

R Object Structure

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

```
1 str(l)
```

```
List of 4
 $      : chr "linear regression"
 $ model: chr "y ~ x"
 $ coefs: num [1:2] 1.2 0.5
 $      :List of 2
  ..$ x: int [1:10] 1 2 3 4 5 6 7 8 9 10
  ..$ y: num [1:10] 1.7 2.2 2.7 3.2 3.7 4.2 4.7 5.2 5.7 6.2
```

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
```

```
1 l[3]
```

```
$coefs
[1] 1.2 0.5
```

```
1 str(l[3])
```

```
List of 1
 $ coefs: num [1:2] 1.2 0.5
```

```
1 l[[3]]
```

```
[1] 1.2 0.5
```

```
1 # Only works with named elements
2 l$coefs
```

```
[1] 1.2 0.5
```

Attributes

Attributes

- All R objects can have attributes associated with them.
- Attributes contain metadata that can be attached to any R object.
- Technically, 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.

Attributes: Examples

```r
v <- c(1, 2, 3, 4)
```

```r
# Setting attributes as a named list
attributes(v) <- list(some_info = "This is a vector")
```

```r
attributes(v)
```

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

```r
# Setting individual attributes
attr(v, "more_info") <- "This vector contains numbers"
```

```r
attr(v, "more_info")
```

```
[1] "This vector contains numbers"
```

```r
attributes(v)
```

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

$more_info
[1] "This vector contains numbers"
```

```r
# To set names for vector elements we can use names() function
names(v) <- c("a", "b", "c", "d")
v
```

```
a b c d
1 2 3 4
attr(,"some_info")
[1] "This is a vector"
attr(,"more_info")
[1] "This vector contains numbers"
```

Factors

- Factors form the basis of categorical data analysis in R
- Values of nominal 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

```r
gender <- c("male", "female", "female", "non-binary", "male")
gender
```

```
[1] "male"       "female"     "female"     "non-binary" "male"
```

```r
typeof(gender)
```

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

```r
# We use factor() function to convert character vector into factor
# Only unique elements of character vector are considered as a level
gender <- factor(gender)
gender
```

```
[1] male       female     female     non-binary male
Levels: female male non-binary
```

```r
class(gender)
```

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

```r
# Note that the data type of this vector is integer (and not character)
typeof(gender)
```

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

Factors: Example Continued

```r
# Note that R automatically sorted the categories alphabetically
levels(gender)
```

```
[1] "female"     "male"       "non-binary"
```

```r
# You can change the reference category using relevel() function
gender <- relevel(gender, ref = "male")
levels(gender)
```

```
[1] "male"       "female"     "non-binary"
```

```r
# Or define an arbitrary ordering of levels
# using 'levels' argument in factor() function
gender <- factor(gender, levels = c("non-binary", "male", "female"))
levels(gender)
```

```
[1] "non-binary" "male"       "female"
```

```r
# Under the hood factors continue to be integer vectors
as.integer(gender)
```

```
[1] 2 3 3 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.

```r
var_1 <- sample(c("female", "male", "non-binary"), size = 50, replace = TRUE)
var_2 <- sample(c(-1, 0, 1), size = 50, replace = TRUE)
```

```r
table(var_1, var_2)
```

```
            var_2
var_1        -1  0  1
  female      4  3 11
  male        6 10  6
  non-binary  3  3  4
```

Factors in Crosstabs

```r
var_2 <- factor(
  var_2,
  levels = c(0, -1, 1)
)
```

```r
table(var_2)
```

```
var_2
 0 -1  1
16 13 21
```

```r
var_2 <- factor(
  var_2,
  levels = c(0, -1, 1),
  labels = c("centre", "left", "right")
)
```

```r
table(var_1, var_2)
```

```
             var_2
var_1         centre left right
  female           3    4    11
  male            10    6     6
  non-binary       3    3     4
```

Arrays

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

```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
```

```r
class(a)
```

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

```r
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
```

```r
class(a)
```

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

Matrix: Example

```r
m <- 1:12
```

```r
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
```

```r
# 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
```

```r
# 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

```
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
```

```
# Most common way
a[1, 2, 2]
```

```
[1] 10
```

```
# Specifying drop = FALSE after indices retains the original dimensionality of matrix/array
a[1, 2, 2, drop = FALSE]
```

```
, , 1

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

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

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

Matrix Subsetting: Example

```
m
```

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

```
# As with arrays drop = FALSE prevents from this object being collapsed into 1-dimensional vector
m[, 1, drop = FALSE]
```

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

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

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

```
# 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 ~20K 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:

```
?length
```

```
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 in R