COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

DSFBA: Wrangling

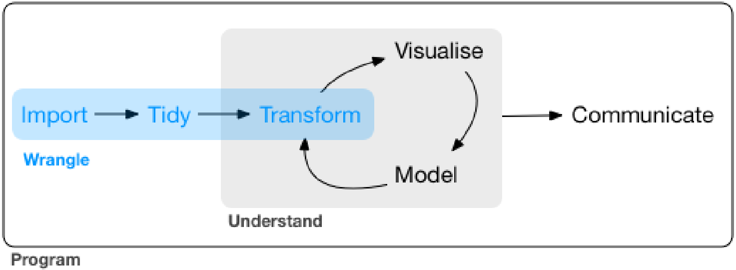
Data Science for Business Analytics

Thibault Vatter

Department of Statistics, Columbia University

11/10/2021

Today and tomorrow



Most of the material (e.g., the picture above) is borrowed from

R for data science

Outline

Outline

Warm-up

- Does every year have 365 days?
- Does every day have 24 hours?
- Does every minute have 60 seconds?

Refering to an instant in time

- Two types of date/time data:
  - A **date**.
    - Tibbles print this as `<date>`.
  - A **date-time** is a date plus a time.
    - Uniquely identifies an instant in time (typically to the nearest second).
    - Tibbles print this as `<dttm>`.
    - Elsewhere in R, `POSIXct`.
- **Use the simplest possible data type satisfying your needs!**

Creating date/times

- The **lubridate** package:
  - Makes it easier to work with dates and times in R.
  - Not part of core tidyverse because only needed when working with dates/times.

```
library(lubridate)
today()
#> [1] "2021-11-10"
now()
#> [1] "2021-11-10 09:07:40 EST"
```

- Other (usual) ways to create a date/time:
  - From a string.
  - From individual date-time components.
  - From an existing date/time object.

```
as_datetime(today())
#> [1] "2021-11-10 UTC"
as_date(now())
#> [1] "2021-11-10"
```

From a string


```
ymd("2017-01-31")
#> [1] "2017-01-31"
mdy("January 31st, 2017")
#> [1] "2017-01-31"
dmy("31-Jan-2017")
#> [1] "2017-01-31"

ymd_hms("2017-01-31 20:11:59")
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
#> [1] "2017-01-31 08:01:00 UTC"
```

- Additionally:

```
ymd(20170131)
#> [1] "2017-01-31"
ymd(20170131, tz = "UTC")
#> [1] "2017-01-31 UTC"
```

From individual components


```
flights %>%
  select(year:day, hour, minute, dep_time) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))
#> # A tibble: 336,776 x 7
#>     year month   day  hour minute dep_time departure
#>    <int> <int> <int> <dbl>  <dbl>    <int> <dttm>
#>  1  2013     1     1     5     15      517 2013-01-01 05:15:00
#>  2  2013     1     1     5     29      533 2013-01-01 05:29:00
#>  3  2013     1     1     5     40      542 2013-01-01 05:40:00
#>  4  2013     1     1     5     45      544 2013-01-01 05:45:00
#>  5  2013     1     1     6      0      554 2013-01-01 06:00:00
#>  6  2013     1     1     5     58      554 2013-01-01 05:58:00
#>  7  2013     1     1     6      0      555 2013-01-01 06:00:00
#>  8  2013     1     1     6      0      557 2013-01-01 06:00:00
#>  9  2013     1     1     6      0      557 2013-01-01 06:00:00
#> 10  2013     1     1     6      0      558 2013-01-01 06:00:00
#> # ... with 336,766 more rows
```

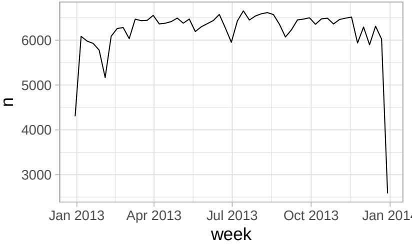
- For dep_time and others such as arr_time:

```
flights_dt <- flights %>%
  mutate(dep_time = make_datetime(
    year, month, day, dep_time %/% 100, dep_time %% 100))
```

Rounding

- Rounding:
  - `floor_date()` rounds down.
  - `round_date()` rounds to.
  - `ceiling_date()` rounds up.

```
flights_dt %>%
  filter(!is.na(dep_time)) %>%
  count(week = floor_date(dep_time, "week")) %>%
  ggplot(aes(week, n)) +
  geom_line()
```



Getting/setting the components

- Getting the components:

```
datetime <- ymd_hms("2016-07-08 12:34:56")
map_dbl(list(year, month, mday, yday, wday), function(f) f(datetime))
#> [1] 2016    7    8  190    6
```

- Setting the components:

```
year(datetime) <- 2020
datetime
#> [1] "2020-07-08 12:34:56 UTC"
month(datetime) <- 01
datetime
#> [1] "2020-01-08 12:34:56 UTC"
hour(datetime) <- hour(datetime) + 1
datetime
#> [1] "2020-01-08 13:34:56 UTC"
```

- Alternatively:

```
update(datetime, year = 2019)
#> [1] "2019-01-08 13:34:56 UTC"
```

Time spans

- Goal: to do arithmetic (i.e., subtraction, addition, and division) with dates/times.
- Three classes that represent time spans:
  - **Durations** (number of seconds).
  - **Periods** (human units like weeks and months).
  - **Intervals** (a starting and ending point).

Durations

- A **duration** always record a time span in seconds.
- Larger units created at the standard rate.
  - E.g., 60s/mn, 60mn/h, 24h/d, 7d/w, 365d/y.

```
dseconds(15)
#> [1] "15s"
dminutes(10)
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
#> [1] "0s"                "86400s (~1 days)"  "172800s (~2 days)"
#> [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
#> [1] "1814400s (~3 weeks)"
dyears(1)
#> [1] "31557600s (~1 years)"
```

Durations arithmetics

- Add and multiply durations:

```
2 * dyears(1)
#> [1] "63115200s (~2 years)"
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38869200s (~1.23 years)"
```

- Add and subtract durations to and from dates/datetimes:

```
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)
```

- What happens here?

```
one_pm <- ymd_hms("2016-03-12 13:00:00", tz = "America/New_York")
one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
```

Periods

- Work with "human" times, like days (no fixed length in secs):

```
one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
seconds(15)
#> [1] "15S"
minutes(10)
#> [1] "10M 0S"
hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
days(7)
#> [1] "7d 0H 0M 0S"
months(1:3)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S"
weeks(3)
#> [1] "21d 0H 0M 0S"
years(1)
#> [1] "1y 0m 0d 0H 0M 0S"
```

Periods arithmetics

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- Add and multiply periods:

```
10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"
```

- Add periods to dates/datetimes:

```
# A leap year
ymd("2016-01-01") + dyears(1)
#> [1] "2016-12-31 06:00:00 UTC"
ymd("2016-01-01") + years(1)
#> [1] "2017-01-01"

# Daylight Savings Time
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

Intervals

- What should the following code return?

```
years(1) / days(1)
```

- A duration with a starting point:

```
next_year <- today() + years(1)
(today() %--% next_year) / ddays(1)
#> [1] 365
```

Summary

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

	date	date time	duration	period	interval	number
date	-		- +	- +		- +
date time		-	- +	- +		- +
duration	- +	- +	- + ∕			- + × ∕
period	- +	- +		- +		- + × ∕
interval			∕	∕		
number	- +	- +	- + ×	- + ×	- + ×	- + × ∕

- Pick the simplest data structure that solves your problem:
  - If you only care about physical time, use a duration.
  - If you need to add human times, use a period.
  - If you need to figure out how long a span is in human units, use an interval.

Time zones

```
Sys.timezone()
#> [1] "America/New_York"
length(OlsonNames())
#> [1] 608
head(OlsonNames())
#> [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
#> [4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
```

Same instant in different time zones

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- Same instant, different place:

```
(x1 <- ymd_hms("2015-06-01 12:00:00", tz = "America/New_York"))
#> [1] "2015-06-01 12:00:00 EDT"
(x2 <- ymd_hms("2015-06-01 18:00:00", tz = "Europe/Copenhagen"))
#> [1] "2015-06-01 18:00:00 CEST"
(x3 <- ymd_hms("2015-06-02 04:00:00", tz = "Pacific/Auckland"))
#> [1] "2015-06-02 04:00:00 NZST"
x1 - x2
#> Time difference of 0 secs
x1 - x3
#> Time difference of 0 secs
```

- Note the behavior of ‘c():

```
x4 <- c(x1, x2, x3)
x4
#> [1] "2015-06-01 12:00:00 EDT" "2015-06-01 12:00:00 EDT"
#> [3] "2015-06-01 12:00:00 EDT"
```

Changing the time zone

- Keep the instant in time:

```
x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
x4a
#> [1] "2015-06-02 02:30:00 +1030" "2015-06-02 02:30:00 +1030"
#> [3] "2015-06-02 02:30:00 +1030"
x4a - x4
#> Time differences in secs
#> [1] 0 0 0
```

- Change the instant in time:

```
x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
x4b
#> [1] "2015-06-01 12:00:00 +1030" "2015-06-01 12:00:00 +1030"
#> [3] "2015-06-01 12:00:00 +1030"
x4b - x4
#> Time differences in hours
#> [1] -14.5 -14.5 -14.5
```

Outline

Factors

- Factors are:
  - Used to work with categorical variables (i.e., that have a fixed and known set of possible values).
  - Useful to display character vectors in a non-alphabetical order.
- The **forcats** package:
  - Range of helpers for working with factors.

```
library(forcats)
```

Creating factors

Columbia University in the City of New York

- Imagine that you have a variable that records month:

```
x1 <- c("Dec", "Apr", "Jan", "Mar")
```

- Using a string to record this variable has two problems:
  - Twelve possible months and nothing saving you from typos.
  - It doesn't sort in a useful way.

```
sort(x1)
#> [1] "Apr" "Dec" "Jan" "Mar"
```

Creating factors II

- Start by creating a list of the valid **levels**:

```
month_levels <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
                  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
```

- Then create a factor:

```
y1 <- factor(x1, levels = month_levels)
y1
#> [1] Dec Apr Jan Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
sort(y1)
#> [1] Jan Mar Apr Dec
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
factor(x1) ## without levels
#> [1] Dec Apr Jan Mar
#> Levels: Apr Dec Jan Mar
```

Creating factors III

- Notice:

```
x2 <- c("Dec", "Apr", "Jam", "Mar")
y2 <- factor(x2, levels = month_levels)
y2
#> [1] Dec  Apr  <NA> Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

- Other ordering:

```
factor(x1, levels = unique(x1))
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar
factor(x1) %>%
  fct_inorder()
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar
```

forcats::gss_cat

- Sample from the General Social Survey:

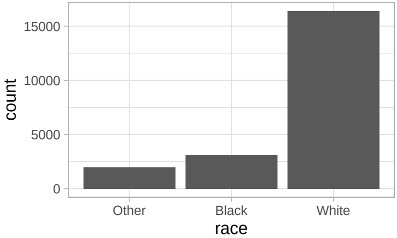
```
gss_cat
#> # A tibble: 21,483 x 9
#>     year marital     age race  rincome   partyid  relig  denom  tvhours
#>    <int> <fct>     <int> <fct> <fct>     <fct>    <fct>  <fct>    <int>
#>  1  2000 Never m~     26 White $8000 t~ Ind,nea~ Prote~ South~      12
#>  2  2000 Divorced     48 White $8000 t~ Not str~ Prote~ Bapti~      NA
#>  3  2000 Widowed      67 White Not app~ Indepen~ Prote~ No de~       2
#>  4  2000 Never m~     39 White Not app~ Ind,nea~ Ortho~ Not a~       4
#>  5  2000 Divorced     25 White Not app~ Not str~ None   Not a~       1
#>  6  2000 Married      25 White $20000 ~ Strong ~ Prote~ South~      NA
#>  7  2000 Never m~     36 White $25000 ~ Not str~ Chris~ Not a~       3
#>  8  2000 Divorced     44 White $7000 t~ Ind,nea~ Prote~ Luthe~      NA
#>  9  2000 Married      44 White $25000 ~ Not str~ Prote~ Other        0
#> 10  2000 Married      47 White $25000 ~ Strong ~ Prote~ South~       3
#> # ... with 21,473 more rows
```

- More info with ?gss_cat.

See levels of a factor from a tibble

- A barplot:

```
ggplot(gss_cat, aes(race)) +
  geom_bar()
```



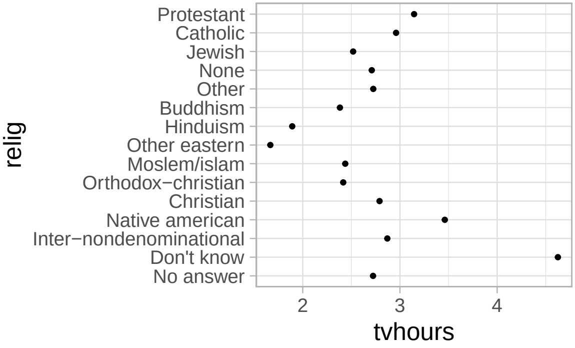
- Or a count:

```
gss_cat %>%
  count(race)
#> # A tibble: 3 x 2
#>   race      n
#>   <fct> <int>
#> 1 Other  1959
#> 2 Black  3129
#> 3 White 16395
```

What's wrong here?

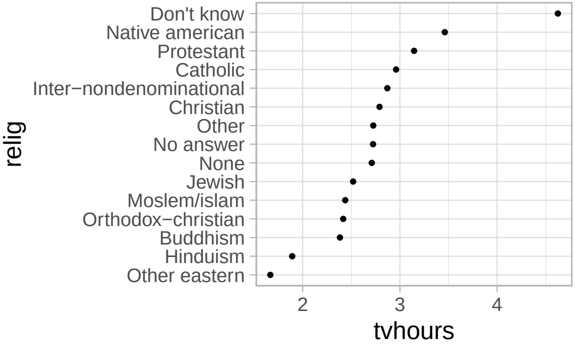

```
relig_summary <- gss_cat %>%
  group_by(relig) %>%
  summarize(age = mean(age, na.rm = TRUE),
            tvhours = mean(tvhours, na.rm = TRUE),
            n = n())

ggplot(relig_summary, aes(tvhours, relig)) +
  geom_point()
```



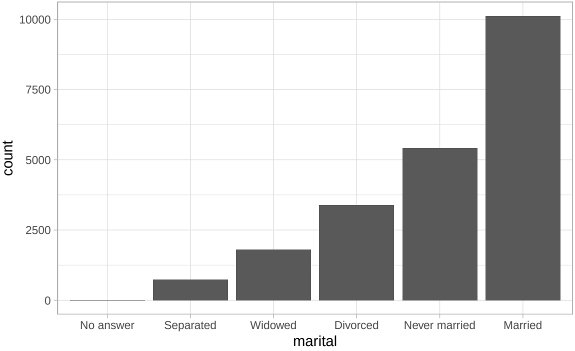
Modifying factor order


```
relig_summary %>%
  mutate(relig = fct_reorder(relig, tvhours)) %>%
  ggplot(aes(tvhours, relig)) +
  geom_point()
```



Modify factor order II

```
gss_cat %>%
  mutate(marital = marital %>% fct_infreq() %>% fct_rev()) %>%
  ggplot(aes(marital)) +
  geom_bar()
```



Modifying factor levels

- More powerful than changing the orders of the levels is changing their values:
  - To clarify labels for publication.
  - To collapse levels for high-level displays.
- What's wrong here?

```
gss_cat %>%
  count(partyid)
#> # A tibble: 10 x 2
#>    partyid                n
#>    <fct>              <int>
#>  1 No answer            154
#>  2 Don't know             1
#>  3 Other party          393
#>  4 Strong republican   2314
#>  5 Not str republican  3032
#>  6 Ind,near rep        1791
#>  7 Independent         4119
#>  8 Ind,near dem        2499
#>  9 Not str democrat    3690
#> 10 Strong democrat     3490
```

Modifying factor levels II


```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat")) %>%
  count(partyid)
#> # A tibble: 10 x 2
#>    partyid                   n
#>    <fct>                 <int>
#>  1 No answer               154
#>  2 Don't know                1
#>  3 Other party             393
#>  4 Republican, strong     2314
#>  5 Republican, weak       3032
#>  6 Independent, near rep  1791
#>  7 Independent            4119
#>  8 Independent, near dem  2499
#>  9 Democrat, weak         3690
#> 10 Democrat, strong       3490
```

Collapsing factors

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat",
    "Other"                 = "No answer",
    "Other"                 = "Don't know",
    "Other"                 = "Other party" )) %>%
  count(partyid)
#> # A tibble: 8 x 2
#>   partyid                   n
#>   <fct>                 <int>
#> 1 Other                   548
#> 2 Republican, strong     2314
#> 3 Republican, weak       3032
#> 4 Independent, near rep  1791
#> 5 Independent            4119
#> 6 Independent, near dem  2499
#> 7 Democrat, weak         3690
#> 8 Democrat, strong       3490
```

Collapsing factors II


```
gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
    dem = c("Not str democrat", "Strong democrat")
  )) %>%
  count(partyid)
#> # A tibble: 4 x 2
#>   partyid     n
#>   <fct>   <int>
#> 1 other     548
#> 2 rep      5346
#> 3 ind      8409
#> 4 dem      7180
```

Collapsing factor III

Columbia University
in the City of New York

```
gss_cat %>%
  mutate(relig = fct_lump(relig)) %>%
  count(relig)
#> # A tibble: 2 x 2
#>   relig          n
#>   <fct>      <int>
#> 1 Protestant 10846
#> 2 Other      10637

gss_cat %>%
  mutate(relig = fct_lump(relig, n = 3)) %>%
  count(relig, sort = TRUE)
#> # A tibble: 4 x 2
#>   relig          n
#>   <fct>      <int>
#> 1 Protestant 10846
#> 2 Catholic    5124
#> 3 None        3523
#> 4 Other       1990
```

Outline

String basics

Columbia University in the City of New York

```
library(stringr) # package for string manipulation

# To create strings
string1 <- "This is a string"
string2 <- 'To get a "quote" inside a string, use single quotes'
```

- Backslash as escape character:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

- The printed representation is not the string itself:

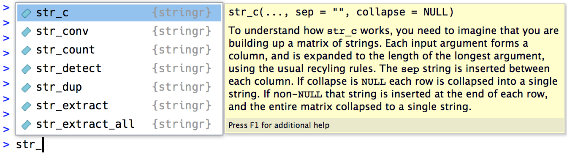
```
x <- c("\"", "\\")
x
#> [1] "\"" "\\"
writeLines(x)
#> "
#> \
```

More on strings

- Special characters:
  - Use "\n", for newline, or,"\t", for tab.
  - Complete list by requesting help on " (?'"', or ?"'")
- Other usefuls things:

```
(x <- "\u00b5") # Non-English characters
#> [1] "µ"
c("one", "two", "three") # Character vectors
#> [1] "one"   "two"   "three"
str_length(c("a", "R for data science", NA)) # String length
#> [1]  1 18 NA
```

- stringr autocomplete:



More on strings II

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- Combining strings:

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

- Missing values:

```
x <- c("abc", NA)
str_c("|-", x, "-|")
#> [1] "|-abc-|" NA
str_c("|-", str_replace_na(x), "-|")
#> [1] "|-abc-|" "|-NA-|"
```

- Recycling:

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
#> [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

- Collapsing a vector of strings:

```
str_c(c("x", "y", "z"), collapse = ", ")
#> [1] "x, y, z"
```

Subsetting strings


```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
#> [1] "App" "Ban" "Pea"
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
str_sub("a", 1, 5)
#> [1] "a"
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "apple"  "banana" "pear"
```

- See also `str_to_upper()` or `str_to_title()`.

Locales


```
# Turkish has two i's: with and without a dot, and it
# has a different rule for capitalising them:
str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

- The locale:
  - An ISO 639 language code, which is a two or three letter abbreviation
  - If blank, R uses the current locale, as provided by your operating system.

Regular expressions

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now, they have two problems. ——— Jamie Zawinski

- A language that allows you to describe patterns in strings.
- Allows you for instance to:
  - Determine which strings match a pattern.
  - Find the positions of matches.
  - Extract the content of matches.
  - Replace matches with new values.
  - Split a string based on a match.
- Read the chapter on strings from the book!

Basic matches

- The simplest patterns match exact strings:

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
```

```
apple
banana
pear
```

- Next step is ., which matches any character (except a newline):

```
str_view(x, ".a.")
```

```
apple
banana
pear
```

- If "." matches any character, how to match the character "."?

Basic matches II

- If "." matches any character, how to match the character "."?
  - Need to use an "escape" (like string, a backslash \).
  - So to match an ., need the regexp `\.`.
  - But \ is also an escape symbol in strings.
  - So to create the regexp `\.`, use the string `"\\."`.

```
# To create the regexp, we need \\
dot <- "\\."
# But the expression itself only contains one:
writeLines(dot)
#> \.
# And this tells R to look for an explicit .
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

abc

a.c

bef

Basic matches III

- If \ is an escape character, how do you match a literal \?
  - Need to escape it, i.e. create the regexp \\.
  - To create that regexp with a string, which also needs to escape \, need to write "\\\\"
  - I.e., need four backslashes to match one!

```
x <- "a\\b"
writeLines(x)
#> a\b
str_view(x, "\\\\")
```

a\b

Anchors

- By default, regexps match any part of a string.
- Often useful to *anchor* the regexp:
  - ^ to match the start of the string.
  - $ to match the end of the string.

```
x <- c("apple", "banana", "pear")
```

```
str_view(x, "^a")
```

apple

banana

pear

```
str_view(x, "a$")
```

apple

banana

pear

- To remember, Evan Misshula's mnemonic: if you begin with power (^), you end up with money ($).

Anchors II

Columbia University in the City of New York

- To force a regexp to only match a complete string, anchor it with both ^ and $:

```
x <- c("apple pie", "apple", "apple cake")
```

```
str_view(x, "apple")
```

apple pie
apple
apple cake

```
str_view(x, "^apple$")
```

apple pie
apple
apple cake

Character classes and alternatives

- Some special patterns match more than one character:
  - Already seen `.` (matches any character apart from a newline).
  - Two other useful tools:
    - `\d`: matches any digit.
    - `\s`: matches any whitespace (e.g. space, tab, newline).
  - To create a regexp containing `\d` or `\s`:
    - Need to escape the \ for the string.
    - So type `"\\d"` or `"\\s"`.
- The other two tools are:
  - **Character classes**
    - `[abc]`: matches a, b, or c.
    - `[^abc]`: matches anything except a, b, or c.
  - **Alternatives**
    - `abc|d..f`: matches either `"abc"`, or `"deaf"`.

Character classes

- Can be used as an alternative to backslash escapes.

```
str_view(c("abc",
           "a.c",
           "a*c",
           "a c"),
         "a[.]c")
```

abc

a.c

a*c

a c

```
str_view(c("abc",
           "a.c",
           "a*c",
           "a c"),
         ".[*]c")
```

abc

a.c

a*c

a c

```
str_view(c("abc",
           "a.c",
           "a*c",
           "a c"),
         "a[ ]")
```

abc

a.c

a*c

a c

- Used to pick between one or more alternative patterns.
- Works for most regex metacharacters: $. | ? * + ( ) [{.
- But some have special meaning even inside a character class.
  - Must be handled with backslash escapes:] \ ^ and -.

Alternatives

- Note that the precedence for | is low:
  - `abc|xyz`: matches `abc` or `xyz`, not `abcyz` or `abxyz`.
- Same as mathematical expressions: if it gets confusing, use parentheses.

```
str_view(c("grey", "gray"), "gr(e|a)y")
```

```
grey
gray
```

Repetition

- To control how many times a pattern matches:
  - `?`: 0 or 1.
  - `+`: 1 or more.
  - `*`: 0 or more.

```
# 1888 is the longest year in Roman numerals
x <- "MDCCCLXXXVIII"
```

```
str_view(x, "CC?")
```

MD[CC]CLXXXVIII

```
str_view(x, "CC+")
```

MD[CCC]LXXXVIII

```
str_view(x, 'C[LX]+')
```

MDCC[CLXXX]VIII

- The precedence of these operators is high:
  - `colou?r`: matches either US or British spellings.
  - Most uses will need parentheses, like `bana(na)+`.

Repetition

- To specify the number of matches precisely:
  - `{n}`: exactly n.
  - `{n,}`: n or more.
  - `{,m}`: at most m.
  - `{n,m}`: between n and m.

```
str_view(x, "C{2}")
```

MDCCCLXXXVIII

```
str_view(x, "C{2,}")
```

MDCCCLXXXVIII

```
str_view(x, "C{2,3}")
```

MDCCCLXXXVIII

Grouping and backreferences

- Earlier: parentheses as a way to disambiguate complex expressions.
- But parentheses also create a *numbered* capturing group.
- A capturing group stores *the part of the string* matched by the part of the regexp inside the parentheses.
- Refer to the same text as previously matched by a capturing group with *backreferences*, like \1, \2 etc.

```
str_view(fruit, "(..)\\1", match = TRUE)
```

```
banana
coconut
cucumber
jujube
papaya
salal berry
```

- Cool applications in chapter 14.4!