Columbia University
in the City of New York

DSFBA: Data Structures and Subsetting

Data Science for Business Analytics

Thibault Vatter

Department of Statistics, Columbia University

09/16/2020

Outline

1 Data structures

2 Atomic vectors

3 Attributes

4 S3 atomic vectors

5 Lists

6 Data frames and tibbles

7 Subsetting

Warm-up

- Type the following into your console:

```
# Create a vector in R
x <- c(5, 29, 13, 87)
x
#> [1]  5 29 13 87
```

- Two important ideas:
    - Commenting (we will come back to this)
    - Assignment
        - The `<-` symbol means assign `x` the value `c(5, 29, 13, 87)`.
        - Could use `=` instead of `<-` but this is discouraged.
        - All assignments take the same form: `object_name <- value`.
        - `c()` means “concatenate".
        - Type `x` into the console to print its assignment.

Warm-up cont'd

- Type the following into your console:

```r
# Create a vector in R
x <- c(5, 29, 13, 87)
x
#> [1]  5 29 13 87
```

- Note: the [1] tells us that 5 is the first element of the vector.

```r
# Create a vector in R
x <- 1:50
x
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
#> [22] 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
#> [43] 43 44 45 46 47 48 49 50
```

Data structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

- Almost all other objects are built upon these foundations.
- R has no 0-dimensional, or scalar types.
- Best way to understand what data structures any object is composed of is `str()` (short for structure).

```r
x <- c(5, 29, 13, 87)
str(x)
#>  num [1:4] 5 29 13 87
```

Vector

- Two flavors:
  - atomic vectors,
  - lists.
- Three common properties:
  - Type, `typeof()`, what it is.
  - Length, `length()`, how many elements it contains.
  - Attributes, `attributes()`, additional arbitrary metadata.
- Main difference: elements of an atomic vector must be the same type, whereas those of a list can have different types.

Outline

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

1 Data structures

2 Atomic vectors

3 Attributes

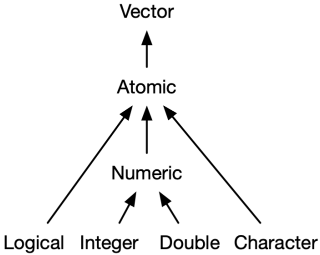
4 S3 atomic vectors

5 Lists

6 Data frames and tibbles

7 Subsetting

Atomic vectors



- Four primary types of atomic vectors: logical, integer, double, and character (which contains strings).
- Integer and double vectors are known as numeric vectors.
- There are two rare types: complex and raw (won't be discussed further).

Scalars

Special syntax to create an individual value, AKA a **scalar**:

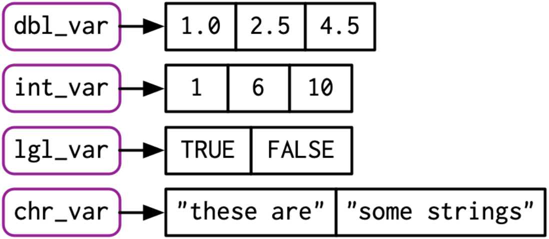
- Logicals:
  - In full (`TRUE` or `FALSE`),
  - Abbreviated (`T` or `F`).
- Doubles:
  - Decimal (`0.1234`), scientific (`1.23e4`), or hexadecimal (`0xcafe`) form.
  - Special values unique to doubles: `Inf`, `-Inf`, and `NaN` (not a number).
- Integers:
  - Similar to doubles but
    - must be followed by `L` (`1234L`, `1e4L`, or `0xcafeL`),
    - and can not contain fractional values.
- Strings:
  - Surrounded by `"` (`"hi"`) or `'` (`'bye'`).
  - Special characters escaped with \; see `?Quotes` for details.

Making longer vectors with `c()`

To create longer vectors from shorter ones, use `c()`:

```r
lgl_var <- c(TRUE, FALSE)
int_var <- c(1L, 6L, 10L)
dbl_var <- c(1, 2.5, 4.5)
chr_var <- c("these are", "some strings")
```

Depicting vectors as connected rectangles:



Making longer vectors cont’d

- With atomic vectors, `c()` returns atomic vectors (i.e., flattens):

```
c(c(1, 2), c(3, 4))
#> [1] 1 2 3 4
```

- Determine the type and length of a vector with `typeof()` and `length()`:

```
typeof(lgl_var)
#> [1] "logical"
typeof(int_var)
#> [1] "integer"
typeof(dbl_var)
#> [1] "double"
typeof(chr_var)
#> [1] "character"
```

Missing or unknown values

- Represented with `NA` (short for not applicable/available).
- Missing values tend to be infectious:

```r
NA > 5
#> [1] NA
10 * NA
#> [1] NA
!NA
#> [1] NA
```

- Exception: when some identity holds for all possible inputs. . .

```r
NA ^ 0
#> [1] 1
NA | TRUE
#> [1] TRUE
NA & FALSE
#> [1] FALSE
```

Missing or unknown values cont'd

- Propagation of missingness leads to a common mistake:

```
x <- c(NA, 5, NA, 10)
x == NA
#> [1] NA NA NA NA
```

- Instead, use `is.na()`:

```
is.na(x)
#> [1]  TRUE FALSE  TRUE FALSE
```

NULL

- Closely related to vectors.
- Special because it has a unique type, is always length zero, and can't have any attributes.

```
typeof(NULL)
#> [1] "NULL"

length(NULL)
#> [1] 0

x <- NULL
attr(x, "y") <- 1
#> Error in attr(x, "y") <- 1: attempt to set an attribute on NULL
```

- Can test for NULLs with `is.null()`:

```
is.null(NULL)
#> [1] TRUE
```

NULL cont'd

- NULL commonly represents
  - an absent vector.
    - For example, NULL is often used as a default function argument.
    - Contrast this with NA, which indicates that an *element* of a vector is absent.
  - an empty vector (a vector of length zero) of arbitrary type.

```r
c()
#> NULL
c(NULL, NULL)
#> NULL
c(NULL, 1:3)
#> [1] 1 2 3
```

- If you're familiar with SQL, you'll know about relational NULL, but the database NULL is actually equivalent to R's NA.

Testing and coercion

- Test if a vector is of a given type with `is.*()`, but be careful:
  - `is.logical()`, `is.integer()`, `is.double()`, and `is.character()` do what you might expect.
  - Avoid `is.vector()`, `is.atomic()`, and `is.numeric()` or carefully read the documentation.
- For atomic vectors:
  - Type is a property of the entire vector (all elements of the same type).
  - When combining different types: **coercion** in a fixed order (character → double → integer → logical).

```r
str(c("a", 1))
#>  chr [1:2] "a" "1"
```

Testing and coercion cont'd

- Often happens automatically:
  - Most mathematical functions (+, log, etc.) coerce to numeric.
  - Useful for logical vectors because TRUE/FALSE become 1/0.

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
#> [1] 0 0 1
c(sum(x), mean(x)) # Total number of TRUEs and proportion that are TRUE
#> [1] 1.000 0.333
```

- Additionally:
  - Deliberately coerce by using as.*() (as.logical(), as.integer(), as.double(), or as.character()).
  - Failed coercion of strings → warning and missing value.

```
as.integer(c("1", "1.5", "a"))
#> Warning: NAs introduced by coercion
#> [1]  1  1 NA
```

Outline

1 Data structures

2 Atomic vectors

3 Attributes

4 S3 atomic vectors

5 Lists

6 Data frames and tibbles

7 Subsetting

Attributes

How about matrices, arrays, factors, or date-times?

- Built on top of atomic vectors by adding attributes.
- In the next few slides:
  - The `dim` attribute to make matrices and arrays.
  - The `class` attribute to create “S3” vectors, including factors, dates, and date-times.

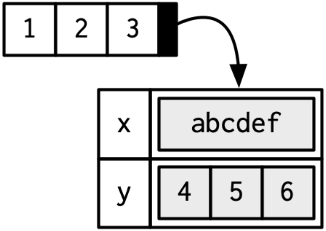
Getting and setting

- Similar to name-value pairs attaching metadata to an object.
- Attributes can be retrieved/modified
  - individually with `attr()`,
  - or “En masse” with `attributes()`/`structure()`.

```
a <- 1:3
attr(a, "x") <- "abcdef"
attr(a, "x")
#> [1] "abcdef"

attr(a, "y") <- 4:6
str(attributes(a))
#> List of 2
#>  $ x: chr "abcdef"
#>  $ y: int [1:3] 4 5 6

# Or equivalently
a <- structure(
  1:3,
  x = "abcdef",
  y = 4:6
)
```



Getting and setting cont'd

- Attributes should generally be thought of as ephemeral.
- For example, most attributes are lost by most operations:

```r
attributes(a[1])
#> NULL
attributes(sum(a))
#> NULL
```

- There are only two attributes that are routinely preserved:
  - `names`, a character vector giving each element a name.
  - `dim`, short for dimensions, an integer vector, used to turn vectors into matrices or arrays.
- To preserve other attributes, need to create your own S3 class!

Names

- You can name a vector in three ways:

```
# When creating it
x <- c(a = 1, b = 2, c = 3)

# By assigning a character vector to names()
x <- 1:3
names(x) <- c("a", "b", "c")

# Inline, with setNames()
x <- setNames(1:3, c("a", "b", "c"))
```

- Avoid `attr(x, "names")` (more typing and less readable).
- Remove names with `unname(x)` or `names(x) <- NULL`.

Dimensions

- The `dim` attribute allow a vector allows it to behave like a 2-dimensional **matrix** or a multi-dimensional **array**.
- Most important feature: multidimensional subsetting, which we'll see later.
- Create matrices and arrays with `matrix()`:

```
# Two scalar arguments specify row and column sizes
a <- matrix(1:6, nrow = 2, ncol = 3)
a
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

Dimensions cont'd

- Or arrays with `array()`:

```
# One vector argument to describe all dimensions
b <- array(1:12, c(2, 3, 2))
b
#> , , 1
#> 
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#> 
#> , , 2
#> 
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12
```

Dimensions cont'd

- Alternatively, use the assignment form of `dim()`:

```
# You can also modify an object in place by setting dim()
c <- 1:6
dim(c) <- c(3, 2)
c
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
```

- Functions for working with vectors, matrices and arrays:

Vector	Matrix	Array
`names()`	`rownames()`, `colnames()`	`dimnames()`
`length()`	`nrow()`, `ncol()`	`dim()`
`c()`	`rbind()`, `cbind()`	`abind::abind()`
—	`t()`	`aperm()`
`is.null(dim(x))`	`is.matrix()`	`is.array()`

Dimensions cont'd

- A vector without a `dim` attribute set is often thought of as 1-dimensional, but actually has `NULL` dimensions.
- You also can have matrices with a single row or single column, or arrays with a single dimension:
  - They may print similarly, but will behave differently.
  - The differences aren't too important, but it's useful to know they exist in case you get strange output from a function.
  - As always, use `str()` to reveal the differences.

```
str(1:3)                   # 1d vector
#>  int [1:3] 1 2 3
str(matrix(1:3, ncol = 1)) # column vector
#>  int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1)) # row vector
#>  int [1, 1:3] 1 2 3
str(array(1:3, 3))         # "array" vector
#>  int [1:3(1d)] 1 2 3
```

Outline

1 Data structures

2 Atomic vectors

3 Attributes

4 **S3 atomic vectors**

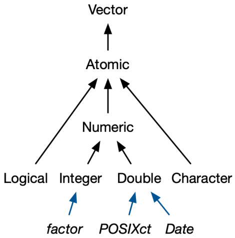
5 Lists

6 Data frames and tibbles

7 Subsetting

S3 atomic vectors

- One of the most important attributes is `class`.
  - Turns an object into an **S3 object** (which behaves differently when passed to a **generic** function).
  - Every S3 object
    - is built on top of a base type,
    - stores additional information in other attributes.
- In the next few slides, three important classes in R:
  - Categorical data (values come from a fixed set of levels): **factor** vectors.
  - Dates (day resolution): **Date** vectors.
  - Date-times (second or sub-second resolution): **POSIXct** vectors.

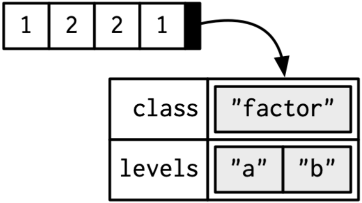


Factors

- A vector that can contain only predefined values.
- Used to store categorical data.
- Built on top of an integer vector with two attributes:
    - a `class` (defines a behavior different from integer vectors),
    - and `levels` (defines the set of allowed values).

```
x <- factor(c("a", "b", "b", "a"))
x
#> [1] a b b a
#> Levels: a b

typeof(x)
#> [1] "integer"
attributes(x)
#> $levels
#> [1] "a" "b"
#>
#> $class
#> [1] "factor"
```



Factors cont'd

- Useful when you know the set of possible values but they're not all present in a given dataset.
- When tabulating a factor you'll get counts of all categories, even unobserved ones:

```r
sex_char <- c("m", "m", "m")
table(sex_char)
#> sex_char
#> m
#> 3

sex_factor <- factor(sex_char, levels = c("m", "f"))
table(sex_factor)
#> sex_factor
#> m f
#> 3 0
```

Factors cont'd

- Ordered factors:
  - Behave like regular factors, but the order of the levels is meaningful (e.g., low, medium, high)
  - This property is automatically leveraged by some modelling/visualisation functions.

```
grade <- ordered(c("b", "b", "a", "c"), levels = c("c", "b", "a"))
grade
#> [1] b b a c
#> Levels: c < b < a
```

- While factors look like character vectors, be careful:
  - Some string methods (like `gsub()` and `grepl()`) will automatically coerce factors to strings.
  - Others (like `nchar()`) will throw an error.
  - Still others will (like `c()`) use the underlying integer values.
  - Best to explicitly convert factors to character vectors if you need string-like behavior.

Factors cont'd

- In base R:
  - Before R 4.0, factors used to be everywhere because many functions (e.g. `read.csv()`/`data.frame()`) automatically converted character vectors to factors.
  - Suboptimal because there's no way to know the set of all possible levels or their correct order!
- The tidyverse:
  - Never automatically coerces characters to factors.
  - Provides the `forcats` package specifically for working with factors.
  - More on that later.

Dates

- Built on top of double vectors.
- A class `Date` and no other attributes.

```
today <- Sys.Date()

typeof(today)
#> [1] "double"
attributes(today)
#> $class
#> [1] "Date"
```

- Value of the double = the number of days since 1970-01-01[1]:

```
date <- as.Date("1970-02-01")
unclass(date)
#> [1] 31
```

[1] Known as the Unix Epoch.

Dates-times

- Two ways of storing this information: POSIXct, and POSIXlt.
- Odd names:
    - “POSIX” is short for “Portable Operating System Interface”,
    - “ct” stands for calendar time (`time_t` in `C`),
    - and “lt” for local time (`struct tm` type in `C`).
- Focus on `POSIXct` (the simplest):
    - Built on top of a double vector.
    - Value = number of seconds since 1970-01-01.

```
now_ct <- as.POSIXct("2018-08-01 22:00", tz = "UTC")
now_ct
#> [1] "2018-08-01 22:00:00 UTC"

typeof(now_ct)
#> [1] "double"
attributes(now_ct)
#> $class
#> [1] "POSIXct" "POSIXt"
#>
#> $tzone
#> [1] "UTC"
```

Dates-times cont'd

- The tzone attribute:
  - Controls only how the formatting; not the represented instant.
  - The time is not printed if it is midnight.

```r
structure(now_ct, tzone = "Asia/Tokyo")
#> [1] "2018-08-02 07:00:00 JST"
structure(now_ct, tzone = "America/New_York")
#> [1] "2018-08-01 18:00:00 EDT"
structure(now_ct, tzone = "Australia/Lord_Howe")
#> [1] "2018-08-02 08:30:00 +1030"
structure(now_ct, tzone = "Europe/Paris")
#> [1] "2018-08-02 CEST"
```

Outline

1 Data structures

2 Atomic vectors

3 Attributes

4 S3 atomic vectors

5 Lists

6 Data frames and tibbles

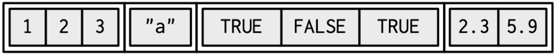
7 Subsetting

Lists

- A step up in complexity from atomic vectors.
- Each element can be any type.
- Construct lists with `list()`.

```
l1 <- list(
  1:3,
  "a",
  c(TRUE, FALSE, TRUE),
  c(2.3, 5.9)
)

typeof(l1)
#> [1] "list"
str(l1)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

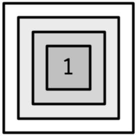


1 2 3 "a" TRUE FALSE TRUE 2.3 5.9

Lists cont'd

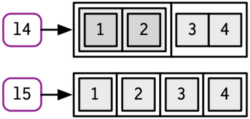
- Sometimes called **recursive** vectors:

```
l3 <- list(list(list(1)))
str(l3)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : num 1
```



- c() will combine several lists into one:

```
l4 <- list(list(1, 2), c(3, 4))
l5 <- c(list(1, 2), c(3, 4))
str(l4)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(l5)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```



Testing and coercion

- The `typeof()` a list is `list`.
- Test for a list with `is.list()`.
- Coerce to a list with `as.list()`.

```r
list(1:3)
#> [[1]]
#> [1] 1 2 3
as.list(1:3)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

- Flatten a list into an atomic vector with `unlist()`, but rules are. . .
  - complex & not well documented :(

Outline

1 Data structures

2 Atomic vectors

3 Attributes

4 S3 atomic vectors

5 Lists

6 Data frames and tibbles

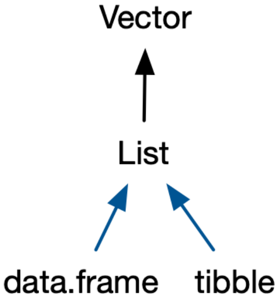
7 Subsetting

Data frames and tibbles

- The two most important S3 vectors built on top of lists.
- If you do data analysis in R, you'll use them.
- A data frame is a named list of vectors with attributes for (column) `names`, `row.names`, and its class, `data.frame`.

```
df1 <- data.frame(x = 1:3, y = letters[1:3])
typeof(df1)
#> [1] "list"

attributes(df1)
#> $names
#> [1] "x" "y"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] 1 2 3
```



Vector

List

data.frame

tibble

Data frames and tibbles cont'd

- Similar to a list, but with an additional constraint:
  - The length of each of its vectors must be the same.
  - "Rectangular structure":
    - Share properties of both matrices and lists.
    - Has `rownames()`/`colnames()` & its `names()` are the column names.
    - Has `nrow()`/`ncol()` & its `length()` is the number of columns.
- Data frames:
  - One of the biggest and most important ideas in R!
  - . . . .. but
    - 20 years have passed since their creation.
    - Lead to the creation of the `tibble`, a modern version.

Tibbles

- Provided by the `tibble` package.
- **Main difference: lazy (do less) & surly (complain more).**
- Technically:
    - Share the same structure as `data.frame`.
    - Only difference is that the class vector includes `tbl_df`.
    - Allows tibbles to behave differently.

```
library(tibble)

df2 <- tibble(x = 1:3, y = letters[1:3])
typeof(df2)
#> [1] "list"

attributes(df2)
#> $names
#> [1] "x" "y"
#>
#> $row.names
#> [1] 1 2 3
#>
#> $class
#> [1] "tbl_df"     "tbl"        "data.frame"
```

Creating a `data.frame` or a `tibble`

- Supply name-vector pairs to `data.frame()` or `tibble()`.

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c")
)
```

```
df2 <- tibble(
  x = 1:3,
  y = c("a", "b", "c")
)
```

```
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"

str(df2)
#> tibble [3 x 2] (S3: tbl_df/tbl/data.frame)
#>  $ x: int [1:3] 1 2 3
#>  $ y: chr [1:3] "a" "b" "c"
```

- Next few slides: some of the differences between the two.
  - Non-syntactic names.
  - Recycling shorter inputs.
  - Variables created during construction.
  - Printing.

Non-syntactic names

- Strict rules about what constitutes a valid name.
  - **Syntactic** names consist of letters[2], digits, `.` and _ but can't begin with _ or a digit.
  - Additionally, can't use any of the **reserved words** like `TRUE`, `NULL`, `if`, and `function` (see the complete list in `?Reserved`).
- A name that doesn't follow these rules is **non-syntactic**.

```
_abc <- 1
#> Error: unexpected input in "_"

if <- 10
#> Error: unexpected assignment in "if <-"
```

[2] what constitutes a letter is determined by your current locale, avoid this by sticking to ASCII characters (i.e. A-Z) as much as possible.

Non-syntactic names cont'd

- To override these rules and use any name:

```
`_abc` <- 1
`_abc`
#> [1] 1

`if` <- 10
`if`
#> [1] 10
```

- Don't deliberately create but understand such names:
  - You'll come across them, e.g whith data created outside of R.
- In data frames and tibbles:

```
names(data.frame(`1` = 1))
#> [1] "X1"

names(data.frame(`1` = 1, check.names = FALSE))
#> [1] "1"

names(tibble(`1` = 1))
#> [1] "1"
```

Recycling shorter inputs

- Both `data.frame()` and `tibble()` recycle shorter inputs, but
  - data frames automatically recycle columns that are an integer multiple of the longest column,
  - tibbles will only recycle vectors of length one.

```
data.frame(x = 1:4, y = 1:2)
#>   x y
#> 1 1 1
#> 2 2 2
#> 3 3 1
#> 4 4 2
data.frame(x = 1:4, y = 1:3)
#> Error in data.frame(x = 1:4, y = 1:3): arguments imply differing
#> number of rows: 4, 3
```

Recycling shorter inputs cont'd

- Both `data.frame()` and `tibble()` recycle shorter inputs, but
  - data frames automatically recycle columns that are an integer multiple of the longest column,
  - tibbles will only recycle vectors of length one.

```r
tibble(x = 1:4, y = 1)
#> # A tibble: 4 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     3     1
#> 4     4     1
tibble(x = 1:4, y = 1:2)
#> Error: Tibble columns must have compatible sizes.
#> * Size 4: Existing data.
#> * Size 2: Column `y`.
#> i Only values of size one are recycled.
```

Variables created during construction

- `tibble()` allows you to refer to variables created during construction:

```
tibble(
  x = 1:3,
  y = x * 2
)
#> # A tibble: 3 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1     2
#> 2     2     4
#> 3     3     6
```

(Inputs are evaluated left-to-right.)

Printing

```
iris
#>     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
#> 1            5.1         3.5          1.4         0.2     setosa
#> 2            4.9         3.0          1.4         0.2     setosa
#> 3            4.7         3.2          1.3         0.2     setosa
#> 4            4.6         3.1          1.5         0.2     setosa
#> 5            5.0         3.6          1.4         0.2     setosa
#> 6            5.4         3.9          1.7         0.4     setosa
#> 7            4.6         3.4          1.4         0.3     setosa
#> 8            5.0         3.4          1.5         0.2     setosa
#> 9            4.4         2.9          1.4         0.2     setosa
#> 10           4.9         3.1          1.5         0.1     setosa
#> 11           5.4         3.7          1.5         0.2     setosa
#> 12           4.8         3.4          1.6         0.2     setosa
#> 13           4.8         3.0          1.4         0.1     setosa
#> 14           4.3         3.0          1.1         0.1     setosa
#> 15           5.8         4.0          1.2         0.2     setosa
#> 16           5.7         4.4          1.5         0.4     setosa
#> 17           5.4         3.9          1.3         0.4     setosa
#> 18           5.1         3.5          1.4         0.3     setosa
#> 19           5.7         3.8          1.7         0.3     setosa
#> 20           5.1         3.8          1.5         0.3     setosa
#> 21           5.4         3.4          1.7         0.2     setosa
```

Printing cont'd

```
dplyr::starwars
#> # A tibble: 87 x 14
#>    name  height  mass hair_color skin_color eye_color birth_year
#>    <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl>
#>  1 Luke~    172    77 blond      fair       blue            19
#>  2 C-3PO    167    75 <NA>       gold       yellow         112
#>  3 R2-D2     96    32 <NA>       white, bl~ red             33
#>  4 Dart~    202   136 none       white      yellow          41.9
#>  5 Leia~    150    49 brown      light      brown           19
#>  6 Owen~    178   120 brown, gr~ light      blue            52
#>  7 Beru~    165    75 brown      light      blue            47
#>  8 R5-D4     97    32 <NA>       white, red red             NA
#>  9 Bigg~    183    84 black      light      brown           24
#> 10 Obi-~    182    77 auburn, w~ fair       blue-gray       57
#> # ... with 77 more rows, and 7 more variables: sex <chr>,
#> #   gender <chr>, homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

- Only the first 10 rows + the columns that fit on screen.
- Each column is labelled with its abbreviated type.
- Wide columns are truncated.
- In RStudio, color highlights important information.

Testing and coercing

- To check if an object is a data frame or tibble:

```
is.data.frame(df1)
#> [1] TRUE
is.data.frame(df2)
#> [1] TRUE
```

- Typically, it should not matter if you have a tibble or data frame, but if you need to be certain:

```
is_tibble(df1)
#> [1] FALSE
is_tibble(df2)
#> [1] TRUE
```

- Coerce an object to a data frame or tibble with `as.data.frame()` or `as_tibble()`.

Outline

1 Data structures

2 Atomic vectors

3 Attributes

4 S3 atomic vectors

5 Lists

6 Data frames and tibbles

7 Subsetting

Subsetting

- R's subsetting operators are fast and powerful.
  - Allows to succinctly perform complex operations in a way that few other languages can match.
  - Easy to learn but hard to master because of a number of interrelated concepts:
    - Six ways to subset atomic vectors.
    - Three subsetting operators, `[[`, `[`, and `$`.
    - The operators interact differently with different vector types.
    - Subsetting can be combined with assignment.
- Subsetting is a natural complement to `str()`:
  - `str()` shows the pieces of any object (its structure).
  - Subsetting pulls out the pieces that you're interested in.
- Outline:
  - Selecting multiple elements with `[`.
  - Selecting a single element with `[[` and `$`.
  - Subsetting and assignment.

[for atomic vectors

- We'll look at the following vector:

```
x <- c(2.1, 4.2, 3.3, 5.4)
```

- Note that the number after the decimal point represents the original position in the vector.
- There are six things that you can use to subset a vector:
    - Positive integers.
    - Negative integers.
    - Logical vectors.
    - Nothing.
    - Zero.
    - Character vectors.

[for atomic vectors cont'd

- **Positive integers** return elements at the specified positions:

```
x[c(3, 1)]
#> [1] 3.3 2.1
x[order(x)]
#> [1] 2.1 3.3 4.2 5.4

x[c(1, 1)] # Duplicate indices will duplicate values
#> [1] 2.1 2.1

x[c(2.1, 2.9)] # Real numbers are silently truncated to integers
#> [1] 4.2 4.2
```

- **Negative integers** exclude elements at the specified positions:

```
x[-c(3, 1)]
#> [1] 4.2 5.4
```

- Can't mix positive and negative integers in a single subset:

```
x[c(-1, 2)]
#> Error in x[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

[for atomic vectors cont'd

- **Logical vectors** select elements where the corresponding logical value is TRUE (probably the most useful):

```r
x[c(TRUE, TRUE, FALSE, FALSE)]
#> [1] 2.1 4.2
x[x > 3]
#> [1] 4.2 3.3 5.4
```

- In x[y], what happens if x and y are different lengths?
  - Behavior controlled by the **recycling rules** with the shorter recycled to the length of the longer.
  - Convenient and easy to understand when x OR y is length one, but avoid for other lengths because of inconsistencies in base R.

```r
x[c(TRUE, FALSE)]
#> [1] 2.1 3.3
# Equivalent to
x[c(TRUE, FALSE, TRUE, FALSE)]
#> [1] 2.1 3.3
```

[for atomic vectors cont'd

- **Nothing** returns the original vector (not useful for 1D vectors, but important for matrices, data frames, and arrays:

```
x[]
#> [1] 2.1 4.2 3.3 5.4
```

- **Zero** returns a zero-length vector (not usually done on purpose):

```
x[0]
#> numeric(0)
```

[for atomic vectors cont'd

- If the vector is named, you can also use **character vectors** to return elements with matching names:

```
(y <- setNames(x, letters[1:4]))
#>   a   b   c   d
#> 2.1 4.2 3.3 5.4
y[c("d", "c", "a")]
#>   d   c   a
#> 5.4 3.3 2.1

# Like integer indices, you can repeat indices
y[c("a", "a", "a")]
#>   a   a   a
#> 2.1 2.1 2.1

# When subsetting with [, names are always matched exactly
z <- c(abc = 1, def = 2)
z[c("a", "d")]
#> <NA> <NA>
#>   NA   NA
```

[for atomic vectors cont'd

- Note that a missing value in the index always yields a missing value in the output:

```
x[c(TRUE, TRUE, NA, FALSE)]
#> [1] 2.1 4.2  NA
```

- Factors are not treated specially when subsetting:
  - Subsetting will use the underlying integer vector, not the character levels.
  - Typically unexpected, so avoid!

```
y[factor("b")]
#>   a
#> 2.1
```

[for lists

- Exactly as for atomic vectors.
- Using [always returns a list; [[and $ (see in a few slides), lets you pull out elements of a list.

[for matrices and arrays

- Subset higher-dimensional structures in three ways:
  - With multiple vectors.
  - With a single vector.
  - With a matrix.
- The most common way:
  - Supply a 1D index for each dimension, separated by a comma.
  - Blank subsetting is now useful!

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a[1:2, ]
#>      A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
a[c(TRUE, FALSE, TRUE), c("B", "A")]
#>      B A
#> [1,] 4 1
#> [2,] 6 3
a[0, -2]
#>      A C
```

[for matrices and arrays cont'd

- By default, [simplifies the results to the lowest possible dimensionality.
  - For example, both of the following expressions return 1D vectors.
  - You'll learn how to avoid "dropping" dimensions later.

```
a[1, ]
#> A B C
#> 1 4 7
a[1, 1]
#> A
#> 1
```

[for matrices and arrays cont'd

- Can subset them with a vector as if they were 1D.
- Note that arrays in R are stored in column-major order:

```
vals <- outer(1:5, 1:5, FUN = "paste", sep = ",")
vals
#>      [,1]  [,2]  [,3]  [,4]  [,5]
#> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
#> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"

vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

[for matrices and arrays cont'd

- Can also subset higher-dimensional data structures with an integer matrix (or, if named, a character matrix).
  - Each row in the matrix specifies the location of one value.
  - Each column corresponds to a dimension in the array.
  - E.g., use a 2 column matrix to subset a matrix, a 3 column matrix to subset a 3D array, etc.
  - The result is a vector of values.

```
select <- matrix(ncol = 2, byrow = TRUE, c(
  1, 1,
  3, 1,
  2, 4
))

vals[select]
#> [1] "1,1" "3,1" "2,4"
```

[for data frames and tibbles

- Characteristics of both lists and matrices.
- When subsetting with a single index:
  - Behave like lists and index the columns.
  - E.g. `df[1:2]` selects the first two columns.
- When subsetting with two indices:
  - Behave like matrices.
  - E.g. `df[1:3, ]` selects the first three *rows* (and all columns)[3].

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])

df[df$x == 2, ]
#>   x y z
#> 2 2 2 b
df[c(1, 3), ]
#>   x y z
#> 1 1 3 a
#> 3 3 1 c
```

[3] In Python `df[1:3, 1:2]` would select three columns and two rows.

[for data frames and tibbles cont'd

- Two ways to select columns from a data frame:

```
# Like a list
df[c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
# Like a matrix
df[, c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
```

[for data frames and tibbles cont'd

- Important difference if you select a single column:
  - Matrix subsetting simplifies by default.
  - List subsetting does not.

```
str(df[, "x"])
#>  int [1:3] 1 2 3
str(df["x"])
#> 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
```

- Subsetting a tibble with [always returns a tibble:

```
df <- tibble::tibble(x = 1:3, y = 3:1, z = letters[1:3])

str(df["x"])
#> tibble [3 x 1] (S3: tbl_df/tbl/data.frame)
#>  $ x: int [1:3] 1 2 3
str(df[, "x"])
#> tibble [3 x 1] (S3: tbl_df/tbl/data.frame)
#>  $ x: int [1:3] 1 2 3
```

Preserving dimensionality

- For matrices and arrays, dimensions with length 1 are dropped:

```r
a <- matrix(1:4, nrow = 2)
str(a[1, ])
#>  int [1:2] 1 3

str(a[1, , drop = FALSE])
#>  int [1, 1:2] 1 3
```

- Data frames with a single column returns just that column:

```r
df <- data.frame(a = 1:2, b = 1:2)
str(df[, "a"])
#>  int [1:2] 1 2

str(df[, "a", drop = FALSE])
#> 'data.frame':    2 obs. of  1 variable:
#>  $ a: int  1 2
```

Preserving dimensionality cont’d

- The default drop = TRUE is a common source of bugs:
  - Your code with a dataset with multiple columns works.
  - Six months later, you use it with a single column dataset and it fails with a mystifying error.
  - **Always use ‘drop = FALSE‘ when subsetting a 2D object!**
  - Tibbles default to drop = FALSE and [always returns a tibble.
- Factor subsetting also has a drop argument:
  - Controls whether or not levels (rather than dimensions) are preserved defaults to FALSE.
  - When using drop = TRUE, use a character vector instead.

```r
z <- factor(c("a", "b"))
z[1]
#> [1] a
#> Levels: a b
z[1, drop = TRUE]
#> [1] a
#> Levels: a
```

Selecting a single element

The other two subsetting operators:

- `[[` is used for extracting single items.
- `x$y` is a useful shorthand for `x[["y"]]`.

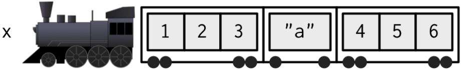
[[

- [[is most important when working with lists because subsetting a list with [always returns a smaller list.

  If list x is a train carrying objects, then x[[5]] is the object in car 5; x[4:6] is a train of cars 4-6.
  — @RLangTip, https://twitter.com/RLangTip/status/268375867468681216

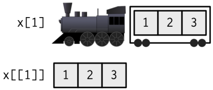
- Use this metaphor to make a simple list:

```
x <- list(1:3, "a", 4:6)
```

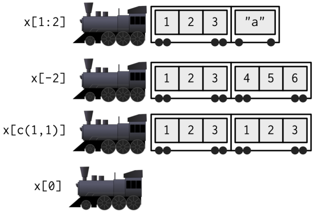


[[cont'd

- When extracting a single element, you have two options:
  - Create a smaller train, i.e., fewer carriages, with [.
  - Extract the contents of a particular carriage with [[.



- When extracting multiple (or even zero!) elements, you have to make a smaller train.



$

- Shorthand operator:
  - `x$y` is roughly equivalent to `x[["y"]]`.
  - Often used to access variables in a data frame.
  - E.g., `mtcars$cyl` or `diamonds$carat`.
- One common mistake with `$`:

```
var <- "cyl"
# Doesn't work - mtcars$var translated to mtcars[["var"]]
mtcars$var
#> NULL

# Instead use [[
mtcars[[var]]
#>  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

$ cont’d

- The one important difference between $ and [[is (left-to-right) partial matching:

```
x <- list(abc = 1)
x$a
#> [1] 1
x[["a"]]
#> NULL
```

- To avoid this, the following is highly recommended:

```
options(warnPartialMatchDollar = TRUE)
x$a
#> Warning in x$a: partial match of 'a' to 'abc'
#> [1] 1
```

(For data frames, you can also avoid this problem by using tibbles, which never do partial matching.)

Data frames and tibbles again

- Data frames have two undesirable subsetting behaviors.
  - When you subset columns with `df[, vars]`:
    - Returns a vector if `vars` selects one variable.
    - Otherwise, returns a data frame.
    - Frequent unless you use `drop = FALSE`.
  - When extracting a single column with `df$x`:
    - If there is no column `x`, selects any variable that starts with `x`.
    - If no variable starts with `x`, returns `NULL`.
    - Easy to select the wrong variable/a variable that doesn't exist.
- Tibbles tweak these behaviors:
  - [always returns a tibble.
  - $ doesn't do partial matching and warns if it can't find a variable (makes tibbles surly).

```r
df1 <- data.frame(xyz = "a")
str(df1$x)
#>  chr "a"
```

```r
df2 <- tibble(xyz = "a")
str(df2$x)
#> Warning: Unknown or uninitialised column: `x`.
#>  NULL
```

Subsetting and assignment –>

- Subsetting operators can be combined with assignment.
  - Modifies selected values of an input vector
  - Called subassignment.
- The basic form is `x[i] <- value`:

```
x <- 1:5
x[c(1, 2)] <- c(101, 102)
x
#> [1] 101 102   3   4   5
```

- Recommendation:
  - Make sure that `length(value)` is the same as `length(x[i])`,
  - and that `i` is unique.
  - Otherwise, you'll end-up in recycling hell.

Subsetting and assignment cont'd

- Subsetting lists with NULL
  - `x[[i]] <- NULL` removes a component.
  - To add a literal NULL, use `x[i] <- list(NULL)`.

```r
x <- list(a = 1, b = 2)
x[["b"]] <- NULL
str(x)
#> List of 1
#>  $ a: num 1
```

```r
y <- list(a = 1, b = 2)
y["b"] <- list(NULL)
str(y)
#> List of 2
#>  $ a: num 1
#>  $ b: NULL
```

- Subsetting with nothing can be useful with assignment
  - Preserves the structure of the original object.
  - Compare the following two expressions.

```r
mtcars[] <- lapply(mtcars, as.integer)
is.data.frame(mtcars)
#> [1] TRUE
```

```r
mtcars <- lapply(mtcars, as.integer)
is.data.frame(mtcars)
#> [1] FALSE
```