# Data types and functions

## Programming for Statistical Science

Shawn Santo

# Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Companion videos

- More on atomic vectors
- Generic vectors
- Introduction to functions
- More on functions

Videos were created for STA 323 & 523 - Summer 2020

Additional resources

- Section 3.5 Advanced R
- Section 3.7 Advanced R
- Chapter 6 Advanced R

# Recall

# Vectors

The fundamental building block of data in R is a vector (collections of related values, objects, other data structures, etc).

R has two types of vectors:

- **atomic** vectors
    - homogeneous collections of the *same* type (e.g. all logical values, all numbers, or all character strings).
- **generic** vectors
    - heterogeneous collections of *any* type of R object, even other lists (meaning they can have a hierarchical/tree-like structure).

I will use the term component or element when referring to a value inside a vector.

# Atomic vectors

R has six atomic vector types:

`logical`, `integer`, `double`, `character`, `complex`, `raw`

In this course we will mostly work with the first four. You will rarely work with the last two types - complex and raw.

# Conditional control flow

Conditional (choice) control flow is governed by `if` and `switch()`.

```r
if (condition) {
  # code to run
  # when condition is
  # TRUE
}
```

```r
if (TRUE) {
  print("The condition must have b
}
```

# `if` is not vectorized

To remedy this potential problem of a non-vectorized `if`, you can

1. try to collapse the logical vector to a vector of length 1
   - `any()`
   - `all()`
2. use a vectorized conditional function such as `ifelse()` or `dplyr::case_when()`.

# Loop types

R supports three types of loops: `for`, `while`, and `repeat`.

```r
for (item in vector) {
  ##
  ## Iterate this code
  ##
}
```

```r
while (we_have_a_true_condition) {
  ##
  ## Iterate this code
  ##
}
```

```r
repeat {
  ##
  ## Iterate this code
  ##
}
```

In the `repeat` loop we will need a `break` statement to end iteration.

# Concatenation

Atomic vectors can be constructed using the concatenate, `c()`, function.

```
c(1,2,3)
```

```
#> [1] 1 2 3
```

```
c("Hello", "World!")
```

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

```
c(1,c(2, c(3)))
```

```
#> [1] 1 2 3
```

Atomic vectors are always flat.

# More on atomic vectors

# Atomic vectors

| `typeof()` | `mode()` | `storage.mode()` |
|---|---|---|
| logical | logical | logical |
| double | numeric | double |
| integer | numeric | integer |
| character | character | character |
| complex | complex | complex |
| raw | raw | raw |

- Function `typeof()` can handle any object
- Functions `mode()` and `storage.mode()` allow for assignment

# Examples of type and mode

```r
typeof(c(T, F, T))
```

```
#> [1] "logical"
```

```r
typeof(7)
```

```
#> [1] "double"
```

```r
typeof(7L)
```

```
#> [1] "integer"
```

```r
typeof("S")
```

```
#> [1] "character"
```

```r
typeof("Shark")
```

```
#> [1] "character"
```

```r
mode(c(T, F, T))
```

```
#> [1] "logical"
```

```r
mode(7)
```

```
#> [1] "numeric"
```

```r
mode(7L)
```

```
#> [1] "numeric"
```

```r
mode("S")
```

```
#> [1] "character"
```

```r
mode("Shark")
```

```
#> [1] "character"
```

# Atomic vector type observations

- Numeric means an object of type integer or double.
- Integers must be followed by an L, except if you use operator `:`.

```
x <- 1:100
y <- as.numeric(1:100)
c(typeof(x), typeof(y))
```

```
#> [1] "integer" "double"
```

```
object.size(x)
object.size(y)
```

```
#> 448 bytes
#> 848 bytes
```

- There is no "string" type or mode, only "character".

# Logical predicates

The `is.*(x)` family of functions performs a logical test as to whether `x` is of type `*`. For example,

```r
is.integer(T)
```

```
#> [1] FALSE
```

```r
is.double(pi)
```

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

```r
is.character("abc")
```

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

```r
is.numeric(1L)
```

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

```r
is.integer(pi)
```

```
#> [1] FALSE
```

```r
is.double(pi)
```

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

```r
is.integer(1:10)
```

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

```r
is.numeric(1)
```

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

Function `is.numeric(x)` returns `TRUE` when `x` is integer or double.

# Coercion

Previously, we looked at R's coercion hierarchy:

$$\texttt{character} \rightarrow \texttt{double} \rightarrow \texttt{integer} \rightarrow \texttt{logical}$$

Coercion can happen implicitly through functions and operations; it can occur explicitly via the `as.*()` family of functions.

# Implicit coercion

```r
x <- c(T, T, F, F, F)
mean(x)
```

```
#> [1] 0.4
```

```r
c(1L, 1.0, "one")
```

```
#> [1] "1"   "1"   "one"
```

```r
0 >= "0"
```

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

```r
(0 == "0") != "TRUE"
```

```
#> [1] FALSE
```

```r
1 & TRUE & 5.0 & pi
```

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

```r
0 == FALSE
```

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

```r
(0 | 1) & 0
```

```
#> [1] FALSE
```

# Explicit coercion

```r
as.logical(sqrt(2))
```

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

```r
as.character(5L)
```

```
#> [1] "5"
```

```r
as.integer("4")
```

```
#> [1] 4
```

```r
as.integer("four")
```

```
#> [1] NA
```

```r
as.numeric(FALSE)
```

```
#> [1] 0
```

```r
as.double(10L)
```

```
#> [1] 10
```

```r
as.complex(5.4)
```

```
#> [1] 5.4+0i
```

```r
as.logical(as.character(3))
```

```
#> [1] NA
```

# Reserved words: `NA`, `NaN`, `Inf`, `-Inf`

- `NA` is a logical constant of length 1 which serves a missing value indicator.
- `NaN` stands for not a number.
- `Inf`, `-Inf` are positive and negative infinity, respectively.

# Missing values

- `NA` can be coerced to any other vector type except raw.

```r
typeof(NA)
```

```
#> [1] "logical"
```

```r
typeof(NA+1)
```

```
#> [1] "double"
```

```r
typeof(NA+1L)
```

```
#> [1] "integer"
```

```r
typeof(NA_character_)
```

```
#> [1] "character"
```

```r
typeof(NA_real_)
```

```
#> [1] "double"
```

```r
typeof(NA_integer_)
```

```
#> [1] "integer"
```

# `NA` in, `NA` out (most of the time)

```
x <- c(-4, 0, NA, 33, 1 / 9)
mean(x)
```

```
#> [1] NA
```

```
NA ^ 4
```

```
#> [1] NA
```

```
log(NA)
```

```
#> [1] NA
```

Some of the `base` R functions have an argument `na.rm` to remove `NA` values in the calculation.

```
mean(x, na.rm = TRUE)
```

```
#> [1] 7.277778
```

# Special non-infectious `NA` cases

```r
NA ^ 0
```

```
#> [1] 1
```

```r
NA | TRUE
```

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

```r
NA & FALSE
```

```
#> [1] FALSE
```

Why does `NA / Inf` result in `NA`?

# Testing for NA

Use function `is.na()` (vectorized) to test for `NA` values.

```r
is.na(NA)
```

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

```r
is.na(1)
```

```
#> [1] FALSE
```

```r
is.na(c(1,2,3,NA))
```

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

```r
any(is.na(c(1,2,3,NA)))
```

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

```r
all(is.na(c(1,2,3,NA)))
```

```
#> [1] FALSE
```

# NaN, Inf, and -Inf

```
-5 / 0
```

```
#> [1] -Inf
```

```
0 / 0
```

```
#> [1] NaN
```

```
1/0 + 1/0
```

```
#> [1] Inf
```

```
1/0 - 1/0
```

```
#> [1] NaN
```

```
NaN / NA
```

```
#> [1] NaN
```

```
NaN * NA
```

```
#> [1] NaN
```

- Functions `is.finite()` and `is.nan()` test for `Inf`, `-Inf`, and `NaN`, respectively.
- Coercion is possible with the `as.*()` family of functions. Be careful with these; they may not always work as you expect.

```
as.integer(Inf)
```

```
#> [1] NA
```

# Atomic vector properties

- Homogeneous
- Elements can have names
- Elements can be indexed by name or position
- Matrices, arrays, factors, and date-times are built on top of atomic vectors by adding attributes.

```
x <- c(-3:2)
attributes(x)
```

```
#> NULL
```

```
x
```

```
#> [1] -3 -2 -1  0  1  2
```

```
attr(x, which = "dim") <- c(2, 3)
attributes(x)
```

```
#> $dim
#> [1] 2 3
```

```
x
```

```
#>      [,1] [,2] [,3]
#> [1,]   -3   -1    1
#> [2,]   -2    0    2
```

# Exercises

1. What is the type of each vector below? Check your answer in R.

```
c(4L, 16, 0)
c(NaN, NA, -Inf)
c(NA, TRUE, FALSE, "TRUE")
c(pi, NaN, NA)
```

2. Write a conditional statement that prints "Can't proceed NA or NaN present!" if a vector contains `NA` or `NaN`. Test your code with vectors `x` and `y` below.

```
x <- NA
y <- c(1:5, NaN, NA, sqrt(3))
```

# Generic vectors

# Lists

Lists are generic vectors, in that they are 1 dimensional (i.e. have a length) and can contain any type of R object. They are heterogeneous structures.

```
list("A", c(TRUE,FALSE), (1:4)/2, function(x) x^2)
```

```
#> [[1]]
#> [1] "A"
#>
#> [[2]]
#> [1]  TRUE FALSE
#>
#> [[3]]
#> [1] 0.5 1.0 1.5 2.0
#>
#> [[4]]
#> function(x) x^2
```

# Structure

For complex objects, function `str()` will display the structure in a compact form.

```
str(list("A", c(TRUE,FALSE), (1:4)/2, function(x) x^2))
```

```
#> List of 4
#>  $ : chr "A"
#>  $ : logi [1:2] TRUE FALSE
#>  $ : num [1:4] 0.5 1 1.5 2
#>  $ :function (x)
#>   ..- attr(*, "srcref")= 'srcref' int [1:8] 1 39 1 53 39 53 1 1
#>   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7
```

# Coercion and testing

Lists can be complex structures and even include other lists.

```
x <- list("a", list("b", c("c", "d"), list(1:5)))
```

```
> str(x)
List of 2
$ : chr "a"
$ :List of 3
  ..$ : chr "b"
  ..$ : chr [1:2] "c" "d"
  ..$ :List of 1
  .. ..$ : int [1:5] 1 2 3 4 5
```

# Coercion and testing

Lists can be complex structures and even include other lists.

```
x <- list("a", list("b", c("c", "d"), list(1:5)))
```

```
> str(x)
List of 2
$ : chr "a"
$ :List of 3
  ..$ : chr "b"
  ..$ : chr [1:2] "c" "d"
  ..$ :List of 1
  .. ..$ : int [1:5] 1 2 3 4 5
```

# Coercion and testing

Lists can be complex structures and even include other lists.

```
x <- list("a", list("b", c("c", "d"), list(1:5)))
```

```
> str(x)
List of 2
$ : chr "a"
$ :List of 3
  ..$ : chr "b"
  ..$ : chr [1:2] "c" "d"
  ..$ :List of 1
  .. ..$ : int [1:5] 1 2 3 4 5
```

```
typeof(x)
```

```
#> [1] "list"
```

You can test for a list and coerce an object to a list with `is.list()` and `as.list()`, respectively.

# Flattening

Function `unlist()` will turn a list into an atomic vector. Keep R's coercion hierarchy in mind if you use this function.

```
y <- list(1:5, pi, c(T, F, T, T))
unlist(y)
```

```
#>  [1] 1.000000 2.000000 3.000000 4.000000 5.000000 3.141593 1.000000 0.000000
#>  [9] 1.000000 1.000000
```

```
x <- list("a", list("b", c("c", "d"), list(1:5)))
unlist(x)
```

```
#> [1] "a" "b" "c" "d" "1" "2" "3" "4" "5"
```

# List properties

- Lists are heterogeneous.
- Lists elements can have names.

```
list(stocks = c("AAPL", "BA", "PFE", "C"),
     eps    = c(1.1, .9, 2.3, .54),
     index  = c("DJIA", "NASDAQ", "SP500"))
```

```
#> $stocks
#> [1] "AAPL" "BA"   "PFE"  "C"
#>
#> $eps
#> [1] 1.10 0.90 2.30 0.54
#>
#> $index
#> [1] "DJIA"   "NASDAQ" "SP500"
```

- Lists can be indexed by name or position.
- Lists let you extract sublists or a specific object.

# Exercise

Create a list based on the JSON product order data below.


```
[
 {
  "id": {
    "oid": "5968dd23fc13ae04d9000001"
  },
  "product_name": "sildenafil citrate",
  "supplier": "Wisozk Inc",
  "quantity": 261,
  "unit_cost": "$10.47"
 },

 {
  "id": {
    "oid": "5968dd23fc13ae04d9000002"
  },
  "product_name": "Mountain Juniperus ashei",
  "supplier": "Keebler-Hilpert",
  "quantity": 292,
  "unit_cost": "$8.74"
 }
]
```

# Functions

# Fundamentals

A function is comprised of arguments (formals), body, and environment. The first two will be our main focus as we use and develop these objects.

```r
f <- function(x, y, z) {

  # combine words
  paste(x, y, z, sep = " ")

}

f(x = "just", y = "three",
  z = "words")
```

```
#> [1] "just three words"
```

```r
formals(f)
```

```
#> $x
#> 
#> 
#> $y
#> 
#> 
#> $z
```

```r
body(f)
```

```
#> {
#>     paste(x, y, z, sep = " ")
#> }
```

```r
environment(f)
```

```
#> <environment: R_GlobalEnv>
```

# Exiting

Most functions end by returning a value (implicitly or explicitly) or in error.

**Implicit return**

```
centers <- function(x) {

  c(mean(x), median(x))

}
```

**Explicit return**

```
standardize <- function(x) {

  stopifnot(length(x) > 1)
  x_stand <- (x - mean(x)) / sd(x)
  return(x_stand)

}
```

R functions can return any object.

# Calls

Function calls involve the function's name and, at a minimum, values to its required arguments. Arguments can be given values by

1. position

   ```
   z <- 1:30
   mean(z, .3, FALSE)
   ```

   ```
   #> [1] 15.5
   ```

2. name

   ```
   mean(x = z, trim = .3, na.rm = FALSE)
   ```

   ```
   #> [1] 15.5
   ```

3. partial name matching

   ```
   mean(x = z, na = FALSE, t = .3)
   ```

   ```
   #> [1] 15.5
   ```

# Call style

The best choice is

```
mean(z, trim = .3)
```

```
#> [1] 15.5
```

Leave the argument's name out for the commonly used (required) arguments, and always specify the argument names for the optional arguments.

# Scope

R uses lexical scoping. This provides a lot of flexibility, but it can also be problematic if a user is not careful. Let's see if we can get an idea of the scoping rules.

```
y <- 1

f <- function(x){

  y <- x ^ 2
  return(y)

}

f(x = 3)
y
```

What is the result of `f(x = 3)` and `y`?

```r
y <- 1
z <- 2

f <- function(x){

  y <- x ^ 2

  g <- function() {

    c(x, y, z)

  } # closes body of g()

  g()

} # closes body of f()


f(x = 3)
c(y, z)
```

What is the result of `f(x = 3)` and `c(y, z)`?

R first searches for a value associated with a name in the current environment. If the object is not found the search is widened to the next higher scope.

# Lazy evaluation

Arguments to R functions are not evaluated until needed.

```
f <- function(a, b, x) {
  print(a)
  print(b ^ 2)
  0 * x
}

f(5, 6)
```

```
#> [1] 5
#> [1] 36

#> Error in f(5, 6): argument "x" is missing, with no default
```

# Four function forms

| Form | Description | Example(s) |
|---|---|---|
| Prefix | name comes before arguments | `log(x, base = exp(1))` |
| Infix | name between arguments | `+`, `%>%`, `%/%` |
| Replacement | replace values by assignment | `names(x) <- c("a", "b")` |
| Special | all others not defined above | `[[`, `for`, `break`, `(` |

# Help

To get help on any function, type `?fcn_name` in your console, where `fcn_name` is the function's name. For infix, replacement, and special functions you will need to surround the function with backticks.

```
?sd
```

```
?`for`
```

```
?`names<-`
```

```
?`%/%`
```

Using function `help()` is an alternative to `?`.

# Best practices

- Write a function when you have copied code more than twice.
- Try to use a verb for your function's name.
- Keep argument names short but descriptive.
- Add code comments to explain the "why" of your code.
- Link a family of functions with a common prefix: `pnorm()`, `pbinom()`, `ppois()`.
- Keep data arguments first, then other required arguments, then followed by default arguments. The `...` argument can be placed last.

***To understand computations in R, two slogans are helpful:***

- ***Everything that exists is an object.***
- ***Everything that happens is a function call.***

***John Chambers***

# References

1. Grolemund, G., & Wickham, H. (2019). R for Data Science. https://r4ds.had.co.nz/
2. Wickham, H. (2019). Advanced R. https://adv-r.hadley.nz/