GR5206: lecture 1

Computational Statistics
And Introduction to Data Science

Thibault Vatter
Department of Statistics, Columbia University

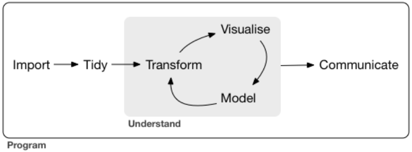
9/6/2019

Outline

Statistical computing

- Essential for modern statisticians to be fluent in **statistical computing** (i.e., statistical programming).
- At the end of this course, you will have:
  - The ability to read and write code for (statistical) data analysis.
  - An understanding of programming topics such as functions, objects, data structures, debugging.
  - Use R programming to solve common statistical tasks, i.e., graphics, regression, testing, . . .

What you will learn



- **Import** data from the web, a database, a stored file, etc.
- **Wrangle**:
  - **Tidy**: usually means that rows/columns are observations/variables.
  - **Transform**: narrowing in on observations of interests, creating new variables, calculating summary statistics.
- **Analyze**:
  - **Visualize**:
    - E.g., show unexpected things, or raise new questions.
    - Doesn't scale well as it requires human interpretation.
  - **Model**:
    - Sufficiently precise questions can be answered with a model.
    - Mathematical/computational tools generally scale well.
    - Even when it doesn't, computers are usually cheaper than brains!
- **Communicate** your results.
- Surrounding all these tools is **programming**.

- What's the difference between data science and statistics?

  "A data scientist is just a sexier word for statistician."

  — Nate Silver (outdated)

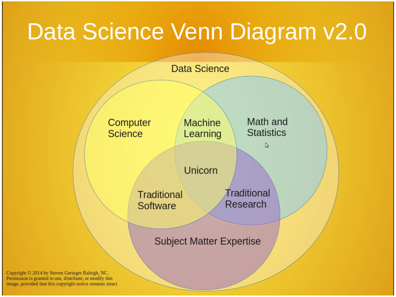
  "A data scientist is a better computer scientist than a statistician and is a better statistician than a computer scientist."

  — Unknown (still accurate)

- What does a data scientist do?
  - There is not one correct answer.
  - Transform data into valuable information!
  - A data scientist spends a significant portion of time processing data and less time modeling data.

What is Data Science?

- **Wikipedia:** "the extraction of knowledge from data"
- Precise definition a bit unclear and controversed...
- Practitioners "agree" on the components of data science:
  - database management
  - gathering and cleaning
  - exploratory analysis
  - predictive modeling
  - data summary and visualization

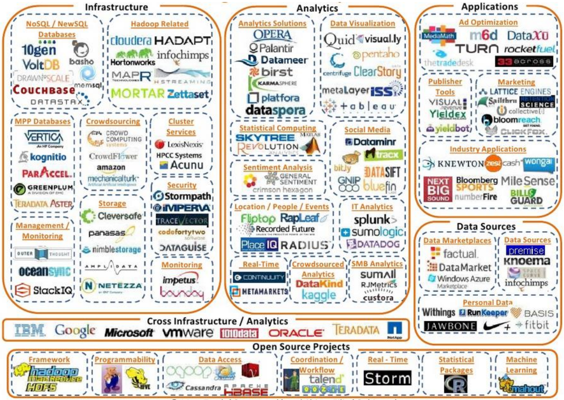


Applications

Some of the hiring partners of *The Data Incubator*

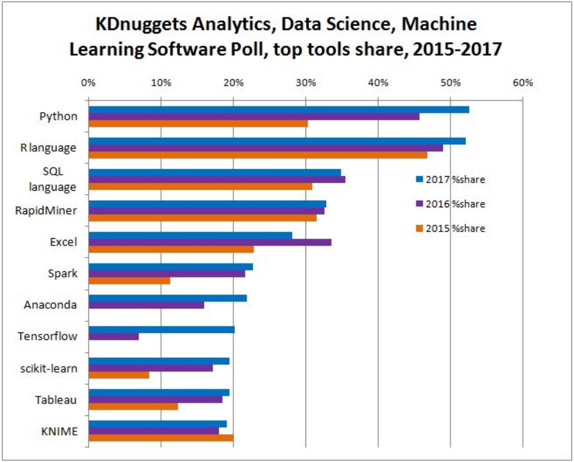
- E-marketing
- Recommender systems
- Sport analytics
- Biotechnology
- Image or speech recognition
- Fraud and risk detection
- Social media
- Credit scoring
- E-commerce
- Government analysis
- Gaming
- Price comparisons
- Airline routes planing
- Delivery logistics

Technology ecosystem



source: rosebt.com

Most popular?



source: kdnuggets.com

Outline

Some notes on the course

- Prerequisite:
  - GR5204 Statistical Inference (or GU4204)
  - GR5205 Linear Regression Models (or GU4205)
- Warning
  - If you haven't taken (aren't taking) these courses, you'll have to do extra work at times to catch up.
- Other prerequisites:
  - Basic knowledge of linear algebra, multivariate calculus, and probability.
  - We will review some topics in class but can't cover everything!
  - If you haven't taken a linear algebra course, expect to spend some extra time catching up.

Honor Code

Columbia's intellectual community relies on academic integrity and responsibility as the cornerstone of its work. Graduate students are expected to exhibit the highest level of personal and academic honesty as they engage in scholarly discourse and research. ***In practical terms, you must be responsible for the full and accurate attribution of the ideas of others in all of your research papers and projects; you must be honest when taking your examinations; you must always submit your own work and not that of another student, scholar, or internet source.*** *Graduate students are responsible for knowing and correctly utilizing referencing and bibliographical guidelines."*

- More on Columbia's academic integrity webpage.
- Failure to observe these rules will have serious academic consequences, up to and including dismissal from the university.

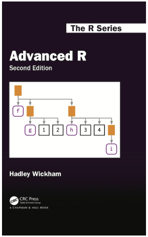
Course Description

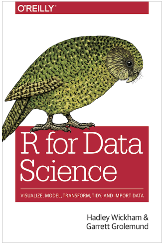
Columbia University in the City of New York

Friday 2:40 PM - 5:25 PM,
417 International Affairs

- Instructor: Thibault Vatter
  - Office Building: 1255 Amsterdam Ave, Room 901B (SSW, 9th floor)
  - Office Hours: by appointment.
  - E-mail: tv2233\protect\T1\textdollar@\protect\T1\textdollarcolumbia.edu (use [GR5206] in the title of the email)
- Teaching Assistants:
  - Jitong Qi (jq2273)
  - Julien Boussard (jb4365)
  - Yichuan Luo (yl4073)
  - Chirong Zhang (cz2533)

Resources





- Advanced R (Hadley Wichkham)
- R for data science (Garrett Grolemund and Hadley Wickham)
- Rstudio cheat sheets
- The CRAN website

- Most of the material in the slides is taken from the two books.
- They are available online for free, and the slides will be on CourseWorks.

Evaluations

- 40% HWs
- 30% Midterm
- 30% Final
- Remarks:
  - Grades based on academic performance only.
  - Please write [GU5206] in the subject heading of all e-mail correspondence with instructor/TA.
  - Disability services
  - Academic integrity

Points	Grade
93 or more	A
90 to 92	A-
87 to 89	B+
83 to 86	B
80 to 82	B-
77 to 79	C+
70 to 76	C
60 to 69	D
Below 59	F

(A+ to the top 2 students, assuming they get at least 93.)

Midterm & exam

- Code challenges to be completed on your laptop.
- Material allowed:
  - Open-book.
  - No cellphone or anything that allows you to use internet.
  - Your laptop will be "locked" once you start the exam (more on that later).
- Midterm will be given in class.
- No make-up exams will be given.
  - If you can't make it for religious reasons, let me know in advance.
  - If you can't make it for medical reasons, let me know as soon as possible and bring a doctor's note afterwards.

Homeworks

- There will be one weekly HW.
- Collaboration is allowed in solving the problems, but each student should submit in her or his own solutions.
- Homework must be completed on Ed before the deadline (more on that later).
- HW cannot be submitted to the TA by e-mail.
- No late homework accepted.
- If you miss a due date for medical reasons, let me know as soon as possible and bring a doctor's note afterwards.
- Lowest score will be dropped.

Tentative outline

Date	Topic	Reference
09/06	Data structures and subsetting	Adv-R chapters 3 and 4
09/13	Control flows and functions	Adv-R chapters 5 and 6
09/20	Functional programming	Adv-R chapters 9, 10 and 11
09/27	Object-oriented programming	Adv-R chapters 12 and 13
10/04	Simulation and optimization	GR5204
10/11	Distributions as model	GR5204
10/18	**Midterm exam**	
10/25	Wrangling I	R4DS chapters 5, 9, 10, 11, 12 and 18
11/01	Visualization	R4DS chapter 3 and 20
11/08	Wrangling II	R4DS chapters 13, 14, 15 and 16
11/15	Modeling I	GR5205, R4DS chapters 21, 22, 23, 24 and 25
11/22	Modeling II	GR5205, R4DS chapters 21, 22, 23, 24 and 25
12/06	**Final exam**	

CourseWorks

Class announcements/e-mails will be made/sent from and important material available in CourseWorks:

- You are expected to check the course page regularly.
- A copy of the most recently updated syllabus will be on CourseWorks.
- Occasionally, there will be other course related handouts posted in CourseWorks.
- Lecture slides will be posted on Courseworks, but blackboard lectures will not.
- **You are responsible for making sure that CourseWorks announcements/e-mails are going to an e-mail you check at least once daily.**

Ed

Demo!

Ed's forum

- Ask questions!
  - The best way to get answers is to ask!
  - Use Ed rather than emailing me/the TA so everyone can benefit from the response (or to get answers from classmates who are up as late as you are).
- Edit questions and answers wiki-style.
  - Think of Piazza as a Q&A wiki for your class.
  - Every question has just a single students' answer that students can edit collectively (and a single instructors' answer for instructors).
- Add a followup to comment or ask further questions.
  - To comment on or ask further questions about a post, start a followup discussion.
  - Mark it resolved when the issue has been addressed, and add any relevant information back into the Q&A above.
- Go anonymous.

Outline

S and R

- S
  - A statistical programming language
  - First appeared in 1976
  - Developed by John Chambers and (in earlier versions) Rick Becker and Allan Wilks of Bell Labs
  - John Chambers, *[the aim is] to turn ideas into software, quickly and faithfully*
- R
  - Modern implementation of S
  - First appeared in 1993
  - Created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand
  - Currently developed by the *R Development Core Team*

Some "technical" details about R

- **Part of the GNU free software project**
- Source code written primarily in C, Fortran, and R
- **Available for Windows, macOS, and Linux**
- Multi-paradigm: object-oriented, functional, procedural
- Dynamically typed
- Scripting language (interpreted)
- **Wide variety of statistical and graphical techniques**
- **Easily extensible through functions and packages**
- **Read/write from/to various data sources**

What about Excel?

source: fantasyfootballanalytics.net

Excel is great for certain things. . .

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

	Student	Midterm	Final	Asn #1	Asn #2	Asn #3	Asn #4	Asn #5	Asn #6	V1	V2	Final Points	Letter Grade
2	Student 1	95.5	91.78	100	100	100	100	100	100	94.54	93.42	94.54	A
3	Student 2	93.2	89.04	100	100	100	100	100	100	92.48	91.23	92.48	A
4	Student 3	95.5	86.3	100	100	100	100	100	100	91.80	89.04	91.80	A
5	Student 4	94.3	86.3	100	100	100	100	100	100	91.44	89.04	91.44	A
6	Student 5	95.5	82.88	100	100	75	100	100	100	89.26	85.47	89.26	A
7	Student 6	79.5	86.3	100	100	100	100	100	100	87.00	89.04	89.04	A
8	Student 7	84.1	85.6	100	100	100	100	100	100	88.03	88.48	88.48	A
9	Student 8	94.3	80.14	100	100	100	100	100	100	88.36	84.11	88.36	A
10	Student 9	94.3	80	100	100	100	100	100	100	88.29	84.00	88.29	A
11	Student 10	89.8	82.19	100	100	100	100	100	100	88.04	85.75	88.04	A
12	Student 11	90.9	81.51	100	100	100	100	100	100	88.03	85.21	88.03	A
13	Student 12	93.2	78.77	100	100	100	100	100	100	87.35	83.02	87.35	A
14	Student 13	89.8	81.5	100	75	100	100	100	100	86.86	84.37	86.86	A
15	Student 14	86.4	81.5	100	100	100	100	100	100	86.67	85.20	86.67	A
16	Student 15	93.2	76.71	100	100	100	100	100	100	86.32	81.37	86.32	A
17	dent 16	97.7	71.23	100	100	100	100	100	100	84.93	76.98	84.93	A-
18	Student 17	86.4	76.71	100	100	100	100	100	100	84.28	81.37	84.28	A-
19	Student 18	87.5	77.4	100	100	100	100	75	100	84.12	81.09	84.12	A-
20	Student 19	90.9	75.34	100	100	100	75	100	100	84.11	79.44	84.11	A-
21	Student 20	81.8	78.77	100	100	100	100	100	100	83.93	83.02	83.93	A-
22	Student 21	76.1	78.77	100	100	100	100	100	100	82.22	83.02	83.02	B+
23	Student 22	84.1	71.92	100	100	100	100	100	100	81.19	77.54	81.19	B+
24	Student 23	85.2	71.23	100	100	100	100	100	100	81.18	76.98	81.18	B+
25	Student 24	67	76.03	100	100	100	100	100	100	78.12	80.82	80.82	B+
26	Student 25	85.2	69.18	100	100	100	100	100	100	80.15	75.34	80.15	B+
27	Student 26	81.8	70.55	100	100	100	100	100	100	79.82	76.44	79.82	B+
28	Student 27	81.8	70.55	100	100	100	100	100	100	79.82	76.44	79.82	B+

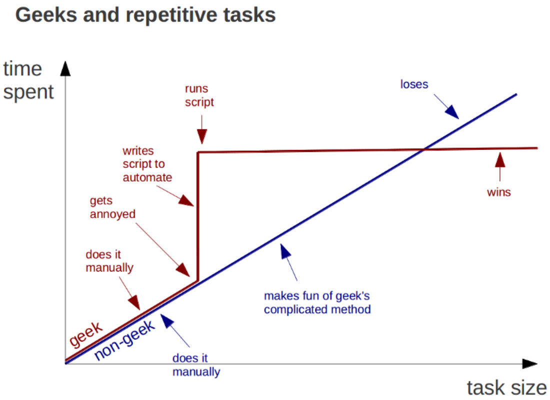
source: github.com/jdwilson4

. . . but not everything

R's advantages:

- **Easier automation**
- **Better reproducibility**
- Faster computation
- Supports larger data sets
- Reads any type of data
- More powerful data manipulation capabilities
- Easier project organization
- Easier to find and fix errors
- Free & open source
- Advanced statistics capabilities
- State-of-the-art graphics
- Runs on many platforms
- Anyone can contribute packages to improve its functionality

Automation and reproducibility



source: trendct.org

How about Python?

source: python.org

CRAN

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2017-11-30, Kite-Eating Tree) R-3.4.3.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

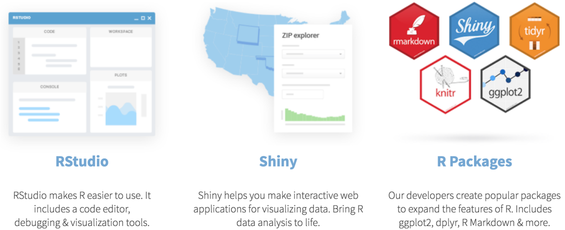
Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

source: cran.r-project.org

RStudio

- An open-source integrated development environment (IDE)
- RStudio Desktop available for Windows, macOS, and Linux



source: rstudio.com

Outline

Base R

- What is `Base R`?

  "The package named base is in a way the core of R and contains the basic functions of the language, particularly, for reading and manipulating data."

  — R for Beginners, Emmanuel Paradis

- `Base R` includes all default code for performing common data manipulation and statistical tasks.
- You might recognize some `Base R` functions:
  - `mean()`, `median()`, `lm()`, `summary()`, `sort()`
  - `data.frame()`, `read.csv()`, `cbind()`, `grep()`, `regexpr()`
  - Many many more...
- If you don't recognize any `Base R` functions, don't worry!

The tidyverse

- Common criticisms of `Base R`:
  - The code doesn't flow as well as other languages.
  - Function names/arguments are often inconsistent/confusing.
  - `Base R` functions sometimes don't return type-stable objects.
  - `Base R` functions are not refined to run as fast as possible.
  - Other complaints exist...
- So what is the tidyverse? A collection of R packages
  - designed for data science,
  - sharing an underlying design philosophy, grammar, and data structures.
- Often perform the same tasks as `Base R`, but:
  - Provides a **pipe** operator to help with the flow of the code.
  - More descriptive function names and consistent inputs.
  - Type-stable.
  - Often faster than common `Base R` functions.

Core `tidyverse` packages

- `ggplot2`: declarative graphics, based on The Grammar of Graphics.
- `dplyr`: grammar of data manipulation.
- `tidyr`: functions that help you get to tidy data.
- `readdr`: reading in rectangular data.
- `purrr`: enhancing R's functional programming (FP).
- `tibble`: a `tibble`, or `tbl_df`, is a modern reimagining of the `data.frame`.
- `stringr`: functions designed to make working with strings as easy as possible.
- `forcats`: useful tools that solve common problems with factors.

More on the tidyverse website!

Base R versus tidyverse

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- Why ever use `Base R`?
  - It gets the job done!
  - To become an expert `R` programmer, you have to know `Base R`.
  - Some `Base R` functions are very common/useful, e.g., `mean()`.
- What should you learn first? Base R or Tidyverse?
  - Some believe you should learn `Base R` first.
  - Some believe you should learn `tidyverse` first.
  - Lately, more are shifting to `tidyverse`.
  - We'll start with `Base R` for the basics and switch to the `tidyverse` for data analysis.

Outline

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:

```
# 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.

```
# 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	

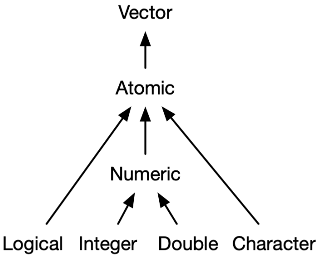
- Almost all other objects are built upon these foundations.
- R has no 0-dimensional, or scalar types.
- Best way to understand what data structures any object is composed of is `str()` (short for structure).

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

Vector

- Two flavours:
  - 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.

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**:

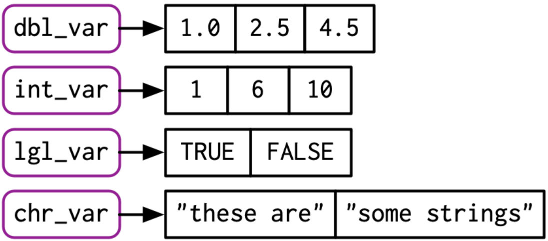
- Logicals:
  - In full (`TRUE` or `FALSE`),
  - Abbreviated (`T` or `F`).
- Doubles:
  - Decimal (`0.1234`), scientific (`1.23e4`), or hexadecimal (`0xcafe`) form.
  - Special values unique to doubles: `Inf`, `-Inf`, and `NaN` (not a number).
- Integers:
  - Similar to doubles but
    - must be followed 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()`:

```
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

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

- 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 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
```

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 $\rightarrow$ double $\rightarrow$ integer $\rightarrow$ 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
```

Attributes

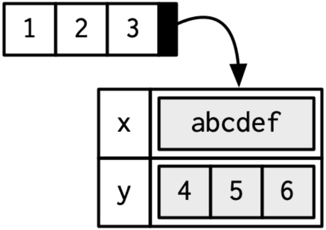
How about matrices, arrays, factors, or date-times?

- 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:

```
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

Columbia University in the City of New York

- 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.

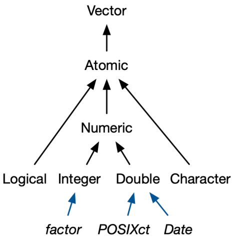
```
str(1:3)                   # 1d vector
#>  int [1:3] 1 2 3
str(matrix(1:3, ncol = 1)) # column vector
#>  int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1)) # row vector
#>  int [1, 1:3] 1 2 3
str(array(1:3, 3))         # "array" vector
#>  int [1:3(1d)] 1 2 3
```

S3 atomic vectors

- One of the most important vector attributes is `class`, which underlies the S3 object system.
  - Having a class attribute turns an object into an **S3 object** (i.e., behave differently when passed to a **generic** function).
  - Every S3 object is built on top of a base type, and stores additional information in other attributes.
  - More about the S3 object system later.

S3 atomic vectors cont'd

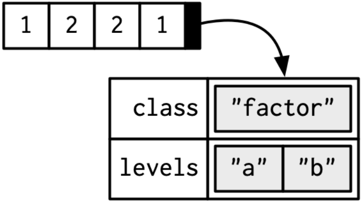
- In the next few slides, four important S3 vectors in R:
  - Categorical dat (values come from a fixed set of levels): **factor** vectors.
  - Dates (day resolution): **Date** vectors.
  - Date-times (second or sub-second resolution): **POSIXct** vectors.
  - Durations (between Dates or Date-times pairs): **difftime** 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).

```r
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:

```
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:
  - Factors are frequent because many functions (e.g. `read.csv()`/`data.frame()`) automatically convert character vectors to factors.
  - Suboptimal because there's no way to know the set of all possible levels or their correct order: the levels are a property of theory or experimental design, not of the data.
  - Use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the "theoretical" data.
- 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 (the `time_t` type in C),
  - and "lt" for local time (the `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 date-time is formatted; not the represented instant of time.
  - 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"
structure(now_ct, tzone = "Europe/Paris")
#> [1] "2018-08-02 CEST"
```

Durations

- Represent the amount of time between pairs of dates or date-times.
- Stored in `difftimes`:
  - Built on top of doubles.
  - `units` attribute determines how to interpret the integer.

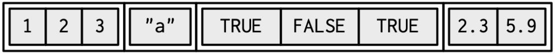
```
one_week_1 <- as.difftime(1, units = "weeks")
one_week_1
#> Time difference of 1 weeks

typeof(one_week_1)
#> [1] "double"
attributes(one_week_1)
#> $class
#> [1] "difftime"
#>
#> $units
#> [1] "weeks"
```

```
one_week_2 <- as.difftime(7, units = "days")
one_week_2
#> Time difference of 7 days

typeof(one_week_2)
#> [1] "double"
attributes(one_week_2)
#> $class
#> [1] "difftime"
#>
#> $units
#> [1] "days"
```

Lists

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

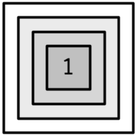
```r
l1 <- list(
  1:3,
  "a",
  c(TRUE, FALSE, TRUE),
  c(2.3, 5.9)
)

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 cont'd

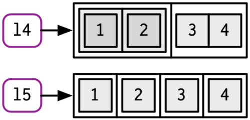
- Sometimes called **recursive** vectors:

```r
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:

```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
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()`, and coerce to a list with `as.list()`.

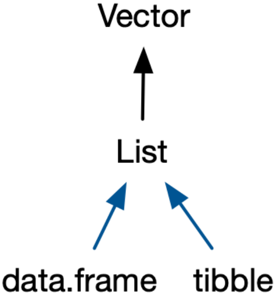
```
list(1:3)
#> [[1]]
#> [1] 1 2 3
as.list(1:3)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

- Turn a list into an atomic vector with `unlist()`, but careful:
  - Rules for the resulting type are complex, not well documented, and not always equivalent to what you'd get with `c()`.

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
```



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" explaining why they share properties of both matrices and lists:
    - It has `rownames()` and `colnames()`, but its `names()` are the column names.
    - It has `nrow()` rows and `ncol()` columns, but its `length()` is the number of columns.
- One of the biggest and most important ideas in R!
- One of the things that makes R different from many other programming languages.
- ..... but
  - 20 years have passed since their creation, and some of the design decisions that made sense at the time can now cause frustration.
  - ..... which lead to the creation of the `tibble`, a modern reimagining of the data frame.

Tibbles

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

- 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`

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

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c")
)
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

- Beware of the default conversion of strings to factors.

```
df1 <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE
)
str(df1)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

Creating a `tibble`

- Similar to creating a data frame, but tibbles never coerce their input (i.e., **lazy**):

```
df2 <- tibble(
  x = 1:3,
  y = c("a", "b", "c")
)
str(df2)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

- Next few slides: some of the differences between `data.frame()` and `tibble()`.
  - 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:

```r
`_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:

```r
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 ro
```

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.

```
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)
#> Tibble columns must have consistent lengths, only values of length one are re
#> * Length 2: Column `y`
#> * Length 4: Column `x`
```

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

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

```
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 13
#>    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 6 more variables: 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()`.

List columns

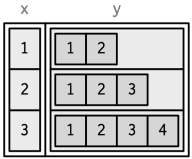
- Since a data frame is a list of vectors, it is possible for a data frame to have a column that is a list.
  - Useful because a list can contain any other object, i.e., you can put any object in a data frame.
  - Allows you to keep related objects together in a row, no matter how complex the individual objects are.
  - We'll see applications later in the course.

List columns cont'd

- In data frames, either add the list-column after creation or wrap the list in `I()`[3].
- In tibbles, easier and printed columns.

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

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
```

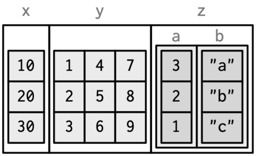
```r
tibble(
  x = 1:3,
  y = list(1:2, 1:3, 1:4)
)
#> # A tibble: 3 x 2
#>         x y
#>     <int> <list>
#> 1       1 <int [2]>
#> 2       2 <int [3]>
#> 3       3 <int [4]>
```



[3] short for identity, indicates that an input should be left as is, and not automatically transformed.

Matrix and data frame columns


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

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



- Careful:
  - Many functions that work with data frames assume that all columns are vectors.
  - The printed display can be confusing.

```r
dfm[1, ]
#>    x y.1 y.2 y.3 z.a z.b
#> 1 10   1   4   7   3   a
```

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.

```
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.

Outline

Subsetting

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

- 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 `$`.
    - The operators interact differently with different vector types.
    - Subsetting can be combined with assignment.
- Subsetting is a natural complement to `str()`:
  - `str()` shows the pieces of any object (its structure).
  - Subsetting pulls out the pieces that you're interested in.
- Outline:
  - Selecting multiple elements with `[`.
  - Selectomg a single element with `[[` and `$`.
  - 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

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

- **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?
  - 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.

```
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

Columbia University in the City of New York

- **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

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- 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

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- 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)[4].

```
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
```

[4] 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

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

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

Preserving dimensionality

Columbia University
in the City of New York

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

```
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:

```
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.

```
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:

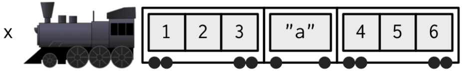
- `[[` is used for extracting single items.
- `x$y` is a useful shorthand for `x[["y"]]`.

- [[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

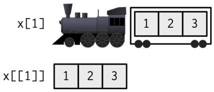
- Use this metaphor to make a simple list:

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

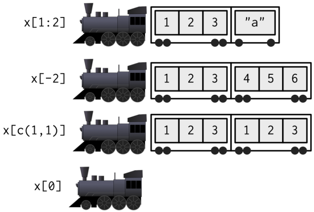


[[cont'd

- 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)
#>  Factor w/ 1 level "a": 1
```

```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
```