MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450
Advanced R programming

Week 1: Foundations of R programming

arp.numbat.space

Outline

First things first

Expectations

- You know R and RStudio
- You have a basic understanding of programming (for loops, if statements, functions)
- You can use Git and GitHub (https://happygitwithr.com)

Unit resources

- Everything on **https://arp.numbat.space**
- Assignments submitted on Github Classroom
- Discussion on Ed

GitHub

- Use your monash edu address.
- Apply to GitHub Global Campus as a student (https://education.github.com).
- Gives you free access to private repos and GitHub Copilot.
- Add GitHub Copilot to RStudio settings.

Outline

R history

- S (1976, Chambers, Becker and Wilks; Bell Labs, USA)
- S-PLUS (1988, Doug Martin; Uni of Washington, USA)
- R (1993, Ihaka and Gentleman; Uni of Auckland, NZ)

R history

- S (1976, Chambers, Becker and Wilks; Bell Labs, USA)
- S-PLUS (1988, Doug Martin; Uni of Washington, USA)
- R (1993, Ihaka and Gentleman; Uni of Auckland, NZ)

R influenced by

- Lisp (functional programming, environments, dynamic typing)
- Scheme (functional programming, lexical scoping)
- S and S-PLUS (syntax)

Why R?

- Free, open source, and on every major platform.
- A diverse and welcoming community
- A massive set of packages, often cutting-edge.
- Powerful communication tools (Shiny, Rmarkdown, quarto)
- RStudio IDE
- Deep-seated language support for data analysis.
- A strong foundation of functional programming.
- Posit
- Easy connection to high-performance programming languages like C, Fortran, and C++.

R challenges

- R users are not usually programmers. Most R code by ordinary users is not very elegant, fast, or easy to understand.
- R users more focused on results than good software practices.
- R packages are inconsistent in design.
- R can be slow.

Outline

Exercises

1. Given the following data frame, how do I create a new column called "3" that contains the sum of `1` and `2`? You may only use `$`, not `[[`. What makes `1`, `2`, and `3` challenging as variable names?

```
df <- data.frame(runif(3), runif(3))
names(df) <- c(1, 2)
```

2. In the following code, how much memory does `y` occupy?

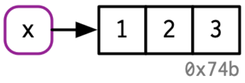
```
x <- runif(1e6)
y <- list(x, x, x)
```

3. On which line does `a` get copied in the following example?

```
a <- c(1, 5, 3, 2)
b <- a
b[[1]] <- 10
```

Binding basics

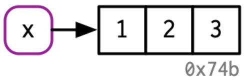
```r
x <- c(1, 2, 3)
```



- Creates an object, a vector of values, `c(1, 2, 3)`.
- Binds that object to a name, `x`.
- A name is a reference (or pointer) to a value.

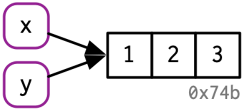
Binding basics

```
x <- c(1, 2, 3)
```



- Creates an object, a vector of values, `c(1, 2, 3)`.
- Binds that object to a name, `x`.
- A name is a reference (or pointer) to a value.

```
y <- x
```



- Binds the same object to a new name, `y`.

Binding basics

```
library(lobstr)
obj_addr(x)
```

```
[1] "0x55baa524b958"
```

```
obj_addr(y)
```

```
[1] "0x55baa524b958"
```

These identifiers are long, and change every time you restart R.

Syntactic names

A **syntactic** name:

- must consist of letters, digits, `.` and `_`
- can't begin with `_`, or a digit, or a `.` followed by a digit
- can't be a **reserved word** like `TRUE`, `NULL`, `if`, and `function`

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

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

Syntactic names

A **syntactic** name:

- must consist of letters, digits, `.` and `_`
- can't begin with `_`, or a digit, or a `.` followed by a digit
- can't be a **reserved word** like `TRUE`, `NULL`, `if`, and `function`

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

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

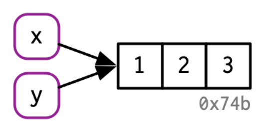
It's possible to override these rules using backticks.

```
`_abc` <- 1
`_abc`
```

Copy-on-modify

Consider the following code. It binds `x` and `y` to the same underlying value, then modifies `y`.

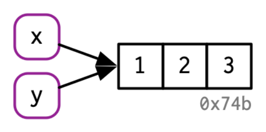
```
x <- c(1, 2, 3)
y <- x
```



Copy-on-modify

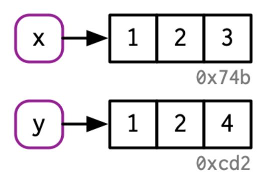
Consider the following code. It binds `x` and `y` to the same underlying value, then modifies `y`.

```
x <- c(1, 2, 3)
y <- x
```




```
y[[3]] <- 4
x
```

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



tracemem()

You can see when an object gets copied using `tracemem()`.

```
x <- c(1, 2, 3)
tracemem(x)
```

```
[1] "<0x55baa5128208>"
```

```
y <- x
y[[3]] <- 4L
```

```
tracemem[0x55baa5128208 -> 0x55baa1a59798]: eval eval eval_with_user_handlers withVisible wit
```

```
y[[3]] <- 5L
```

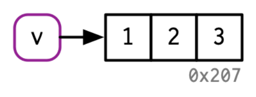
```
tracemem[0x55baa1a59798 -> 0x55baa4fa0578]: eval eval eval_with_user_handlers withVisible wit
```

```
untracemem(x)
```

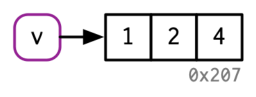
Modify-in-place

If an object has a single name bound to it, R will modify it in place:

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




```
v[[3]] <- 4
```



Function calls

The same rules for copying also apply to function calls.

```
f <- function(a) {
  a
}

x <- c(1, 2, 3)
tracemem(x)
```

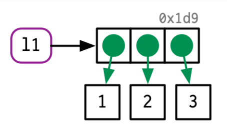
```
[1] "<0x55baa4f8ee68>"
```

```
z <- f(x)
# there's no copy here!
untracemem(x)
```

Lists

Lists store references to their elements, not the elements themselves.

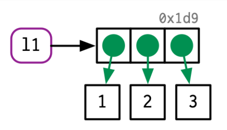
```
l1 <- list(1, 2, 3)
```



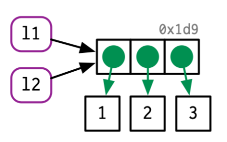
Lists

Lists store references to their elements, not the elements themselves.

```
l1 <- list(1, 2, 3)
```



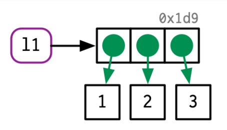

```
l2 <- l1
```



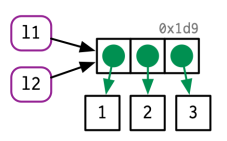
Lists

Lists store references to their elements, not the elements themselves.

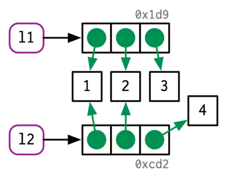
```
l1 <- list(1, 2, 3)
```




```
l2 <- l1
```



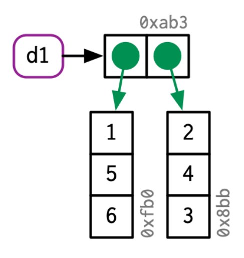

```
l2[[3]] <- 4
```



Data frames

Data frames are lists of vectors.

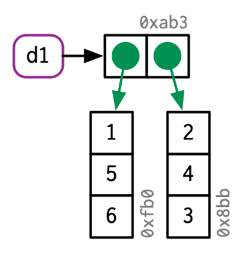
```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```



Data frames

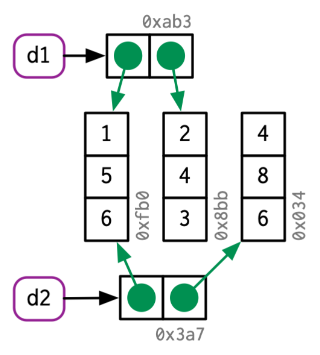
Data frames are lists of vectors.

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```



Modifying a column:

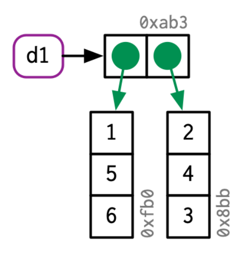
```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
```



Data frames

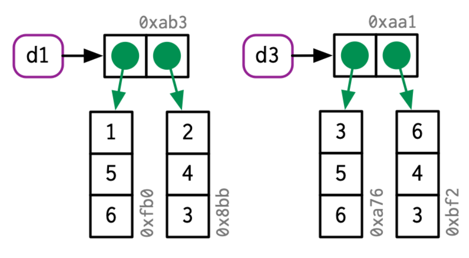
Data frames are lists of vectors.

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```



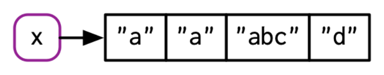
Modifying a row:

```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
```

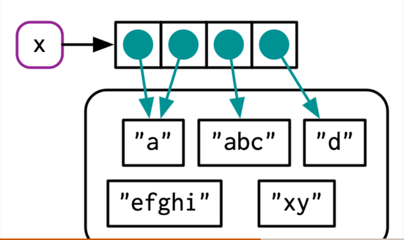


Character vectors

```r
x <- c("a", "a", "abc", "d")
```



- Not quite!
- R actually uses a **global string pool** where each element is a pointer to a string in the pool



Object size

`lobstr::obj_size()` gives the size of an object in memory.

```
obj_size(ggplot2::diamonds)
```

```
3.46 MB
```

```
banana <- "bananas bananas bananas"
obj_size(banana)
```

```
136 B
```

```
obj_size(rep(banana, 100))
```

```
928 B
```

Object size

```
x <- runif(1e6)
obj_size(x)
```

```
8.00 MB
```

```
y <- list(x, x, x)
obj_size(y)
```

```
8.00 MB
```

```
obj_size(x, y)
```

```
8.00 MB
```

ALTREP

```r
obj_size(1:3)
```

```
680 B
```

```r
obj_size(1:1e6)
```

```
680 B
```

```r
obj_size(c(1:1e6, 10))
```

```
8.00 MB
```

```r
obj_size(2 * (1:1e6))
```

```
8.00 MB
```

For loops

Loops have a reputation for being slow, but often that is caused by iterations creating copies.

```
x <- data.frame(matrix(runif(3 * 1e4), ncol = 3))
medians <- vapply(x, median, numeric(1))
tracemem(x)
```

```
for (i in seq_along(medians)) {
  x[[i]] <- x[[i]] - medians[[i]]
}
```

```
tracemem[0x55baa51f7048 -> 0x55baa5180ca8]: eval eval eval_with_user_handlers withVisible wit
tracemem[0x55baa5180ca8 -> 0x55baa5180d98]: [[<-.data.frame [[<- eval eval eval_with_user_han
tracemem[0x55baa5180d98 -> 0x55baa5180e38]: eval eval eval_with_user_handlers withVisible wit
tracemem[0x55baa5180e38 -> 0x55baa5181018]: [[<-.data.frame [[<- eval eval eval_with_user_han
tracemem[0x55baa5181018 -> 0x55baa5181108]: eval eval eval_with_user_handlers withVisible wit
tracemem[0x55baa5181108 -> 0x55baa51811f8]: [[<-.data.frame [[<- eval eval eval_with_user_han
```

- Each iteration copies the data frame two times!

For loops

The same problem but with a list.

```
y <- as.list(x)
tracemem(y)
```

```
for (i in 1:3) {
  y[[i]] <- y[[i]] - medians[[i]]
}
```

```
tracemem[0x55baa516e558 -> 0x55baa51443b8]: eval eval eval_with_user_handlers withVisible wit
```

- Only one copy created

Don’t allocate memory in a for loop

```
# Allocating memory within the loop
system.time(
{
  x <- NULL
  for(i in seq(1e5)) {
    x <- c(x, i)
  }
}
)
```

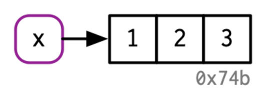
```
   user  system elapsed
  6.358   0.008   6.366
```

```
# Allocating memory before the loop
system.time(
{
  x <- numeric(1e5)
  for(i in seq(1e5)) {
    x[i] <- i
  }
}
)
```

```
   user  system elapsed
  0.006   0.000   0.006
```

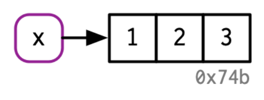
Unbinding and the garbage collector

```
x <- 1:3
```

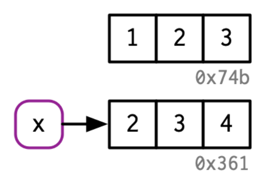


Unbinding and the garbage collector

```r
x <- 1:3
```

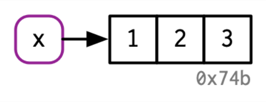



```r
x <- 2:4
```

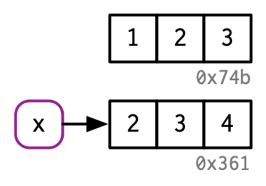


Unbinding and the garbage collector

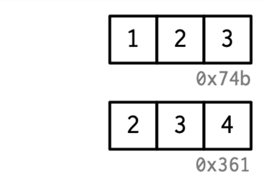
```r
x <- 1:3
```




```r
x <- 2:4
```




```r
rm(x)
```

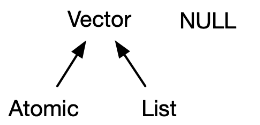


Garbage collection

- Garbage collection (GC) frees up memory by deleting R objects that are no longer used, and by requesting more memory from the operating system if needed.
- R traces every object that's reachable from the global environment (recursively).
- GC runs automatically whenever R needs more memory to create a new object.
- You can force garbage collection by calling `gc()`. But it's *never* necessary.

Outline

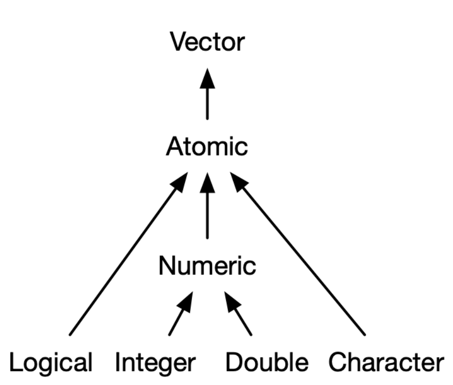
Vectors



- Vectors come in two flavours: atomic vectors and lists
- For atomic vectors, all elements must have same type
- For lists, elements can have different types
- `NULL` is like a generic zero length vector
- Scalars are just vectors of length 1
- Every vector can also have **attributes**: a named list of arbitrary metadata.
- The **dimension** attribute turns vectors into matrices and arrays.
- The **class** attribute powers the S3 object system.

Atomic vectors

- Four primary types of atomic vectors: logical, integer, double, and character (which contains strings).
- Collectively integer and double vectors are known as numeric vectors
- Two rare types:
  - complex
  - raw.



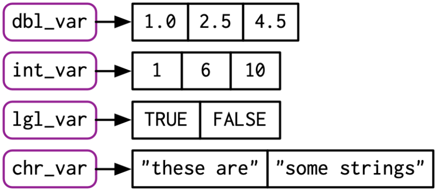
Scalars

- Logicals: `TRUE` or `FALSE`, or abbreviated (`T` or `F`).
- Doubles: decimal (`0.1234`), scientific (`1.23e4`), or hexadecimal (`0xcafe`). Special values: `Inf`, `-Inf`, and `NaN` (not a number).
- Integers: `1234L`, `1e4L`, or `0xcafeL`. Can not contain fractional values.
- Strings: `"hi"` or `'bye'`. Special characters are escaped with `\`.

Making longer vectors with `c()`

Use `c()` to create longer vectors from shorter ones.

```
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")
```



When the inputs are atomic vectors, `c()` always flattens.

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

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

Types and length

You can determine the type of a vector with `typeof()` and its length with `length()`.

```
typeof(lgl_var)
```

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

```
typeof(int_var)
```

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

```
typeof(dbl_var)
```

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

```
typeof(chr_var)
```

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

Missing values

Most computations involving a missing value will return another missing value.

```r
NA > 5
```

```
[1] NA
```

```r
10 * NA
```

```
[1] NA
```

```r
!NA
```

```
[1] NA
```

Missing values

Exceptions:

```r
NA ^ 0
```

```
[1] 1
```

```r
NA | TRUE
```

```
[1] TRUE
```

```r
NA & FALSE
```

```
[1] FALSE
```

Missing values

Use `is.na()` to check for missingness

```
x <- c(NA, 5, NA, 10)
x == NA
```

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

```
is.na(x)
```

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

There are actually four missing values: `NA` (logical), `NA_integer_` (integer), `NA_real_` (double), and `NA_character_` (character).

Coercion

- For atomic vectors, all elements must be the same type.
- When you combine different types they are **coerced** in a fixed order: logical → integer → double → character.

```
str(c("a", 1))
```

```
 chr [1:2] "a" "1"
```

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
```

```
[1] 0 0 1
```

```
sum(x)
```

```
[1] 1
```

```
as.integer(c("1", "1.5", "a"))
```

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

Exercises

4. Predict the output of the following:

```
c(1, FALSE)
c("a", 1)
c(TRUE, 1L)
```

5. Why is `1 == "1"` true? Why is `-1 < FALSE` true? Why is `"one" < 2` false?

6. Why is the default missing value, `NA`, a logical vector? What's special about logical vectors? (Hint: think about `c(FALSE, NA_character_)`.)

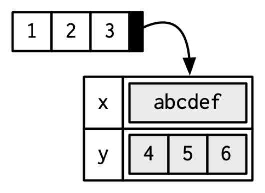
Getting and setting attributes

- You can think of attributes as name-value pairs that attach metadata to an object.
- Individual attributes can be retrieved and modified with `attr()`, or retrieved en masse with `attributes()`, and set en masse with `structure()`.

```
a <- 1:3
attr(a, "x") <- "abcdef"
a
```

```
[1] 1 2 3
attr(,"x")
[1] "abcdef"
```

Getting and setting attributes




```
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
)
str(attributes(a))
```

```
List of 2
 $ x: chr "abcdef"
 $ y: int [1:3] 4 5 6
```

Names

- Names are a type of attribute.
- 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"))
```

```
x
```

```
a b c
1 2 3
```

Names

- Avoid using `attr(x, "names")` as it requires more typing and is less readable than `names(x)`.
- You can remove names from a vector by using `x <- unname(x)` or `names(x) <- NULL`.

Dimensions

- Adding a `dim` attribute to a vector allows it to behave like a 2-dimensional **matrix** or a multi-dimensional **array**.
- You can create matrices and arrays with `matrix()` and `array()`, or by using the assignment form of `dim()`:

```r
# Two scalar arguments specify row and column sizes
x <- matrix(1:6, nrow = 2, ncol = 3)
x
```

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

Dimensions

```r
# One vector argument to describe all dimensions
y <- array(1:12, c(2, 3, 2))
y
```

```
, , 1

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

, , 2
```

Dimensions

```r
# You can also modify an object in place by setting dim()
z <- 1:6
dim(z) <- c(3, 2)
z
```

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

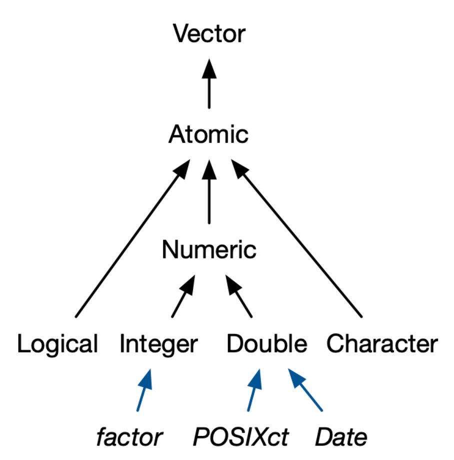
Exercises

7. What does `dim()` return when applied to a 1-dimensional vector?
8. When might you use `NROW()` or `NCOL()`?
9. How would you describe the following three objects? What makes them different from `1:5`?

```
x1 <- array(1:5, c(1, 1, 5))
x2 <- array(1:5, c(1, 5, 1))
x3 <- array(1:5, c(5, 1, 1))
```

S3 atomic vectors

- `class` is a vector attribute.
- It turns object into **S3 object**.
- Four important S3 vectors:
  - **factor** vectors.
  - **Date** vectors with day resolution.
  - **POSIXct** vectors for date-times.
  - **difftime** vectors for durations.



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`, "factor", and `levels`, which defines the set of allowed values.

```
x <- factor(c("a", "b", "b", "a"))
x
```

```
[1] a b b a
Levels: a b
```

Factors

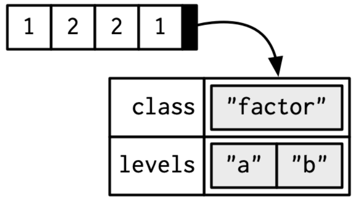
```
typeof(x)
```

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

```
attributes(x)
```

```
$levels
[1] "a" "b"

$class
[1] "factor"
```



Factors

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
```

```
sex_char
m
3
```

```
table(sex_factor)
```

```
sex_factor
m f
3 0
```

Factors

- Be careful: some functions convert factors to integers!
- ggplot preserves ordering of levels in graphs – useful to reorder panels or legends.
- Ordered factors are useful when the order of levels is meaningful.

```
grade <- ordered(c("b", "b", "a", "c"), levels = c("c", "b", "a"))
grade
```

```
[1] b b a c
Levels: c < b < a
```

Dates

- Date vectors are built on top of double vectors.
- Class “Date” with no other attributes:

```
today <- Sys.Date()

typeof(today)
```

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

```
attributes(today)
```

```
$class
[1] "Date"
```

Dates

The value of the double (which can be seen by stripping the class), represents the number of days since 1970-01-01 (the "Unix Epoch").

```
date <- as.Date("1970-02-01")
unclass(date)
```

```
[1] 31
```

Date-times

- Base R provides two ways of storing date-time information, POSIXct, and POSIXlt.
- “POSIX” is short for Portable Operating System Interface
- “ct” stands for calendar time; “lt” for local time
- POSIXct vectors are built on top of double vectors, where the value represents the 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"
```

Date-times

The `tzone` attribute controls only how the date-time is formatted; it does not control the instant of time represented by the vector. Note that the time is not printed if it is midnight.

```
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"
```

Exercises

10. What sort of object does `table()` return? What is its type? What attributes does it have? How does the dimensionality change as you tabulate more variables?

11. What happens to a factor when you modify its levels?

```
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
```

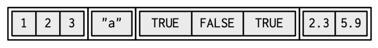
12. What does this code do? How do `f2` and `f3` differ from `f1`?

```
f2 <- rev(factor(letters))
f3 <- factor(letters, levels = rev(letters))
```

Lists

- More complex than atomic vectors
- Elements are *references* to objects of any type

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




```
[1] "list"
```

```
str(l1)
```

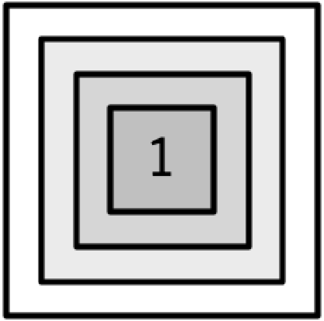
```
List of 4
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : num [1:2] 2.3 5.9
```

Lists

- Lists can be recursive: a list can contain other lists.

```
l3 <- list(list(list(1)))
str(l3)
```

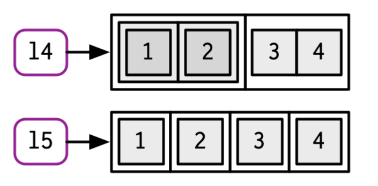
```
List of 1
 $ :List of 1
  ..$ :List of 1
  .. ..$ : num 1
```



Lists

- `c()` will combine several lists into one.

```r
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
```

```r
str(l5)
```

```
List of 4
 $ : num 1
 $ : num 2
 $ : num 3
 $ : num 4
```

Testing and coercion

```
list(1:3)
```

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

```
as.list(1:3)
```

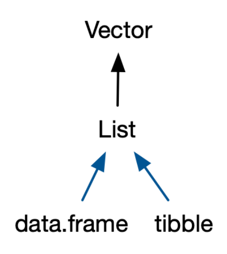
```
[[1]]
[1] 1

[[2]]
[1] 2

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

- You can turn a list into an atomic vector with `unlist()`.

Data frames and tibbles



- Most important S3 vectors built on lists: data frames and tibbles.

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

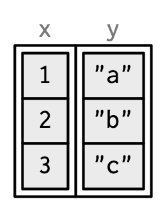
```
[1] "list"
```

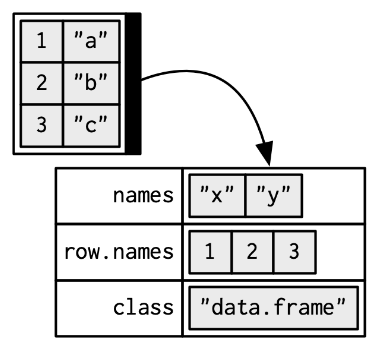
```r
attributes(df1)
```

```
$names
[1] "x" "y"

$class
[1] "data.frame"

$row.names
[1] 1 2 3
```





Data frames and tibbles

- A data frame has a constraint: the length of each of its vectors must be the same.
- A data frame has `rownames()` and `colnames()`. The `names()` of a data frame are the column names.
- A data frame has `nrow()` rows and `ncol()` columns. The `length()` of a data frame gives the number of columns.

Tibbles

- Modern reimagining of the data frame.
- tibbles are “lazy and surly”: they do less and complain more.

```
library(tibble)
df2 <- tibble(x = 1:3, y = letters[1:3])
typeof(df2)
```

```
[1] "list"
```

```
attributes(df2)
```

```
$class
[1] "tbl_df"     "tbl"        "data.frame"

$row.names
[1] 1 2 3

$names
[1] "x" "y"
```

Creating data frames and tibbles

```
names(data.frame(`1` = 1))
```

```
[1] "X1"
```

```
names(tibble(`1` = 1))
```

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

Creating data frames and tibbles

```r
data.frame(x = 1:4, y = 1:2)
```

```
  x y
1 1 1
2 2 2
3 3 1
4 4 2
```

```r
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.
```

Creating data frames and tibbles

```
tibble(
  x = 1:3,
  y = x * 2,
  z = 5
)
```

```
# A tibble: 3 x 3
      x     y     z
  <int> <dbl> <dbl>
1     1     2     5
2     2     4     5
3     3     6     5
```

Row names

Data frames allow you to label each row with a name, a character vector containing only unique values:

```r
df3 <- data.frame(
  age = c(35, 27, 18),
  hair = c("blond", "brown", "black"),
  row.names = c("Bob", "Susan", "Sam")
)
df3
```

```
      age  hair
Bob    35 blond
Susan  27 brown
Sam    18 black
```

Row names

- tibbles do not support row names
- convert row names into a regular column with either `rownames_to_column()`, or the `rownames` argument:

```
as_tibble(df3, rownames = "name")
```

```
# A tibble: 3 x 3
  name    age hair
  <chr> <dbl> <chr>
1 Bob      35 blond
2 Susan    27 brown
3 Sam      18 black
```

Printing

```
dplyr::starwars
```

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

Printing

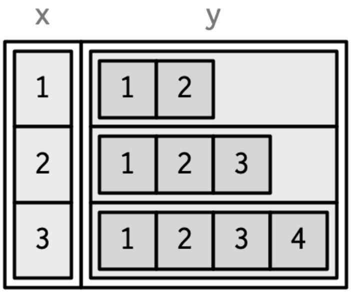
- Tibbles only show first 10 rows and all columns that fit on screen. Additional columns shown at bottom.
- Each column labelled with its type, abbreviated to 3–4 letters.
- Wide columns truncated.

List columns

```r
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)
```

```r
data.frame(
  x = 1:3,
  y = I(list(1:2, 1:3, 1:4))
)
```

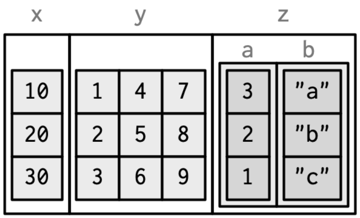
```
  x          y
1 1       1, 2
2 2    1, 2, 3
3 3 1, 2, 3, 4
```




```r
tibble(
  x = 1:3,
  y = list(1:2, 1:3, 1:4)
)
```

Matrix and data frame columns

```
dfm <- tibble(
  x = 1:3 * 10,
  y = matrix(1:9, nrow = 3),
  z = data.frame(a = 3:1, b = letters[1:3])
)
str(dfm)
```




```
tibble [3 x 3] (S3: tbl_df/tbl/data.frame)
 $ x: num [1:3] 10 20 30
 $ y: int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
 $ z:'data.frame':  3 obs. of  2 variables:
  ..$ a: int [1:3] 3 2 1
  ..$ b: chr [1:3] "a" "b" "c"
```

Exercises

13. Can you have a data frame with zero rows? What about zero columns?
14. What happens if you attempt to set rownames that are not unique?
15. If `df` is a data frame, what can you say about `t(df)`, and `t(t(df))`? Perform some experiments, making sure to try different column types.
16. What does `as.matrix()` do when applied to a data frame with columns of different types? How does it differ from `data.matrix()`?

NULL

```r
length(NULL)
```

```
[1] 0
```

You can test for `NULL`s with `is.null()`:

```r
x <- NULL
x == NULL
```

```
logical(0)
```

```r
is.null(x)
```

```
[1] TRUE
```