

INTRODUCTION TO DATA ANALYSIS

BASICS OF R

LEARNING GOALS

- ▶ become familiar with R, its syntax and basic notions
- ▶ become familiar with the key functionality from the tidyverse
- ▶ understand and write simple R scripts
- ▶ be able to write documents in Rmarkdown



BASE R

- ▶ special purpose programming language for statistical computing
 - ▶ statistics, data mining, visualization ...
- ▶ first released in 1993 as a descendant of S
- ▶ free (GNU General Public License)
- ▶ authority says: **do not treat R as a programming language**
 - ▶ rather a tool optimized for manipulating, plotting and analyzing data

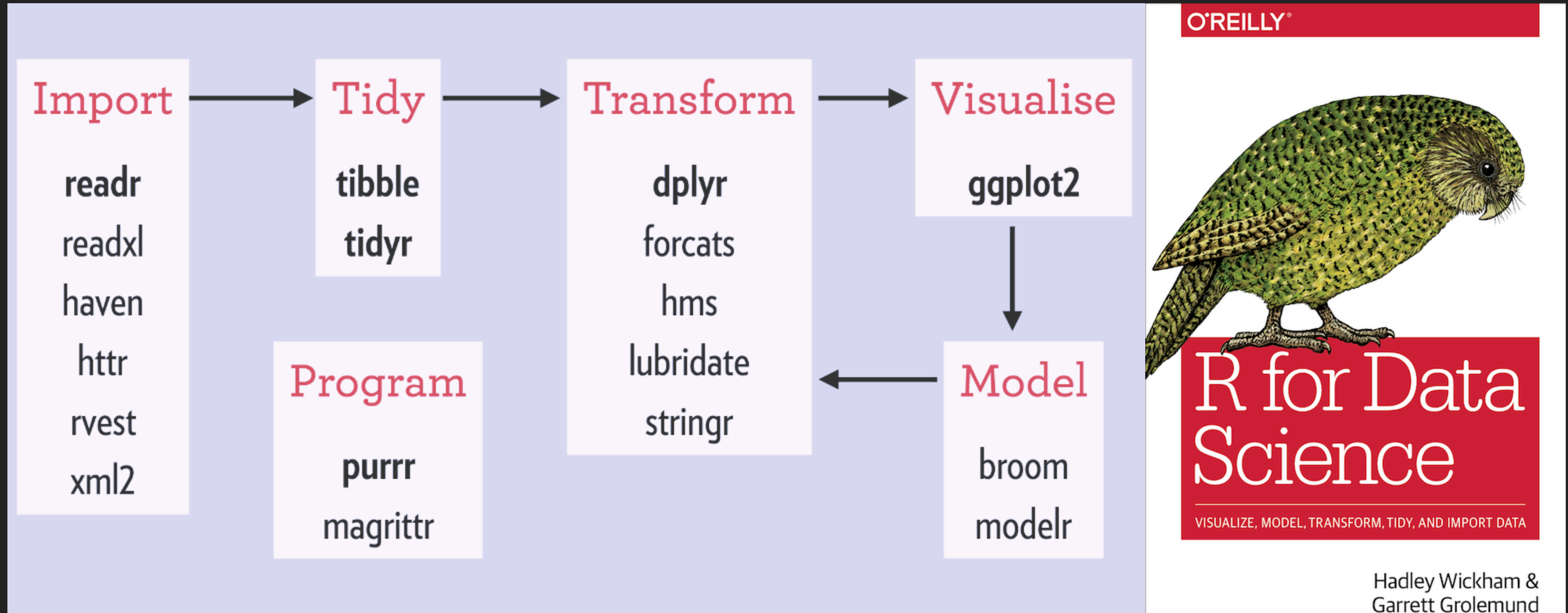
PACKAGES

- ▶ highly extensible via package
 - ▶ official package repository is **CRAN**
 - ▶ additional bleeding-edge packages, e.g., from GitHub

```
# install package (once)  
install.packages("PACKAGE-NAME")  
  
# load package (for every use)  
library(PACKAGE-NAME)  
  
# call a function from a package  
# without loading it  
PACKAGE-NAME::FUNCT-IN-PACKAGE(...)
```

```
# download/update package for this course  
devtools::install_github("n-kall/IDA2019-package")  
  
# load it  
library(IDA2019)
```

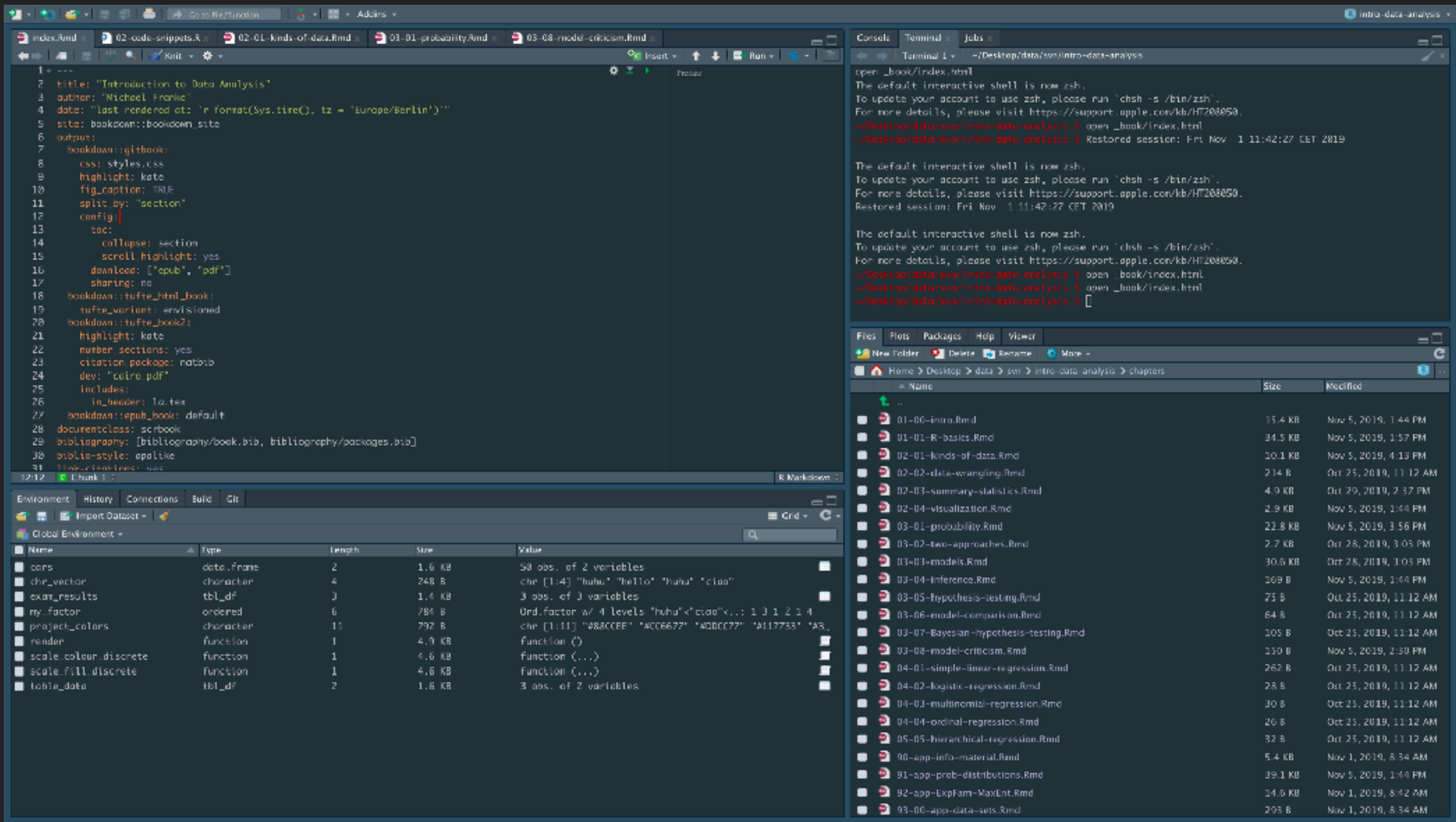
THE TIDYVERSE



INTRODUCTION TO DATA ANALYSIS

RSTUDIO

► use unless highly opinionated!



FIRST STEPS IN R

- ▶ interpreted language
 - ▶ evaluate code line-by-line when developing
 - ▶ store code in **scripts** for reuse
- ▶ multi-paradigmatic language
 - ▶ supports object-oriented programming (but we will not use it (much))
 - ▶ supports **functional programming** (and we will use it a lot)

FUNCTIONS

- ▶ function calls
 - ▶ named and unnamed arguments
 - ▶ default values

```
round(x = 0.138, digits = 2)  # works as intended
round(digits = 2, x = 0.138)  # works as intended
round(0.138, digits = 2)      # works as intended
round(0.138, 2)               # works as intended
round(x = 0.138, 2)           # works as intended
round(digits = 2, 0.138)      # works as intended
round(2, x = 0.138)           # works as intended
round(2, 0.138)               # does not work as intended (returns 2)
```


FUNCTIONS

- ▶ function calls
 - ▶ prefix vs infix notation

```
# both of these calls sum 1, 2, and 3 together  
sum(1,2,3)      # prefix notation  
1 + 2 + 3       # infix notation
```

VARIABLES

- ▶ variable variable assignment
 - ▶ use of `=` is discouraged

```
x <- 6      # assigns 6 to variable x
7 -> y      # assigns 7 to variable y
z = 3       # assigns 3 to variable z
x * y / z   # returns 6 * 7 / 3 = 14
```

GETTING HELP

► internal documentation

```
# documentation for function `lm`  
help(lm)  
# two equivalent ways for obtaining help on search term 'linear'  
help.search("linear")  
??linear
```


DATA TYPES

- ▶ numeric
 - ▶ integer, double, complex
- ▶ logical (= Boolean)
- ▶ special values
 - ▶ `NA`, `NaN`, `Inf`, `NULL`
- ▶ character (= string)
- ▶ factor
- ▶ list
- ▶ data frame / tibble

```
typeof(3)           # returns type "double"  
typeof(TRUE)        # returns type "logical"  
typeof(cars)         # returns 'list' (includes data.frames, tibbles,  
                     objects, ...)  
typeof("huhu")       # return 'character' (= string)  
typeof(mean)         # return 'closure' (= function)  
typeof(c)            # return 'builtin' (= deep system internal stuff)  
typeof(round)         # returns type "special" (= well, special stuff?)
```

▶ casting

```
# casting Boolean value `TRUE` into number format  
as.numeric(TRUE)    # returns 1
```

VECTORS

- ▶ as violable default, expect everything to be a vector-like
 - ▶ single numbers have a ``length``
 - ▶ you can access index `[2]` of a variable storing a single number

```
length(7)  # returns 1
x <- 7
x[1]       # returns 7
x[2] <- 10 # works
```


VECTORS

- ▶ creating numeric vectors
 - ▶ indexing starts at 1

```
# creating a vector by hand
x <- c(4, 7, 1, 1)  # this is now a 4-place vector
x[2]               # value at second position (returns 7)

# creating sequences of numbers
1:10               # returns 1, 2, 3, ..., 10
seq(from = 1, to = 10, by = 1)  # returns 1, 2, 3, ..., 10
seq(from = 1, to = 10, by = 0.5) # returns 1, 1.5, 2, ..., 9.5, 10
seq(from = 0, to = 1, length.out = 11) # returns 0, 0.1, ..., 0.9, 1
```

VECTORS

- ▶ as violable default, expect every operation to apply to vectors

```
x <- c(4, 7, 1, 1)  # this is now a 4-place vector
x + 1               # returns [5, 8, 2, 2]

y <- c(TRUE, FALSE) # Boolean vector
!y                  # negation applies to vector
                    # returns [FALSE, TRUE]
```

MATRICES

- ▶ numeric matrices are two-dimensional vectors
 - ▶ column-major mode

```
# matrices are column-major mode  
matrix(1:4, nrow = 2)  
# 1 3  
# 2 4  
  
# indices are as usual: [row, col]  
matrix(1:4, nrow = 2)[1,2]  
# returns 3
```


MATRICES

- ▶ numeric matrices are two-dimensional vectors
 - ▶ column-major mode

```
# matrices are column-major mode  
matrix(1:4, nrow = 2)  
# 1 3  
# 2 4  
  
# indices are as usual: [row, col]  
matrix(1:4, nrow = 2)[1,2]  
# returns 3
```

CHARACTER VECTORS

- ▶ check ``stringr`` package for useful character functions

```
# character vector of first names
first_names <- c("Jax", "Jamie", "Jason")
# character vector of last names
last_names <- c("Teller", "Lannister", "Stackhouse")
# string concatenation (from `stringr` package)
str_c(first_names, last_names, sep = " ")
# return
# [1] "Jax Teller"
# [2] "Jamie Lannister"
# [3] "Jason Stackhouse"
```

```
# three measures of reaction time in a single string
reaction_times = "123|234|345"
# notice that we need to doubly (!) escape character |
# notice also that the results is a list (see below)
str_split(reaction_times, "\\|", n = 3)
```

CHARACTER VECTORS

- ▶ check ``stringr`` package for useful character functions

```
# three measures of reaction time in a single string  
reaction_times = "123|234|345"  
# notice that we need to doubly (!) escape character |  
# notice also that the results is a list (see below)  
str_split(reaction_times, "\\|", n = 3)
```


FACTORS

- factors store information about instances from a finite set of discrete categories (ordered or unordered)

```
# unordered factor from character vector
chr_vector = c("huhu", "hello", "huhu", "ciao")
factor(chr_vector)
## [1] huhu hello huhu ciao
## Levels: ciao hello huhu

# ordered factor
factor(
  chr_vector,    # the vector to treat as factor
  ordered = T,   # make sure its treated as ordered factor
  levels = c("huhu", "ciao", "hello") # specify order of levels
)
## [1] huhu hello huhu ciao
## Levels: huhu < ciao < hello
```

FACTORS

- ▶ adding elements to a factor must respect known categories

```
# adding to a factor must respect known categories
chr_vector = c("huhu", "hello", "huhu", "ciao")
my_factor <- factor(
  chr_vector,      # the vector to treat as factor
  ordered = T,     # make sure its treated as ordered factor
  levels = c("huhu", "ciao", "hello") # specify order of levels
)
my_factor[5] <- "huhu" # adding a "known category" is okay
my_factor[6] <- "moin" # adding an "unknown category" does not work
my_factor
## [1] huhu  hello huhu  ciao  huhu  <NA>
## Levels: huhu < ciao < hello
```

FACTORS

► useful functions for factors from `forcats` package

```
# show function `fct_expand` in action
chr_vector = c("huhu", "hello", "huhu", "ciao")
my_factor <- factor(
  chr_vector,      # the vector to treat as factor
  ordered = T,     # make sure its treated as ordered factor
  levels = c("huhu", "ciao", "hello") # specify order of levels
)
my_factor[5] <- "huhu" # adding a "known category" is okay
my_factor <- fct_expand(my_factor, "moin") # add new category
my_factor[6] <- "moin" # adding new item now works
my_factor
## [1] huhu hello huhu ciao huhu moin
## Levels: huhu < ciao < hello < moin
```


FACTORS

- ▶ useful functions for factors from `forcats` package

```
my_factor           # original factor
fct_rev(my_factor)  # reverse level order
fct_relevel(         # manually supply new level order
  my_factor,
  c("hello", "ciao", "huhu")
)
```

LISTS

► named vectors ≠ lists

```
# lists are arbitrary key-value pairs
my_list <- as.list(named_vec)
my_list[["Jamie"]] <- c("top", "notch") # add, e.g., a vector as value
my_list
# returns
## $Jax
## [1] 1
##
## $Jamie
## [1] "top" "notch"
## ...
```

```
# named vectors require same data type for each element
named_vec <- c("Jax" = 1, "Jamie" = 2, "Jason" = 3) # works
named_vec[2]
# returns
## Jamie
##      2
```

LISTS

► nested lists are possible

```
## nested lists are possible
my_list = list(
  single_number = 42,
  chr_vector    = c("huhu", "ciao"),
  nested_list   = list(x = 1, y = 2, z = 3)
)
my_list
## $single_number
## [1] 42
##
## $chr_vector
## [1] "huhu" "ciao"
##
## $nested_list
## $nested_list$x
## [1] 1
##
## $nested_list$y
## [1] 2
##
## $nested_list$z
## [1] 3
```


LISTS

► ways of accessing list elements

```
# all of these return the same list element  
my_list$chr_vector  
my_list[["chr_vector"]]  
my_list[[2]]
```

DATA FRAMES

- ▶ data frames store data
 - ▶ data frames are essentially lists where all elements have the same length
 - ▶ "rectangular data"
 - ▶ we can use indexing like in a matrix

```
# gives the value of the cell in row 2, column 3
exp_data[2,3] # return 133
```

```
# fake experimental data
exp_data = data.frame(
  trial = 1:5,
  condition = factor(
    c("C1", "C2", "C1", "C3", "C2"),
    ordered = T
  ),
  response = c(121, 133, 119, 102, 156)
)
exp_data
```

```
# returns
##   trial condition response
## 1     1        C1      121
## 2     2        C2      133
## 3     3        C1      119
## 4     4        C3      102
## 5     5        C2      156
```

DATA FRAMES & TIBBLES

- ▶ tibbles are data frames in the tidyverse
- ▶ some differences:
 - ▶ different output format
 - ▶ different encoding defaults
 - ▶ no “strings as factors” default
 - ▶ dynamic construction possible

```
my_tibble = tibble(x = 1:10, y = x^2) # dynamic construction possible
my_dataframe = data.frame(x = 1:10, y = x^2) # ERROR :/
```

```
# cast the data.frame as tibble
as_tibble(exp_data)
```

```
#returns
```

```
## # A tibble: 5 x 3
```

```
##   trial condition response
```

```
##   <int> <ord>         <dbl>
```

```
## 1     1 C1           121
```

```
## 2     2 C2           133
```

```
## 3     3 C1           119
```

```
## 4     4 C3           102
```

```
## 5     5 C2           156
```


FUNCTIONS

- ▶ many useful predefined functions (obviously)
 - ▶ skim docs and cheat sheets for inspiration
 - ▶ continuously expand your inventory

2.3.1.1 Standard logic

- `&` : “and”
- `|` : “or”
- `!` : “not”
- `negate()` : a pipe-friendly `!` (see Section 2.5 for more on piping)
- `all()` : returns true of a vector if all elements are `T`
- `any()` : returns true of a vector if at least on element is `T`

2.3.1.2 Comparisons

- `<` : smaller
- `>` : greater
- `==` : equal (you can also use `near()` instead of `==` e.g. `near(3/3,1)` returns TRUE)
- `>=` : greater or equal
- `<=` : less or equal
- `!=` : not equal

2.3.1.3 Set theory

- `%in%` : wheter an element is in a vector
- `union(x,y)` : union of `x` and `y`
- `intersect(x,y)` : intersection of `x` and `y`
- `setdiff(x,y)` : all elements in `x` that are not in `y`

2.3.1.4 Sampling and combinatorics

- `runif()` : random number from unit interval [0;1]
- `sample(x, size, replace)` : take `size` samples from `x` (with replacement if `replace` is `T`)
- `choose(n,k)` : number of subsets of size `n` out of a set of size `k` (binomial coefficient)

DEFINING CUSTOM FUNCTIONS

► named functions with/without default values

```
# define a new function
# takes two numbers x & y as argument
# return x * y + 1
cool_function = function(x, y) {
  return(x * y + 1)
}

# apply `cool_function` to some numbers:
cool_function(3,3)      # return 10
cool_function(1,1)      # return 2
cool_function(1:2,1)    # returns vector [2,3]
cool_function(1)         # throws error: 'argument "y" is missing, with no default'
cool_function()          # throws error: 'argument "x" is missing, with no default'
```

```
# default values for each argument
cool_function_2 = function(x = 2, y = 3) {
  return(x * y + 1)
}

# apply `cool_function_2` to some numbers:
cool_function_2(3,3)    # return 10
cool_function_2(1,1)    # return 2
cool_function_2(1:2,1)  # returns vector [2,3]
cool_function_2(1)      # returns 4 (= 1 * 3 + 1)
cool_function_2()       # returns 7 (= 2 * 3 + 1)
```


DEFINING CUSTOM FUNCTIONS

► anonymous functions (for local use)

```
# define a function that takes a function as argument
new_applier_function = function(input, function_to_apply) {
  return(function_to_apply(input))
}

# sum vector with built-in & named function
new_applier_function(
  input = 1:2,           # input vector
  function_to_apply = sum # built-in & named function to apply
) # returns 3

# sum vector with anonymous function
new_applier_function(
  input = 1:2,           # input vector
  function_to_apply = function(input) {
    return(input[1] + input[2])
  }
) # returns 3 as well
```


FOR-LOOPS

- create container -> each loop fills content

```
# fix a vector to transform
input_vector = 1:6
# create output vector for memory allocation
output_vector = integer(length(input_vector))
# iterate over length of input
for (i in 1:length(input_vector)) {
  # multiply by 10 if even
  if (input_vector[i] %% 2 == 0) {
    output_vector[i] = input_vector[i] * 10
  }
  else {
    output_vector[i] = input_vector[i]
  }
}
output_vector

# returns
## [1]  1 20  3 40  5 60
```

MAPPING

- ``map_`` functions are tidyverse dialect for base R's ``apply``

```
map_dbl(  
  input_vector,  
  function(i) {  
    if (input_vector[i] %% 2 == 0) {  
      return (input_vector[i] * 10 )  
    }  
    else {  
      return (input_vector[i])  
    }  
  }  
)
```

```
# returns  
## [1]  1 20  3 40  5 60
```

```
# same with concise notation from `purrr` package  
map_dbl(  
  input_vector,  
  ~ ifelse( .x %% 2 == 0, .x * 10, .x)  
)  
  
# returns  
## [1]  1 20  3 40  5 60
```

SEQUENCING OPERATIONS

- ▶ pipe operator `%>%` from the `magrittr` package

```
# define input
input_vector = c(0.4, 0.5, 0.6)

## 'center-embedding' approach
mean(round(input_vector)) # first round, then take mean

## 'named throughput' approach
tmp = round(input_vector) # intermediate result
mean(tmp)                 # final operation

## 'piping' approach
input_vector %>% round %>% mean
```


HOMEWORK FOR NEXT CLASS

- ▶ reread Chapter 2 of course notes
- ▶ glance at some Rmarkdown tutorial
- ▶ skim Chapter 3 (to the extent that it exists)
- ▶ [**voluntary**] do this experiment **before Friday 12:15** (takes ca. 5 minutes)
 - ▶ [clickable link to experiment](#)