Data Science in Business Analytics

Data Structures and Subsetting

HEC Lausanne
Professor Alex [aleksandr.shemendyuk@unil.ch]

Agenda

1. Data structures
2. Atomic vectors
3. Attributes
4. S3 objects
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
  - Assignment
    - The <- symbol assigns the value `c(5, 29, 13, 87)` to `x`.
    - We can 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 stored value.
    - Number `[1]` indicates that `5` is the first element of the vector.

Warm-up (cont'd)

```
# Create a vector in R
x <- rnorm(50)
x
#>  [1]  1.26295 -0.32623  1.32980  1.27243  0.41464 -1.53995 -0.92857
#>  [8] -0.29472 -0.00577  2.40465  0.76359 -0.79901 -1.14766 -0.28946
#> [15] -0.29922 -0.41151  0.25222 -0.89192  0.43568 -1.23754 -0.22427
#> [22]  0.37740  0.13334  0.80419 -0.05711  0.50361  1.08577 -0.69095
#> [29] -1.28460  0.04673 -0.23571 -0.54289 -0.43331 -0.64947  0.72675
#> [36]  1.15191  0.99216 -0.42951  1.23830 -0.27935  1.75790  0.56075
#> [43] -0.45278 -0.83204 -1.16657 -1.06559 -1.56378  1.15654  0.83205
#> [50] -0.22733
```

Data structures

	Homogeneous[1]	Heterogeneous[2]
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".

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

[1] **Homogeneous**: Contains the **same type** of values.
[2] **Heterogeneous**: Contains **different types** of values.

1d data structures

- Two flavours:
  - **atomic vectors** (homogeneous),
  - **lists** (heterogeneous).
- Three common properties:
  - `typeof()` the variable: `"double"`, `"list"`, etc.
  - `length()` of the variable, how many elements it contains.
  - `attributes()`: additional metadata.
- **Main difference**: elements of an atomic vector must be the same type, whereas those of a list can have different types.

1d data structures (cont'd)

```
# Example: Atomic vector
atomic_vector <- c(first = 1, second = 2, third = 3)
atomic_vector
#>  first second  third
#>      1      2      3
attributes(atomic_vector)
#> $names
#> [1] "first"  "second" "third"
```

```
# Example: List
list_example <- list(x = 1, y = "a", TRUE)
list_example
#> $x
#> [1] 1
#>
#> $y
#> [1] "a"
#>
#> [[3]]
#> [1] TRUE
attributes(list_example)
#> $names
#> [1] "x" "y" ""
```

Agenda

1 Data structures

2 Atomic vectors

3 Attributes

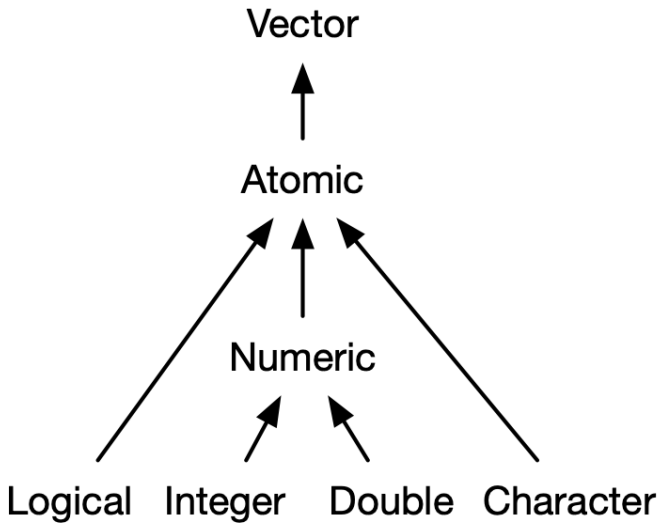
4 S3 objects

5 Lists

6 Data frames and tibbles

7 Subsetting

Atomic vectors



- Four primary types of atomic vectors: `logical`, `integer`, `double`, and `character`.
- `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 **scalar** value:

- **Logical**:
  - `TRUE` or `FALSE`
  - `T` or `F` (abbreviated version).
- **Doubles**:
  - Decimal (`0.1234`), scientific (`1.23e4`), or hexadecimal (`0xcafe`) form.
  - Special values unique to doubles: `Inf`, `-Inf`, and `NaN` (not a number).
- **Integers**:
  - Integer number followed by `L` (`1234L`, `1e4L`, or `0xcafeL`).
- **Strings**:
  - Surrounded by `"` or `'` (for example, `"hi"`, `'bye'`).
  - Special characters escaped with `\`.[3]

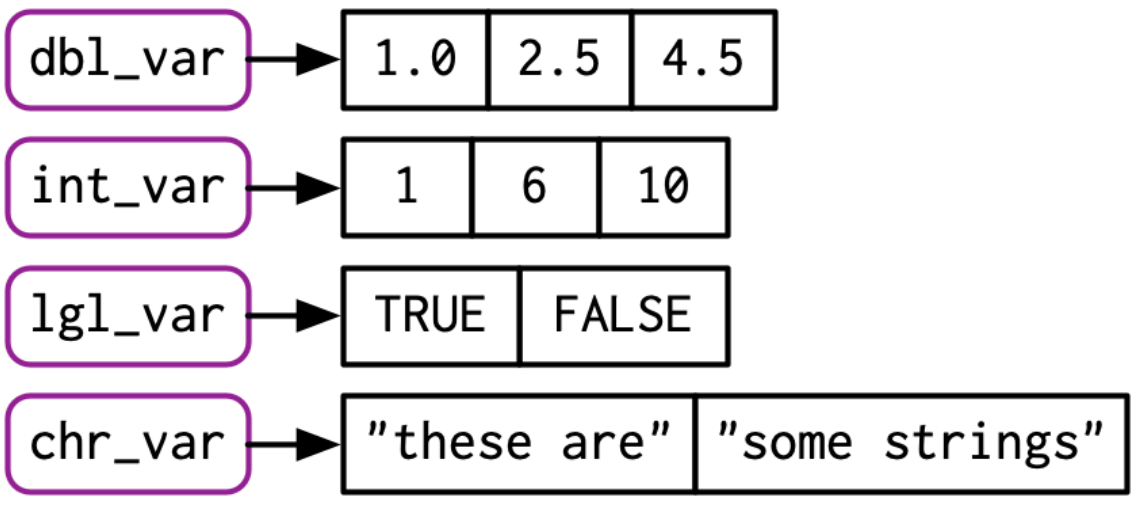
[3] see `?Quotes` for details.

Making longer vectors with `c()`

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

```
dbl_var <- c(1, 2.5, 4.5)
int_var <- c(1L, 6L, 10L)
lgl_var <- c(TRUE, FALSE)
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 `typeof()` and `length()` of a vector.
- Use `is.<type>()` to test if a vector is of a given `<type>`:
  - `is.logical()`, `is.integer()`, `is.double()`, and `is.character()`.

```
typeof(lgl_var)
#> [1] "logical"
length(lgl_var)
#> [1] 2
c(is.logical(lgl_var), is.integer(lgl_var))
#> [1]  TRUE FALSE
```

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

Coercion

- When combining different types, **coercion** is applied in a fixed order: `character` → `double` → `integer` → `logical`.

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

- Deliberately coerce values using `as.<type>()`:
  - `as.logical()`, `as.integer()`, `as.double()`, or `as.character()`.
- Failed coercion of strings → warning and missing value.

```
as.integer(c("1", "1.5", "a"))
#> [1]  1  1 NA
```

- Coercion 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))
#> [1] 1.000 0.333
```

Missing or unknown values

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

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

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

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

Missing or unknown values (cont'd)

- Propagation of missing values 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
```

Caution

Technically there are four missing value types, one for each of the atomic types: `NA` (logical), `NA_integer_` (integer), `NA_real_` (double), and `NA_character_` (character). This distinction is usually unimportant because `NA` will be automatically coerced to the correct type when needed.

Agenda

Attributes

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

- They are built on top of atomic vectors by adding attributes.
- For instance, you can add names to a vector:

```r
# 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")
```

- 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.

Dimensions

- The `dim` attribute allows a vector 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
attributes(a)
#> $dim
#> [1] 2 3
```

- Or use the assignment form of `dim()`:

```
# 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
```

Vectors and matrices

Vector	Matrix
`names()`	`rownames()`, `colnames()`
`length()`	`nrow()`, `ncol()`
`c()`	`rbind()`, `cbind()`
—	`t()`
`is.null(dim(x))`	`is.matrix()`

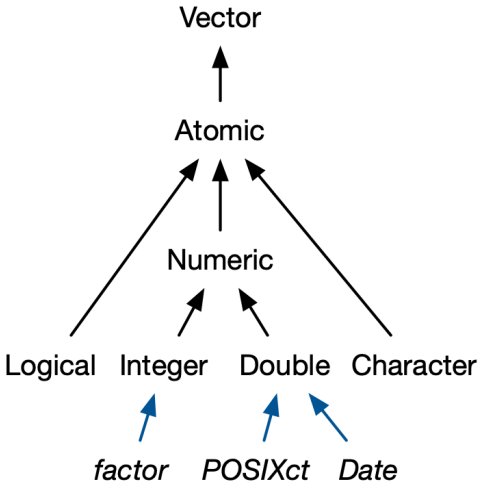
- A vector without a `dim` is often thought of as 1-dimensional, but actually has `NULL` dimensions.
- You can have matrices with a single row or single column:
  - May print similarly, but behave differently.
  - Differences not important, but useful to know they exist.
  - Use `str()` to reveal the 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
```

Agenda

S3 objects

- The `class` attribute:
  - Turns a vector into an **S3 object**, which behaves differently, e.g.
    - Categorical data, where values come from a fixed set of levels: **factor** vectors.
    - Dates, i.e. times at a daily resolution: **Date** vectors.
  - Every S3 object
    - is built on top of a base type,
    - stores additional information in other attributes.

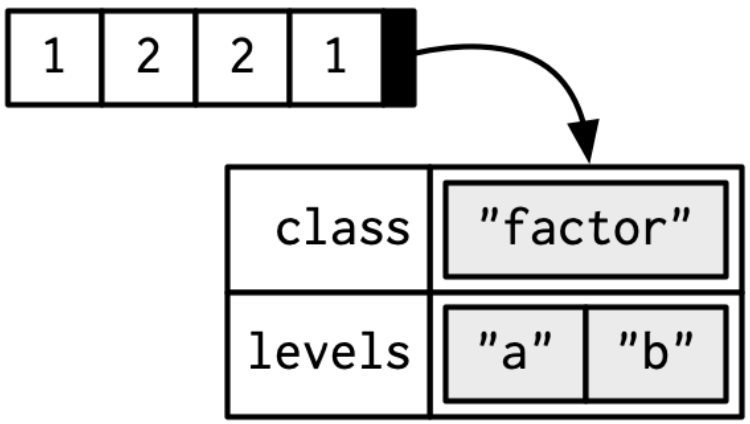


Factors

- A vector that can contain only predefined values.
- Used to store categorical data.
- Built on top of an integer vector with two attributes:
  - `class` (defines a different from integer vectors behaviour),
  - `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 are not all present in a dataset.
- When tabulating a factor, you'll get counts of all categories, even unobserved ones:

```
sex_chr <- rep("f", 3)
table(sex_chr)
#> sex_chr
#> f
#> 3
```

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

- **Ordered factors** behave like regular factors, but the order of the levels is meaningful (e.g., low, medium, high)

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

Dates

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

```
today <- Sys.Date() # Get the current system date
today
#> [1] "2024-09-23"
typeof(today)
#> [1] "double"
attributes(today)
#> $class
#> [1] "Date"
```

- Value of the `double` is the number of days since `"1970-01-01"`:[4]

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

[4] Known as the Unix Epoch.

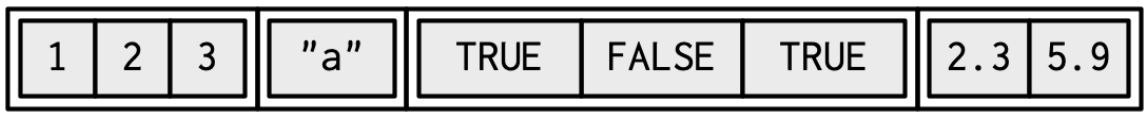
Agenda

Lists

- Each element can be any <type>.

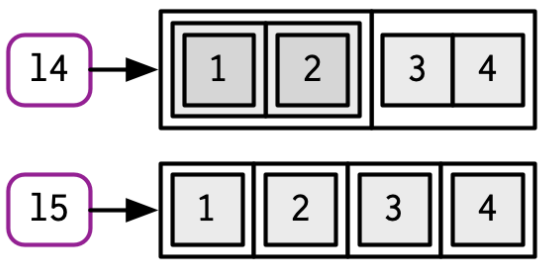
```
l1 <- list(
  1:3,
  "a",
  c(TRUE, FALSE, TRUE),
  c(2.3, 5.9)
)
typeof(l1)
#> [1] "list"
```

```
is.list(l1)
#> [1] TRUE
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
```



- c() combines 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
```

Agenda

1 Data structures

2 Atomic vectors

3 Attributes

4 S3 objects

5 Lists

6 Data frames and tibbles

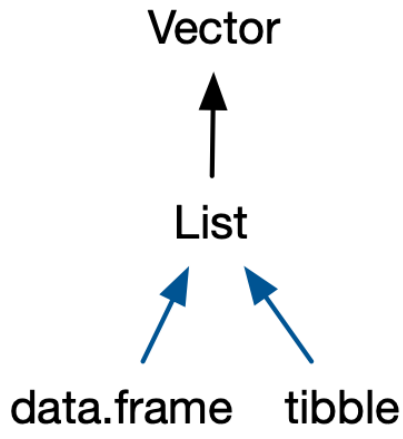
7 Subsetting

Data frames and tibbles

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

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

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



- Similar to a list, but the lengths of all component are equal.
- "Rectangular structure":
  - Share properties of both matrices and lists.
  - Has `rownames()`/`colnames()`/`names()`(= column names).
  - Has `nrow()`/`ncol()`/`length()` (= number of columns).

Data frames and tibbles (cont'd)

- Data frames:
  - One of the biggest and most important ideas in R, but ...
  - 20 years have passed since their creation,
  - which leads to the creation of the `tibble`, a modern version.
- **Main differences**: tibbles are **lazier** (do less) & **safer** (complain more).
- Technically:
  - Similar to `data.frame` but the `class` includes `tbl_df`.
  - Allows tibbles to behave differently.

```
library(tibble)
df2 <- tibble(x = 1:3,
              y = letters[1:3])
typeof(df2)
#> [1] "list"
is.data.frame(df2)
#> [1] TRUE
is_tibble(df2)
#> [1] TRUE
is_tibble(df1)
#> [1] FALSE
```

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

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

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

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

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

```r
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[5], 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 <-"
```

[5]What constitutes a letter is determined by your current locale. Avoid this by sticking to ASCII characters (i.e. A-Z).

Non-syntactic names (cont'd)

- Add a back tick to override these rules and use any name:

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

`Asset Price` <- 10e3
`Asset Price`
#> [1] 10000
```

- Avoid deliberately creating such names!
  - You'll see them data in 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
```

```
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 in `tibble()`:
#> ! Tibble columns must have compatible sizes.
#> * Size 4: Existing data.
#> * Size 2: Column `y`.
#> i Only values of size one are recycled.
data.frame(x = 1:4, y = 1:3)
#> Error in data.frame(x = 1:4, y = 1:3): arguments imply differing number of r
```

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
#> 22           5.1         3.7          1.5         0.4     setosa
#> 23           4.6         3.6          1.0         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 Skyw~    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 Darth Vad~    202   136 none       white      yellow           41.9
#>  5 Leia Orga~    150    49 brown      light      brown            19
#>  6 Owen Lars     178   120 brown, gr~ light      blue             52
#>  7 Beru Whit~    165    75 brown      light      blue             47
#>  8 R5-D4          97    32 <NA>       white, red red              NA
#>  9 Biggs Dar~    183    84 black      light      brown            24
#> 10 Obi-Wan K~    182    77 auburn, w~ fair       blue-gray        57
#> # i 77 more rows
#> # i 7 more variables: sex <chr>, gender <chr>, homeworld <chr>,
#> #   species <chr>, films <list>, vehicles <list>, starships <list>
```

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

Agenda

1 Data structures

2 Atomic vectors

3 Attributes

4 S3 objects

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.
- Next few slides, subset an atomic vector with:
  - Positive integers.
  - Negative integers.
  - Logical vectors.
  - Character vectors.

[for atomic vectors (cont'd)

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

```
x[c(3, 1)]
#> [1] 3.3 2.1
order(x)
#> [1] 1 3 2 4
x[order(x)] # Equivalent to sort(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):

```
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?
  - **Recycling rule:** the shorter length is recycled to the longest.
  - Convenient and easy to understand when `x` or `y` has length one, but avoid for other lengths because of inconsistencies in base R.

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

- A missing value in the index always yields an `NA` in the output:

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

[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])) # letters = c("a", "b", "c", "d", ..., "z")
#>   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 lists

- Exactly as for atomic vectors.
  - `[` always returns a list;
  - `[[` and `$` let you pull out elements of a list.

[for matrices

- Subset matrices 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.
  - Notice the use of blank subsetting!

```
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
```

- By default, [simplifies to the lowest possible dimensionality:

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

```
a[1, 1]
#> A
#> 1
```

[for matrices (cont'd)

- Can subset with a vector as if they were 1D.

```
vals <- outer(1:4, 1:4, FUN = "paste", sep = ",")
vals
#>      [,1]  [,2]  [,3]  [,4]
#> [1,] "1,1" "1,2" "1,3" "1,4"
#> [2,] "2,1" "2,2" "2,3" "2,4"
#> [3,] "3,1" "3,2" "3,3" "3,4"
#> [4,] "4,1" "4,2" "4,3" "4,4"
vals[c(4, 15)]
#> [1] "4,1" "3,4"
```

- Can also subset a matrix with a `matrix` of integers.
  - Each row in the `matrix` specifies the location of a value.
  - Each column in the `matrix` corresponds to a dimension.
  - The result is a vector of values.

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

```
vals[select]
#> [1] "1,1" "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.

```
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
```

[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
```

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

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

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

Selecting a single element from a list

The other two subsetting operators:

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

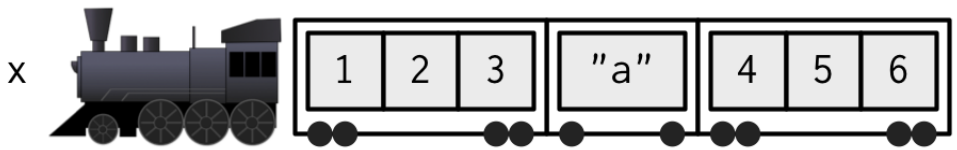
[[

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

  If the #rstats 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, Twitter, 2012-11-13

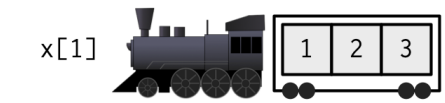
- Use this metaphor to make a simple list:

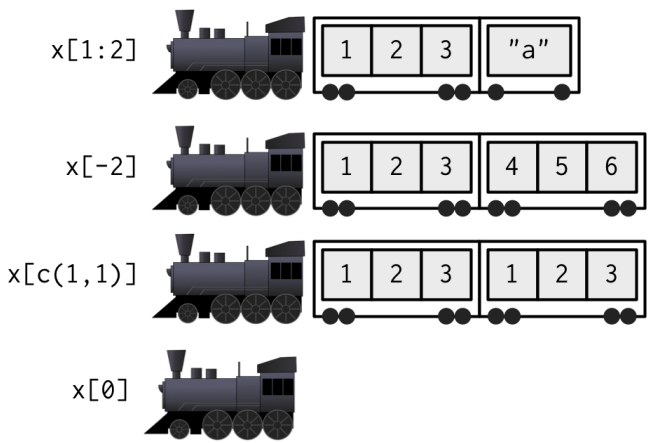
```
x <- list(1:3, "a", 4:6)
```



[[(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"
mtcars$var # Doesn't work - mtcars$var translated to mtcars[["var"]]
#> NULL
mtcars[[var]] # Instead use [[
#>  [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
```

- Important difference between `$` and `[[`: (left-to-right) partial matching!

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

Data frames and tibbles again

- Data frames have two undesirable subsetting behaviours.
  - 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$x`:
    - If there is no column `x`, selects any variable that starts with `x`.
    - If no variable starts with `x`, returns `NULL`.
    - Easy to select a wrong variable or a variable that doesn't exist.
- Tibbles tweak these behaviours:
  - `[` always returns a tibble.
  - `$` doesn't do partial matching and warns if it can't find a variable.

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

```
df2 <- tibble(xyz = "a")
str(df2$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)`.

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

```
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.
  - Coerce all values in `mtcars` into integers:

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

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