Datatypes and I/O

Alejandro Schuler, based on R for Data Science by Hadley Wickham
2019, updated 2021

- create and index vectors of different types
- efficiently manipulate strings, factors, and date-time vectors
- tightly control the intake of tabular data

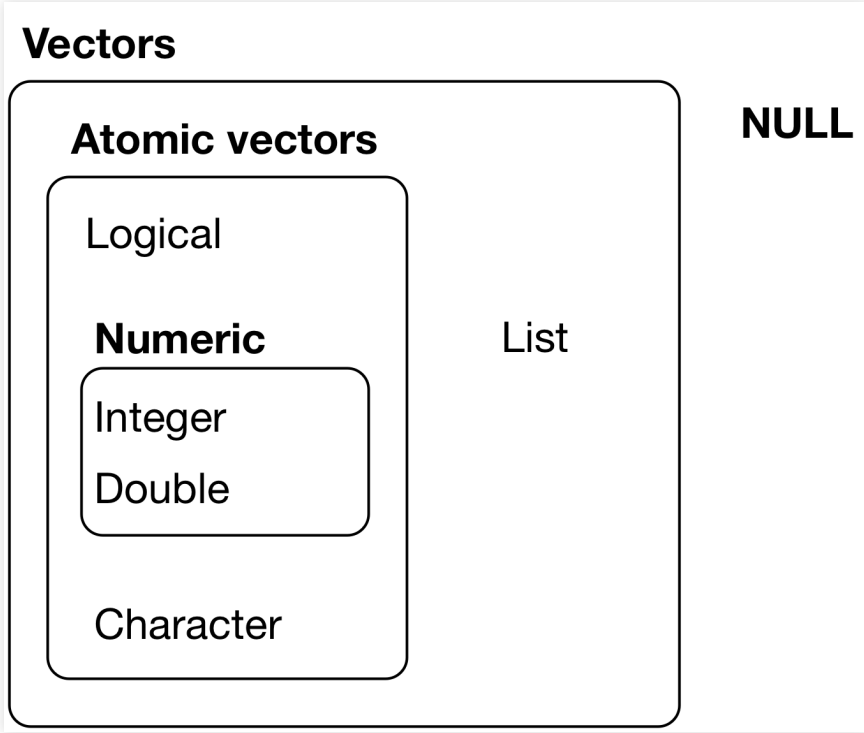
Vectors

Vector basics

“As you dig deeper into R, you need to learn about vectors, the objects that underlie tibbles. If you’ve learned R in a more traditional way, you’re probably already familiar with vectors, as most R resources start with vectors and work their way up to tibbles. I think it’s better to start with tibbles because they’re immediately useful, and then work your way down to the underlying components” - Hadley Wickam

Vector basics

- A vector in R is an ordered sequence of elements
- Each element has a particular **type**, e.g. string, numeric
- If all the elements in the vector are of the same type, then we call it an **atomic vector** or simply a **vector**
- If the elements are of different types, it is called a **list**, which will be discussed later



Why care about vectors?

- We can already manipulate data frames with tidyverse functions

```
orange <- as_tibble(Orange)
orange %>%
  mutate(age_yrs = age/365) %>%
  mutate(approx_age_yr = round(age_yrs)) %>%
  group_by(approx_age_yr) %>%
  summarize(mean_circ = mean(circumference))
# A tibble: 5 x 2
  approx_age_yr mean_circ
          <dbl>     <dbl>
1             0      31
2             1      57.8
3             2      93.2
4             3     140.
5             4     175.
```

- However, each column in a dataframe is actually a **vector** and the functions we use inside `summarize()` and `mutate()` operate on these vectors.
- So if we want to write our own functions to use with data frames, we should understand a little bit about these basic objects

Vector basics

- As we have already seen, vectors can be created with the `c()` function:

```
shoesize <- c(9, 12, 6, 10, 10, 16, 8, 4) # integer vector
people <- c("Vinnie", "Patricia", "Gabriel") # character (string) vector
```

- All vectors have two key properties: **length** and **type**, which you can check as follows:

```
typeof(people)
[1] "character"
typeof(shoesize)
[1] "double"
length(people)
[1] 3
```

Vector basics

- vectors of the same type can be combined with `c()`

```
c(shoesize, c(4, 8, 10, 10))
 [1]  9 12  6 10 10 16  8  4  4  8 10 10
```

- Regular numbers, strings, etc. are actually all treated as vectors of length 1

```
12
[1] 12
```

Logical vectors

```
1:10 %% 3 == 0
 [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
c(TRUE, TRUE, FALSE, NA)
[1]  TRUE  TRUE FALSE    NA
```

- logical vectors can only be one of three values: `TRUE`, `FALSE`, or `NA`.

Basic logical operations

```
TRUE & FALSE
[1] FALSE
TRUE | FALSE
[1] TRUE
!TRUE
[1] FALSE
```

- `&` is logical AND. It is true only if both arguments are true.
- `|` is logical OR. It is true if either argument is true.
- `!` is logical NOT.
- ADVANCED: see `?Logic` for more.

Numeric vectors

```
sqrt(1:10)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278
```

- Numeric vectors can be any number or one of three special values: `Inf` (1/0), `NaN` (0/0), and `NA` (missing)
- R uses scientific notation so `6.023e23` evaluates as `6.023 * 10^23`

Type coercion

- some vector types can be easily converted to other types:

```
as.character(shoesize)
[1] "9"  "12" "6"  "10" "10" "16" "8"  "4"
```

- see `as.character()`, `as.logical()`, `as.numeric()`, etc.
- coercion most often happens implicitly when you use a vector in a context that is expecting a specific type:

```
shoe_gt_8 = shoesize > 8
typeof(shoe_gt_8)
[1] "logical"
sum(shoe_gt_8) # under the hood, sum(as.numeric(shoe_gt_8))
[1] 5
mean(shoe_gt_8) # under the hood, mean(as.numeric(shoe_gt_8))
[1] 0.625
```

Type coercion

- Implicit coercion also happens if you try and combine two vectors of different types

```
(A <- c(TRUE, 1L))
[1] 1 1
typeof(A)
[1] "integer"
(B <- c(1L, 1.5))
[1] 1.0 1.5
typeof(B)
[1] "double"
(C <- c(1.5, "a"))
[1] "1.5" "a"
typeof(C)
[1] "character"
```

Testing type

Use these to see if something is a vector or of the desired type. They all return either TRUE or FALSE

- is_logical()
- is_integer()
- is_double()
- is_character()
- is_atomic()
- is_list()
- is_vector()

```
is_double(0.14) # neat
[1] TRUE
typeof(TRUE) == "logical" # not neat
[1] TRUE
```

- these functions are imported by `tidyverse`. Base R has its own equivalents but they are not well designed and sometimes produce surprising results.

Matrices

- A matrix is a 2D vector (with rows and columns)

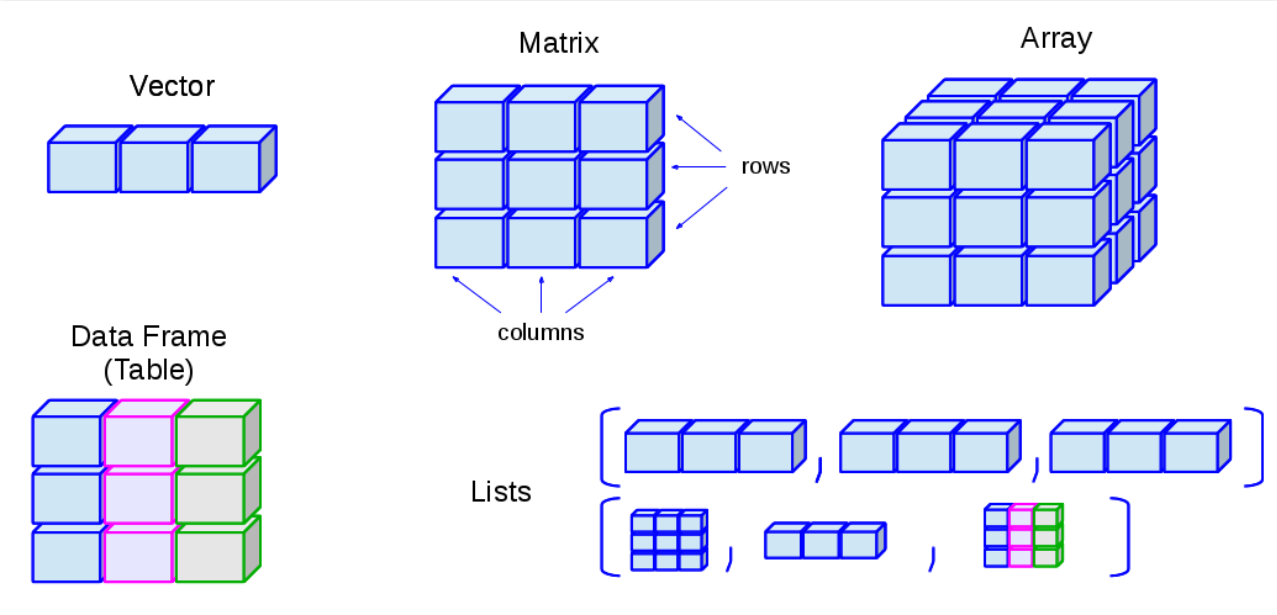
```
my_example_matrix
           [,1]        [,2]        [,3]       [,4]       [,5]
[1,] -0.8147981 -0.12691164  0.15320466  0.2514097  1.8769308
[2,] -0.9599709 -0.15283104 -1.59178959 -0.2188753 -1.5261497
[3,] -0.2265966  0.03906986 -0.03295731  0.7352565  0.3401731
[4,] -1.0282813 -0.64951751  0.83685936  0.7517605  0.3168973
[5,]  0.6658058 -0.68062971  0.43948325 -2.4527679 -0.2359612
```

- All the usual vector rules apply- in particular, all entries of the matrix must be of the same type

```
typeof(my_example_matrix)
[1] "double"
```

- this distinguishes it from a data frame, which can have different types in different columns
- rows and columns can be named (the same way vector entries can be named)
- matrices are far more rare than vectors, but many modeling packages take numeric matrices as input, so it's good to be aware of them
- objects (including data frames) can be coerced to matrices using `as.matrix()`.

Vectors, matrices, data frames



Exercise: coercing a data frame to matrix

```
gtex_data = read_tsv("https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-2021/data/gtex.tissue.zscores.advance2020.txt")  # includes normalized expression data for GTEx individuals across four tissues
head(gtex_data)
# A tibble: 6 x 7
  Gene  Ind         Blood Heart   Lung Liver NTissues
  <chr> <chr>       <dbl> <dbl>  <dbl> <dbl>    <dbl>
1 A2ML1 GTEX-11DXZ -0.14 -1.08 NA     -0.66         3
2 A2ML1 GTEX-11GSP -0.5   0.53  0.76  -0.1          4
3 A2ML1 GTEX-11NUK -0.08 -0.4  -0.26  -0.13         4
4 A2ML1 GTEX-11NV4 -0.37  0.11 -0.42  -0.61         4
5 A2ML1 GTEX-11TT1  0.3  -1.11  0.59  -0.12         4
6 A2ML1 GTEX-11TUW  0.02 -0.47  0.290 -0.66         4
```

Convert this dataset into a matrix and find out what type it is. Does this make sense?

Indexing a vector

- We can get parts of a vector out by indexing it. This is like `filter()`ing a data frame, but it looks a little different with vectors. We use `[ ]` with an index vector inside the brackets

```
x <- c(0.3, 0.1, -5, 12)
```

There are a few ways to subset a vector:

- with a numeric index vector of integers

```
x[c(1,3)]
[1]  0.3 -5.0
```

- with a logical index vector (of the same length)

```
x[c(T,F,T,T)] # T is short for TRUE, F is short for FALSE
[1]  0.3 -5.0 12.0
```

Indexing with integers

```
x
[1]  0.3  0.1 -5.0 12.0
x[1] # same as x[c(1)] since 1 is already a vector (of length 1)
[1] 0.3
x[2:4]
[1]  0.1 -5.0 12.0
x[c(3, 1)]
[1] -5.0  0.3
x[c(1,1,1,1,1,1,4)]
[1]  0.3  0.3  0.3  0.3  0.3  0.3 12.0
```

- Indexing returns a subsequence of the vector. It does not change the original vector. Assign the result to a new variable to save it if you neeed it later.
- R starts counting vector indices from 1.
- You can index using a multi-element index vector.
- You can repeat index positions

Indexing with integers (negatives)

```
x
[1]  0.3  0.1 -5.0 12.0
x[1]
[1] 0.3
x[-1]
[1]  0.1 -5.0 12.0
x[-length(x)]
[1]  0.3  0.1 -5.0
x[c(-1, -length(x))]
[1]  0.1 -5.0
```

- You can't mix positive and negative vector indices in a single index expression. R will complain.
- This is similar to the `df %>% select(-var)` syntax
- What about using `0` as an index? It is ignored.

Indexing with logicals

```
x
[1]  0.3  0.1 -5.0 12.0
x >= 0
[1]  TRUE  TRUE FALSE  TRUE
x[x >= 0]
[1]  0.3  0.1 12.0
```

- Logical values are either `TRUE` or `FALSE`.
- They are typically produced by using a comparison operator or similar test.
- The logical index vector should be the same length as the vector being subsetted

Indexing 2D objects

Similar syntax is used for 2D entities

```
my_example_matrix[1:3, c(2,2)]
            [,1]        [,2]
[1,] -0.12691164 -0.12691164
[2,] -0.15283104 -0.15283104
[3,]  0.03906986  0.03906986
```

- the general pattern is `matrix[row_index, column_index]`.
- leaving either blank returns all rows or columns

```
my_example_matrix[1:3,]
            [,1]        [,2]        [,3]       [,4]       [,5]
[1,] -0.8147981 -0.12691164  0.15320466  0.2514097  1.8769308
[2,] -0.9599709 -0.15283104 -1.59178959 -0.2188753 -1.5261497
[3,] -0.2265966  0.03906986 -0.03295731  0.7352565  0.3401731
my_example_matrix[,c(T,T,F,F,T)]
            [,1]        [,2]       [,3]
[1,] -0.8147981 -0.12691164  1.8769308
[2,] -0.9599709 -0.15283104 -1.5261497
[3,] -0.2265966  0.03906986  0.3401731
[4,] -1.0282813 -0.64951751  0.3168973
[5,]  0.6658058 -0.68062971 -0.2359612
```

Indexing for data frames

- Data frames can be indexed like matrices:

```
gtex_data[1:3, c(2,3)]
# A tibble: 3 x 2
  Ind         Blood
  <chr>       <dbl>
1 GTEX-11DXZ -0.14
2 GTEX-11GSP -0.5
3 GTEX-11NUK -0.08
```

- Or using column names instead of column indices:

```
gtex_data[1:3, c("Ind","Blood")]
# A tibble: 3 x 2
  Ind         Blood
  <chr>       <dbl>
1 GTEX-11DXZ -0.14
2 GTEX-11GSP -0.5
3 GTEX-11NUK -0.08
```

- The `$` operator provides a shortcut to access a single column as a vector:

```
gtex_data$Ind[1:3] # gtex_data %>% pull(Ind)
[1] "GTEX-11DXZ" "GTEX-11GSP" "GTEX-11NUK"
```

Comparing tidyverse vs. vector indexing

```
df <- tibble(x=x, y=rnorm(4), z=rnorm(4))
```

Tidyverse

```
df %>%
  filter(x>0) %>%
  select(x,y)
# A tibble: 3 x 2
      x       y
  <dbl>   <dbl>
1   0.3 -0.139
2   0.1  0.0487
3  12    0.709
```

Vector indexing

```
df[df$x>0, c(1,2)] # df$x takes the column x from the data frame df (details
later)
# A tibble: 3 x 2
      x       y
  <dbl>   <dbl>
1   0.3 -0.139
2   0.1  0.0487
3  12    0.709
```

- What are the advantages/disadvantages of each?

Tidyverse vs. vector indexing

- **Tidyverse**:
  - operations are ordered top-to-bottom/left-to-right in the way that they are being performed so the code can be read like natural language
  - each operation gets its own line, which facilitates finding bugs
  - functions are named sensibly so the code can be understood without knowledge of symbols like `$` or `[`. Even `%>%` is made to look like an arrow to suggest the left-to-right flow
  - variables are always referred to as bare names (no quotes or `$`)
  - will be easier for you to skim over and understand in 6 months
- **Vector indexing**
  - Fewer keystrokes
  - More familiar to programmers from C++ or python

Tidyverse

```
df %>%
  filter(x>0) %>%
  select(x,y)
```

Vector indexing

```
df[df$x>0, c(1,2)]
```

Exercise: translate to tidyverse:

Rewrite the following code using the tidyverse:

```
all_tissues = gtex_data[gtex_data$NTissues==4,]
all_tissues$lung_blood_diff = all_tissues$Lung - all_tissues$Blood
ordering = sort(all_tissues$lung_blood_diff, index.return=T)$ix
all_tissues[ordering,][1:10, c("Gene", "Ind", "Lung", "Blood", "lung_blood_diff")]
# A tibble: 10 x 5
   Gene          Ind          Lung Blood lung_blood_diff
   <chr>         <chr>       <dbl> <dbl>           <dbl>
 1 KLK3          GTEX-147F4 -0.44  15.7            -16.2
 2 DNASE2B       GTEX-12696 -0.92  14.4            -15.4
 3 GAPDHP33      GTEX-UPK5  -1.48  13.8            -15.3
 4 FFAR4         GTEX-12696 -0.67  12.9            -13.6
 5 AC093901.1    GTEX-1AX9I -3.5    9.39           -12.9
 6 KCNT1         GTEX-1KANB  0.62  13.5            -12.8
 7 NAPSA         GTEX-1CB4J -0.44  12.3            -12.8
 8 C3orf70       GTEX-14E1K -2.31  10.4            -12.8
 9 AC012358.7    GTEX-VUSG   1.22  13.6            -12.4
10 RP11-739P1.3  GTEX-VUSG  -0.97  11.2            -12.1
```

If you are unsure about what any line is doing:

- run it and compare the inputs to the outputs
- read the documentation for the relevant functions
- generate your own toy input and test the code with it to experiment

Working with strings

String basics

```
c("1234", "sum(c(1,2,3,4))", "Alejandro", "a long string with weird
characters!@#$!%>?", "NA", NA)
[1] "1234"
[2] "sum(c(1,2,3,4))"
[3] "Alejandro"
[4] "a long string with weird characters!@#$!%>?"
[5] "NA"
[6] NA
```

- character vectors (or string vectors) store strings, which are arbitrary text (including spaces) or `NA`
- even if the text can be interpreted by you as code or numbers, quoting it in `""` tells `R` that it is to be taken literally as text (note `"NA"` is a string, while `NA` is a special value that indicates a missing string if it is in a string vector)

stringr

- Many of the functions we will use to work with strings come from the `stringr` package, which is `tidyverse`-associated and by default loaded with tidyverse.

String basics

- Strings are created by quoting text with `" "` or `' '`. I always use `" "` to be consistent.

```
string1 <- "This is a string"
```

- special characters (see `?"'"`) can be included by "escaping" them with `\`. `\\` is a literal backslash.

```
string2_wrong = "This is a "string""
Error: <text>:1:29: unexpected symbol
1: string2_wrong = "This is a "string
                               ^
```

```
string2_right = "This is a \"string\""
```

- check the length of a string with `str_length()` (why not `length()`?)

```
x = "This is a string"
str_length(x)
[1] 16
length(x)
[1] 1
```

Combining strings

- Use `str_c()` to combine strings

```
str_c("x", "y", "z")
[1] "xyz"
```

- `sep` controls what gets stuck between them (`""` by default)

```
str_c("x", "y", "z", sep = ", ")
[1] "x, y, z"
```

- also works with vectors of strings

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
[1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
str_c(c("1", "2", "3"), c("a", "b", "c"), sep="-")
[1] "1-a" "2-b" "3-c"
```

- and with a single vector, if you set the `collapse` argument

```
name = "Alejandro"
x = c("Good afternoon,", name, "how are you?")
str_c(x, collapse=" ")
[1] "Good afternoon, Alejandro how are you?"
```

Subsetting

- str_sub() does subsetting by start and end letters
- these can be negative numbers to count from the end of the string backwards

```
str_sub("Hello world", start=1, end=5)
[1] "Hello"
str_sub("Hello world", start=-5, end=-1)
[1] "world"
```

- Also works on vectors

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
[1] "App" "Ban" "Pea"
```

Searching in strings

- `str_detect()` tells you if the query string is in the string (or vector of strings) you're looking at

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
[1]  TRUE FALSE  TRUE
```

- `str_subset()` returns the strings in the vector that match the query

```
str_subset(x, "e") # compare to x[str_detect(x,"e")]
[1] "apple" "pear"
```

Exercise: find rows according to string match

```
drugs <- read_csv("https://raw.githubusercontent.com/alejandroschuler/r4ds-
courses/advance-2021/data/drugSurvey.csv")
head(drugs)
# A tibble: 6 x 7
  uniqueID drugName   condition     review                rating date  usefulCount
     <dbl> <chr>      <chr>         <chr>                  <dbl> <chr>       <dbl>
1   163740 Mirtazapi… Depression    "\"I&#039;ve tried…     10 28-F…          22
2   206473 Mesalamine Crohn's Dise… "\"My son has Croh…      8 17-M…          17
3   159672 Bactrim    Urinary Trac… "\"Quick reduction…      9 29-S…           3
4    39293 Contrave   Weight Loss   "\"Contrave combin…      9 5-Ma…          35
5    97768 Cyclafem … Birth Control "\"I have been on …      9 22-O…           4
6   208087 Zyclara    Keratosis     "\"4 days in on fi…      4 3-Ju…          13
```

- Dataset from polling patients that have been taking all sorts of prescription drugs
- Contains columns for: `drugName`, `condition`, `rating`, `date` and `usefulCount` (how many people found the review useful)
- As well as a very messy column, `review` that contains the patient's free-text review of the drug
- Write code that returns all rows where someone's review mentions `"vomiting"`

Counting the number of matches

- `str_count()` counts how many times the query appears in the string

```
x <- c("apple", "banana", "pear")
str_count(x, "a")
[1] 1 3 1
```

- it pairs naturally with `mutate()`

```
drugs %>%
  mutate(n_vomiting = str_count(review, "vomiting")) %>%
  arrange(desc(n_vomiting)) %>%
  select(drugName, condition, n_vomiting) %>%
  head()
# A tibble: 6 x 3
  drugName                        condition                    n_vomiting
  <chr>                           <chr>                             <int>
1 Trintellix                      Depression                            5
2 Ethinyl estradiol / etonogestrel Birth Control                        2
3 Ondansetron                     Nausea/Vomiting                       2
4 Sertraline                      Obsessive Compulsive Disorde          2
5 Oseltamivir                     Influenza                             2
6 Moxifloxacin                    Pneumonia                             2
```

Replacing parts of a string

- sometimes you want to remove something from a string or replace it with something else.

```
drugs$date %>% head()
[1] "28-Feb-12" "17-May-09" "29-Sep-17" "5-Mar-17"  "22-Oct-15" "3-Jul-14"
```

- `str_replace_all()` lets you do this

```
str_replace_all(drugs$date, "-", "/") %>% head()
[1] "28/Feb/12" "17/May/09" "29/Sep/17" "5/Mar/17"  "22/Oct/15" "3/Jul/14"
```

- use `""` as the replacement string to delete the part of the string that matches

```
str_replace_all(drugs$date, "-", "") %>% head()
[1] "28Feb12" "17May09" "29Sep17" "5Mar17"  "22Oct15" "3Jul14"
```

Splitting up a string

- `str_split()` splits a string into multiple strings

```
str_split(drugs$date[1], "-")
[[1]]
[1] "28"  "Feb" "12"
```

- it returns a list, which is like a vector that can contain other vectors or arbitrary length as elements (more on this later). This allows it to operate on multiple strings at once without mashing the results together

```
str_split(drugs$date, "-") %>% head()
[[1]]
[1] "28"  "Feb" "12"

[[2]]
[1] "17"  "May" "09"

[[3]]
[1] "29"  "Sep" "17"

[[4]]
[1] "5"   "Mar" "17"

[[5]]
[1] "22"  "Oct" "15"

[[6]]
[1] "3"   "Jul" "14"
```

Matching complicated patterns: regular expressions

- sometimes you want to match a pattern that is more complicated than a fixed string. For instance, how would you find all strings that have a vowel in them? Or match an email address?
- **regular expressions** (regexps) are a concise way of solving this problem, but they aren't pretty
- don't try and memorize all this, just get an idea of what's possible and then look at the cheat sheet later
- a regular expression is a string that is interpreted in a particular way as a query
- all of the `str_` functions we've talked about take regular expressions as queries.

```
x <- c("apple", "banana", "pear")
str_subset(x, "an")
[1] "banana"
```

- The `.` matches any character

```
str_subset(x, ".a.")
[1] "banana" "pear"
```

Repetition

```
babs = c("bb", "bab", "baab", "baaab")
```

- + after a character means match that character one or more times

```
str_subset(babs, "ba+b")
[1] "bab"   "baab"  "baaab"
```

- * after a character means match that character zero or more times

```
str_subset(babs, "ba*b")
[1] "bb"    "bab"   "baab"  "baaab"
```

- ? after a character means match that character zero or one time

```
str_subset(babs, "ba?b")
[1] "bb"  "bab"
```

- {n,m} after a character means match between n and m repetitions

```
str_subset(babs, "ba{2,3}b")
[1] "baab"  "baaab"
```

Anchors

- anchors tell the regexp to look at particular places in the string
- `^` matches the beginning
- `$` matches the end

```
x <- c("apple", "banana", "pear")
str_subset(x, "^a")
[1] "apple"
str_subset(x, "a$")
[1] "banana"
```

Character classes

- Besides `.`, here are some other special sequences that match categories of characters
- `\\d`: matches any digit.
- `\\s`: matches any whitespace (e.g. space, tab, newline).
- `[abc]`: matches a, b, or c.
- `[a-zA-Z]`: matches any letter
- `[^abc]`: matches anything except a, b, or c.

```
str_subset(c("abc", "xyz"), "[aeiou]")
[1] "abc"
```

Exercise: what does this regex match?

`"^[a-zA-Z]+\\d+[a-zA-Z]+$"`

- If you think you know, come up with examples or counter-examples and test them out with `str_detect()`

Escaping special characters

- `\\` is used to escape special characters in regexps so that they can be interpreted literally

```
str_extract(c("abc", "a.c", "bef"), "a\\.c")
[1] NA    "a.c" NA
```

- It takes two `\\` because a regex is a string. Thus the first `\` puts a literal `\` in the string, which is then interpreted as an escape character by the regex engine.

Extracting matches

- `str_extract()` will get the portion of the string that matches your regex

```
emails = c("karina@stanford.edu", "nathalie@gmail.com", "carlos@kp.org")
str_extract(emails, "@.*")
[1] "@stanford.edu" "@gmail.com"    "@kp.org"
```

Look-arounds

- Sometimes you want to check if a string matches a pattern and then only return part of that pattern.
- For example, if you're looking for history of disease, you may want the disease, but not the term “history of”
- `unique()`: returns all unique values from vector

```
unique(str_extract(drugs$review, "(?<=history of )[a-zA-Z]+(?= )"))
 [1] NA                "health"          "narcotic"        "anxiety"
 [5] "depression"      "motion"          "an"              "diabetes"
 [9] "major"           "pre"             "heart"           "cluster"
[13] "social"          "blood"           "pulmonary"       "early"
[17] "frequent"        "drug"            "problems"        "my"
[21] "cramps"          "cardiac"         "antidepressants" "Heart"
[25] "bipolar"         "stomach"
```

- `(?=...)`: positive look-ahead assertion. Matches if … matches at the current input.
- `(?!...)`: negative look-ahead assertion. Matches if … does not match at the current input.
- `(?<=...)`: positive look-behind assertion. Matches if … matches text preceding the current position, with the last character of the match being the character just before the current position.
- `(?<!...)`: negative look-behind assertion. Matches if … does not match text preceding the current position.

Regex is a broad topic

- There is a lot more to learn. This is a regex that matches valid email addresses:

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-
\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-
\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4]
[0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-
\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

- But you can go very far with very little!
- Regex is not just in R, it is used across almost every programming language

stringr cheat sheet

Factors

Factor basics

- factors are R's representation of variables that can take values in a limited number of categoires
- the `tidyverse`-adjacent `forcats` (forcats = for-categoricals, also an anagram of factor!) package has useful functions for working with them. Like stringr, it is loaded with tidyverse by deffault.
- consider a variable that stores a month of the year:

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

- if you sort this vector, it sorts alphabetically, not by the order of the months
- you can accidentally add months that aren't legitimate: `c(months_vec, "Jam")`
- you can make a factor by hand with `factor()`

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

Factor levels

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

- the “levels” of the factor are the values that it is allowed to take as well as the order that it should be sorted in if desired
- `levels()` returns a character vector of the factor levels
- if the levels are not supplied, it takes the values in the vector as the levels in alphabetical order

```
factor(months_vec)
[1] Dec Apr Jan Mar
Levels: Apr Dec Jan Mar
```

- you can set the order of the levels to be by the order in which they appear in the vector using `fct_inorder()`

```
factor(months_vec) %>%
  fct_inorder()
[1] Dec Apr Jan Mar
Levels: Dec Apr Jan Mar
```

Example: Human Phenotype Ontology (HPO) dataset

- This is an example data frame from the HPO resource (https://hpo.jax.org/app/) which uses structured IDs to describe patient phenotypes and map to associated genes. Also included are broad disease categories assigned to each gene based on several curated disease lists.
- Using the `col_types` argument, we tell R to read in several columns as factors, which are identifiable by the `<fct>` tag
- This data frame contains information about several structured phenotypes (HPO_ID) which have associated descriptions (HPO_Name), related genes (Gene_Name), and broad disease categories associated with that gene (Gene_Disease)

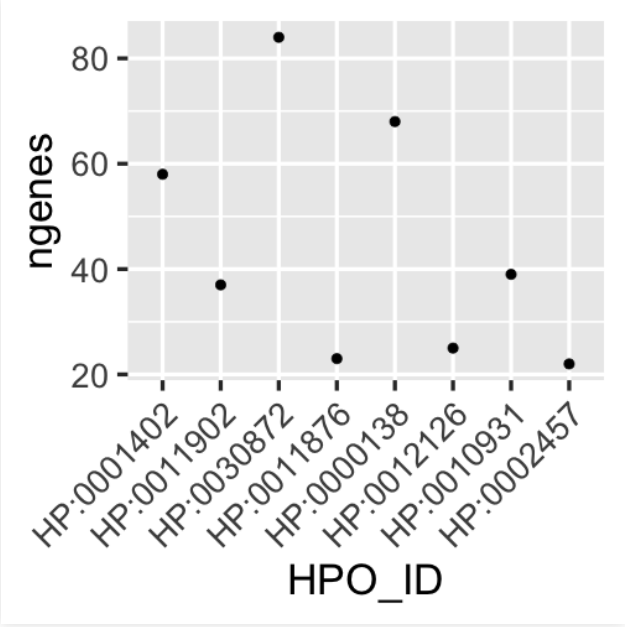
```
hpo_data = read_tsv('https://raw.githubusercontent.com/alejandroschuler/r4ds-
courses/advance-2021/data/HPO_examples_gene_phenotype.txt', col_types='cfff')
head(hpo_data,2)
# A tibble: 2 x 4
  Gene_Name HPO_ID     HPO_Name                 Gene_Disease
  <chr>     <fct>      <fct>                    <fct>
1 ABCB11    HP:0001402 Hepatocellular carcinoma None
2 ABCC8     HP:0011902 Abnormal hemoglobin      None
```

```
hpo_data %>%
  pull(HPO_Name) %>%
  levels()
[1] "Hepatocellular carcinoma"
[2] "Abnormal hemoglobin"
[3] "Abnormal cardiac ventricular function"
[4] "Abnormal platelet volume"
[5] "Ovarian cyst"
```

Ordering factor levels

- It's often useful to change the order of the factor levels in a visualization
- For example, imagine you want to explore the number of genes associated with each phenotype

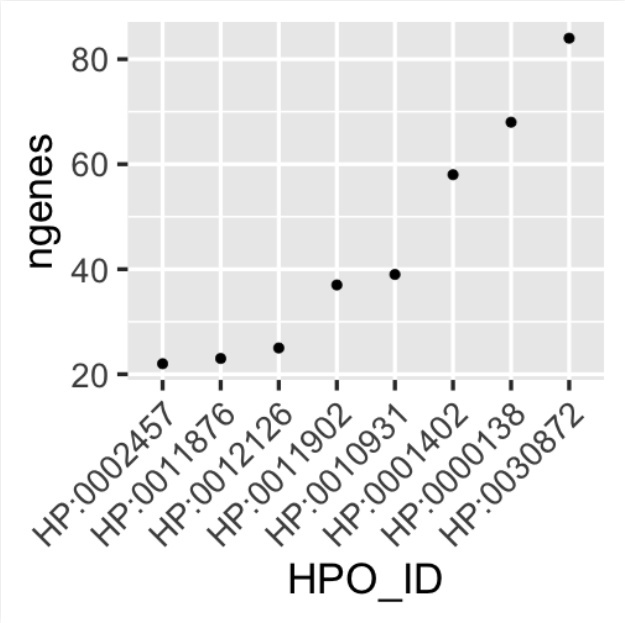
```
num_genes = hpo_data %>%
  group_by(HPO_ID) %>%
  summarise(ngenes = n())
ggplot(num_genes, aes(HPO_ID, ngenes)) +
  geom_point() +
  theme(axis.text.x=element_text(hjust=1,angle=45))
```



Ordering factor levels

- It would be better if the HPO IDs in this plot were ordered according to the number of associated genes

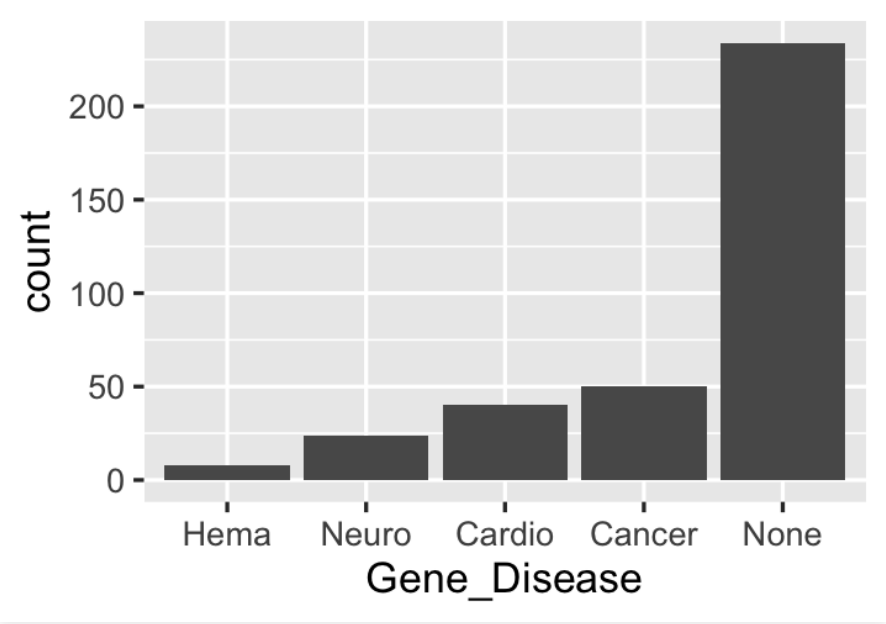
```
num_genes %>%
  mutate(HPO_ID = fct_reorder(HPO_ID, ngenes)) %>%
ggplot() +
  geom_point(aes(HPO_ID, ngenes)) +
  theme(axis.text.x=element_text(hjust=1,angle=45))
```



Ordering factor levels

- `fct_reorder()` takes in a factor vector and a numeric vector that is used to sort the levels
- `fct_infreq()` orders by how often the levels appear in the factor vector
- `fct_rev()` reverses the order

```
hpo_data %>%
  mutate(Gene_Disease = Gene_Disease %>%
    fct_infreq() %>%
    fct_rev()) %>%
ggplot(aes(Gene_Disease)) +
  geom_bar()
```



Recoding factor levels

```
hpo_data %>%
  group_by(Gene_Disease) %>%
  summarize(count=n()) %>%
  arrange(desc(count))
# A tibble: 5 x 2
  Gene_Disease count
  <fct>        <int>
1 None           234
2 Cancer          50
3 Cardio          40
4 Neuro           24
5 Hema             8
```

- These factor levels are vague
- we can change them with `fct_recode()`

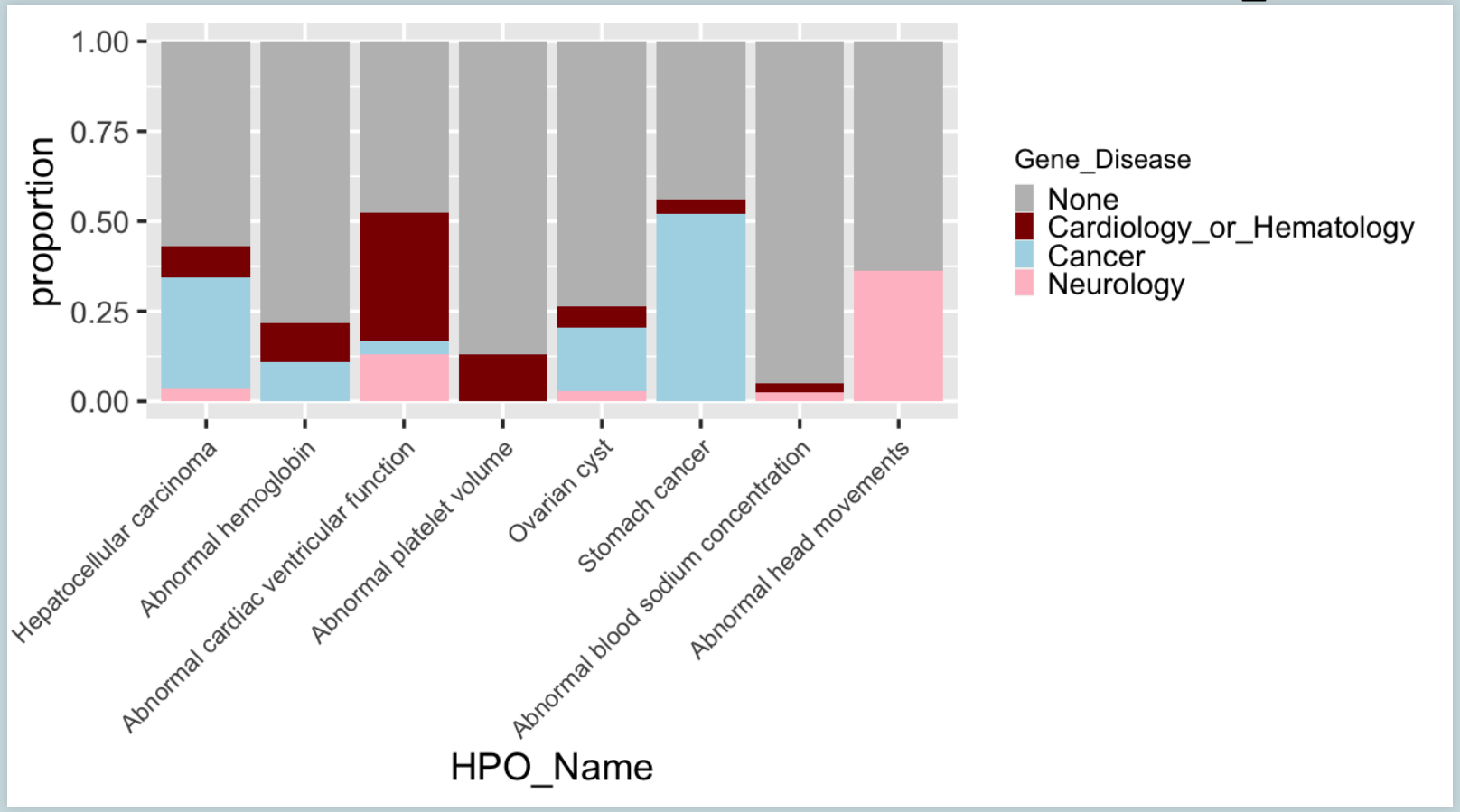
Recoding factor levels

```
hpo_data %>%
  mutate(Gene_Disease = fct_recode(
    Gene_Disease,
    "Cardiology" = "Cardio",
    "Neurology" = "Neuro",
    "Hematology" = "Hema"
  )) %>%
  pull(Gene_Disease) %>%
  levels()
[1] "None"       "Cardiology" "Cancer"     "Neurology"  "Hematology"
```

- Levels that you don't mention are left alone
- You can code multiple old levels to one new level (also see `?fct_collapse()`)

Exercise: plotting proportions across phenotypes

How does the proportion of disease associations (Cancer, Cardio, etc.) for the genes mapped to each given HPO ID vary across each phenotype (HPO_Name)? Recreate the following plot from the `hpo_data` data:



`geom_bar()` may be useful. `scale_fill_manual()` may also be useful to associate the disease categories with particular colors if you so choose.

Dates and times

Date and time objects in R

- Dates and times are not well captured as strings (since you can't do math with them) or as numbers (since date-time arithmetic is irregular), so they need their own data type
- Before we tried making dates into factors with the appropriate order, but it's a pain to always have to set this up and it misses edge cases.
- Luckily, `tidyverse` has a solution. We'll use the `tidyverse`-adjacent `lubridate` package to provide work with that data type. We have to install and load it as well.

```
library(lubridate) # install.package("lubridate")
```

- The data types that we will work with are `date` and `dttm` (date-time, also unhelpfully called POSIXct elsewhere in R).

```
tibble(date_time = now(), # a date-time (dttm), prints as a string
       date = today()) # a date, also prints as a string
# A tibble: 1 x 2
  date_time           date
  <dttm>              <date>
1 2021-07-19 17:36:52 2021-07-19
```

- Always use the simplest possible data type that works for your needs. Date-times are more complicated because of the need to handle time zones.

Creating dates from a string (or number)

- Dates:

```
c(ymd("2017-01-31"), mdy("January 31st, 2017"),
  dmy("31-Jan-2017"), ymd(20170131))
[1] "2017-01-31" "2017-01-31" "2017-01-31" "2017-01-31"
```

- Date-times

```
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"
```

- All of these get printed out as strings, but they are actually `date` or `dttm` objects
- these also all work on vectors, even if they fromatted heterogenously

```
x <- c(20090101, "2009-01-02", "2009 01 03", "2009-1-4",
       "2009-1, 5", "Created on 2009 1 6", "200901 !!! 07")
ymd(x)
[1] "2009-01-01" "2009-01-02" "2009-01-03" "2009-01-04" "2009-01-05"
[6] "2009-01-06" "2009-01-07"
```

- these functions are all pretty smart and detect most common delimitors (-, /, :, etc.), but you should check their input and output with sample data to make sure they are working correctly

Creating dates from components

- Sometimes the dates and times you get are split up (usually in columns)

```
gtex_dates = read_csv("https://raw.githubusercontent.com/erflynn/r4ds-
courses/advance-2020/data/gtex_metadata/gtex_sample_mdy.csv")
gtex_dates %>%
  select(contains("iso")) %>%
  head(5) # look at relevant columns
# A tibble: 5 x 3
  iso_month iso_day iso_year
      <dbl>   <dbl>    <dbl>
1         5      21     2013
2        10       8     2013
3        10       8     2013
4        10       8     2013
5        10      11     2013
```

Creating dates from components

- You can join them into dates or datetimes with `make_date()` or `make_datetime()`

This data frame contains experimental date and time information for GTEx experiments; however, it's in a messy format so we're going to clean it up. This includes the dates of rna isolation (“iso”) and expression (“expr”) for each sample and the start/end time for sample going into pax gene fixative (“pax”). Here, the RNA isolation and expression dates are separated across three columns, we can put this together

```
gtex_dates_clean1 = gtex_dates %>%
  mutate(iso_date = make_date(iso_year, iso_month, iso_day),
         expr_date = make_date(expr_year, expr_month, expr_day))
gtex_dates_clean1 %>%
  select(contains("iso") | contains("expr")) %>%
  head(3)
# A tibble: 3 x 8
  iso_month iso_day iso_year iso_date   expr_month expr_day expr_year expr_date
      <dbl>   <dbl>    <dbl> <date>          <dbl>    <dbl>     <dbl> <date>
1         5      21     2013 2013-05-21          1       15      2014 2014-01-15
2        10       8     2013 2013-10-08          2        9      2014 2014-02-09
3        10       8     2013 2013-10-08          2        9      2014 2014-02-09

# make sure to remove the extra columns we don't need anymore
# but do this after you've double checked you did it right!
gtex_dates_clean2 = gtex_dates_clean1 %>%
  select(-contains("month"), -contains("year"), -contains("day"))
```

Creating dates from components

Information about when a sample went into PAXgene fixative and was taken out is in the pax columns, divided into both dates and times.

```
gtex_dates_clean2 %>%
  select(contains("pax")) %>%
  tail(2)
# A tibble: 2 x 4
  pax_start_date pax_end_date pax_start_time pax_end_time
  <date>         <date>       <chr>          <chr>
1 2013-03-19     2013-03-19   1247           2008
2 2012-10-03     2012-10-04   2205           0524
```

- note that `pax_start_time` and `pax_end_time` are numeric and in an HHMM format! We have to fix that first (why is this bad?)

```
gtex_dates_clean3 = gtex_dates_clean2 %>%
  mutate(pax_start_hr = str_sub(pax_start_time, 1, -3) %>% as.numeric(),
         pax_start_min = str_sub(pax_start_time, -2, -1) %>% as.numeric(),
         pax_end_hr = str_sub(pax_end_time, 1, -3) %>% as.numeric(),
         pax_end_min = str_sub(pax_end_time, -2, -1) %>% as.numeric())
gtex_dates_clean %>% select(contains("pax")) %>% tail(2)
Error in eval(lhs, parent, parent): object 'gtex_dates_clean' not found
# remove extra columns again
gtex_dates_clean4 = gtex_dates_clean3 %>% select(-contains("time"), -
contains("time"))
```

Creating dates from components

- Now we can aggregate the columns with PAX timing information into `dttm`s. Here we use the functions `year()`, `month()`, and `day()` to extract this information from the dates (we'll discuss this more in a few slides).

```
gtex_dates_clean5 = gtex_dates_clean4 %>%
  mutate(
    pax_start_dt = make_datetime(
      year(pax_start_date), month(pax_start_date), day(pax_start_date),
      pax_start_hr, pax_start_min),
    pax_end_dt = make_datetime(
      year(pax_start_date), month(pax_end_date), day(pax_end_date),
      pax_end_hr, pax_end_min)
  )
gtex_dates_clean5 %>%
  select(contains("pax")) %>%
  head(3)
# A tibble: 3 x 8
  pax_start_date pax_end_date pax_start_hr pax_start_min pax_end_hr pax_end_min
  <date>         <date>               <dbl>         <dbl>      <dbl>       <dbl>
1 2012-10-15     NA                      NA            NA         NA          NA
2 2013-06-27     2013-06-28              22            32         14          23
3 2013-01-28     2013-01-28               5            32         21          22
# … with 2 more variables: pax_start_dt <dttm>, pax_end_dt <dttm>

# remove columns we just used
gtex_dates_clean6 <- gtex_dates_clean5 %>%
  select(-pax_start_date, -pax_end_date, -contains("hr"), -contains("min"))
```

Cleaned date data

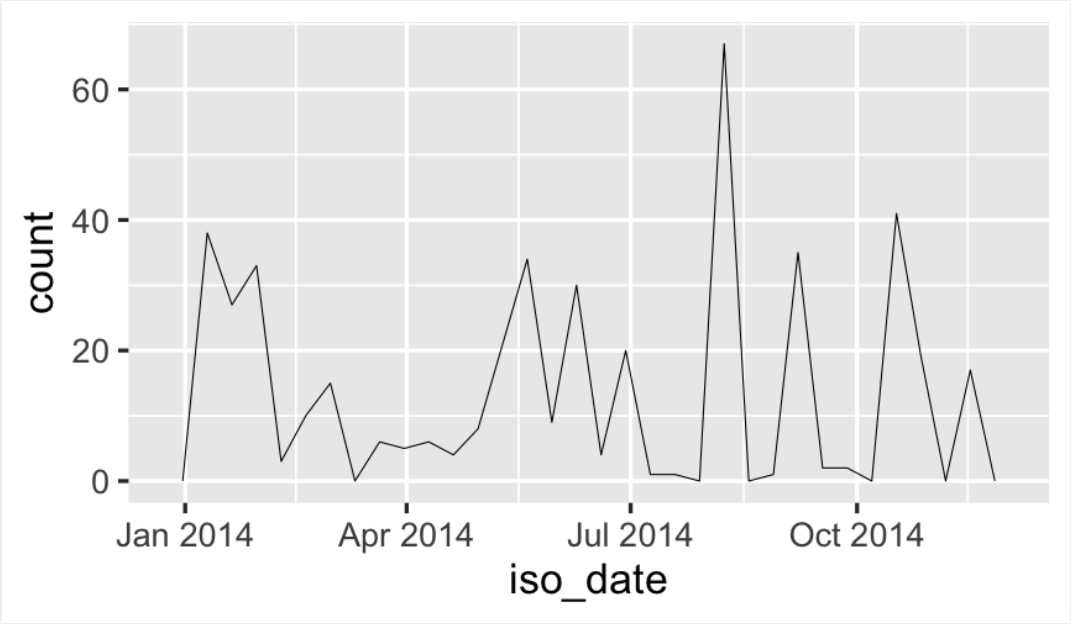
Take a look at both the data frame we started with and the one we have cleaned. Which do you prefer? Which is clearer?

```
head(gtex_dates)
# A tibble: 6 x 12
  subject_id tissue expr_month expr_day expr_year iso_month iso_day iso_year
  <chr>      <chr>        <dbl>    <dbl>     <dbl>     <dbl>   <dbl>    <dbl>
1 GTEX-11DXZ Blood            1       15      2014         5      21     2013
2 GTEX-11DXZ Liver            2        9      2014        10       8     2013
3 GTEX-11DXZ Adren…           2        9      2014        10       8     2013
4 GTEX-11DXZ Heart            2        9      2014        10       8     2013
5 GTEX-11DXZ Blood…           1       23      2014        10      11     2013
6 GTEX-11DXZ Lung             3       22      2014         9      25     2013
# … with 4 more variables: pax_start_date <date>, pax_end_date <date>,
#   pax_start_time <chr>, pax_end_time <chr>
head(gtex_dates_clean6)
# A tibble: 6 x 6
  subject_id tissue iso_date   expr_date  pax_start_dt
  <chr>      <chr>  <date>     <date>     <dttm>
1 GTEX-11DXZ Blood  2013-05-21 2014-01-15 NA
2 GTEX-11DXZ Liver  2013-10-08 2014-02-09 2013-06-27 22:32:00
3 GTEX-11DXZ Adren… 2013-10-08 2014-02-09 2013-01-28 05:32:00
4 GTEX-11DXZ Heart  2013-10-08 2014-02-09 2013-07-01 12:33:00
5 GTEX-11DXZ Blood… 2013-10-11 2014-01-23 2013-02-14 01:52:00
6 GTEX-11DXZ Lung   2013-09-25 2014-03-22 2012-10-31 06:31:00
# … with 1 more variable: pax_end_dt <dttm>
```

Plotting with dates

- ggplot2 understands `dates` and `ddtm`s

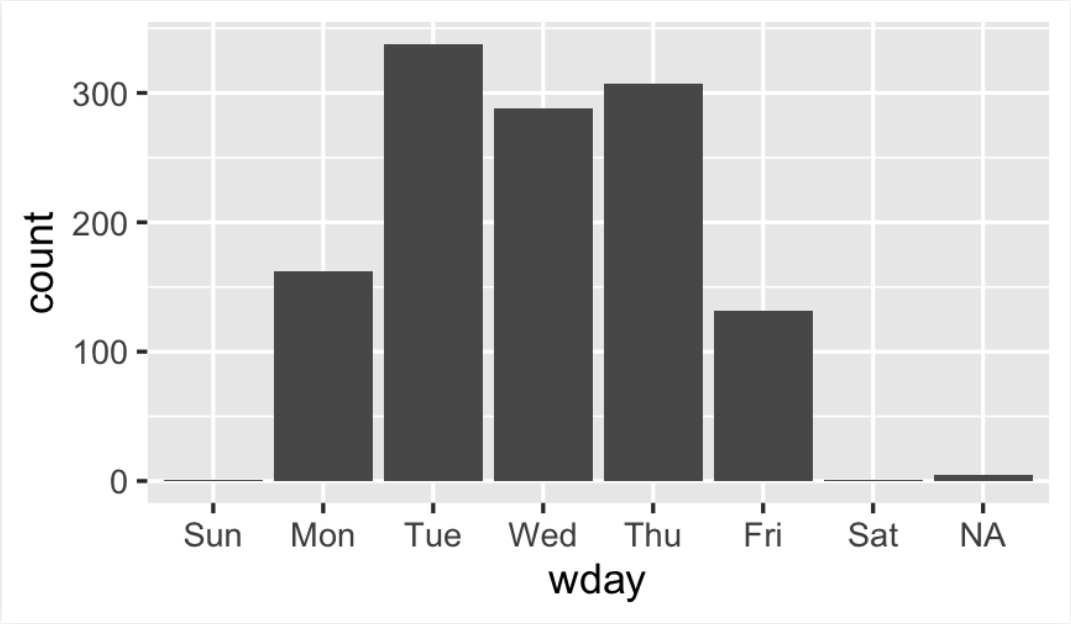
```
# get the dates samples were isolated from just 2014
gtex_dates_clean6 %>%
  filter(ymd(20140101) < iso_date & iso_date < ymd(20141230)) %>%
ggplot(aes(iso_date)) +
  geom_freqpoly(binwidth=10) # bin by every 10 days (if we had dttm, the unit is
seconds)
```



Accessing dttm elements

- You can pull out individual parts of the date with the accessor functions `year()`, `month()`, `mday()` (day of the month), `yday()` (day of the year), `wday()` (day of the week), `hour()`, `minute()`, and `second()`

```
# look at the dates RNA isolation was perfomed. what do you notice?
gtex_dates_clean6 %>%
  mutate(wday = wday(iso_date, label = TRUE)) %>%
  ggplot(aes(x = wday)) +
    geom_bar()
```



Date-time arithmetic

We will disucss `dttm` subtraction, addition, and division. These require the following data types:

- durations, which represent an exact number of seconds.
- periods, which represent human units like weeks and months.
- intervals, which represent a starting and ending point.

Durations

- Durations represent an exact span of time (i.e. in seconds)

```
usa_age <- today() - ymd(17760704)
usa_age
Time difference of 89499 days
```

- Subtracting `dttm`s in R gives something called a `difftime`, which ambiguously represents differences in weeks, days, hours, or seconds. A `duration` always uses seconds so it is preferable.
- You can conver to a duration with `as.duration()`

```
as.duration(usa_age)
[1] "7732713600s (~245.03 years)"
```

- `dseconds()`, `dminutes()`, `dhours()`, `ddays()`, `dweeks()`, and `dyears()` make durations of the given length of time and are vectorized

```
ddays(194)
[1] "16761600s (~27.71 weeks)"
dweeks(1:4)
[1] "604800s (~1 weeks)"  "1209600s (~2 weeks)" "1814400s (~3 weeks)"
[4] "2419200s (~4 weeks)"
```

Duration arithmetic

- Durations can be added together and multiplied by numbers

```
2 * (as.duration(usa_age) + dyears(1) + dweeks(12) + dhours(15))
[1] "15543165600s (~492.53 years)"
```

- Or added and subtracted from `ddtm`s

```
today() - as.duration(usa_age) # should give July 4 1776
[1] "1776-07-04"
```

- Weird things can happen with time zones

```
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) # not 1 PM anymore?!
[1] "2016-03-13 14:00:00 EDT"
```

Periods

- `Period`s are like `Duration`s but in "natural" human units
- `seconds()`, `minutes()`, `hours()`, `days()`, `weeks()`, `months()`, and `years()` make durations of the given length of time

```
days(194)
[1] "194d 0H 0M 0S"
weeks(5:9)
[1] "35d 0H 0M 0S" "42d 0H 0M 0S" "49d 0H 0M 0S" "56d 0H 0M 0S" "63d 0H 0M 0S"
```

- Question: Why is there `months()` but no `dmonths()`?

Period arithmetic

- Periods can be added together and multiplied by numbers, just like Durations

```
2 * (dyears(1) + dweeks(12) + dhours(15))
[1] "77738400s (~2.46 years)"
```

- Or added and subtracted from `ddtm`s

```
today() - as.period(usa_age) # should give July 4 1776
[1] "1776-07-04"
```

- And they do more of what you would expect given daylight savings

```
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 + days(1) # it knows that one "day" on this day is actually 23 hours, not
24
[1] "2016-03-13 13:00:00 EDT"
```

Example using periods

- Let's calculate the length of time samples were in PaxGene fixative and look at how it differs when we look at it as a Duration and a Period. What do you prefer?

```
gtex_dates_clean6 %>%
  mutate(pax_time=pax_end_dt-pax_start_dt)  %>%
  mutate(pax_time_d=as.duration(pax_time),
         pax_time_p=as.period(pax_time)) %>%
  select(contains("pax")) %>%
  tail(2)
# A tibble: 2 x 5
  pax_start_dt        pax_end_dt          pax_time pax_time_d
  <dttm>              <dttm>              <drtn>   <Duration>
1 2013-03-19 12:47:00 2013-03-19 20:08:00 7.35000… 26460s (~7.35 hours)
2 2012-10-03 22:05:00 2012-10-04 05:24:00 7.31666… 26340s (~7.32 hours)
# … with 1 more variable: pax_time_p <Period>
```

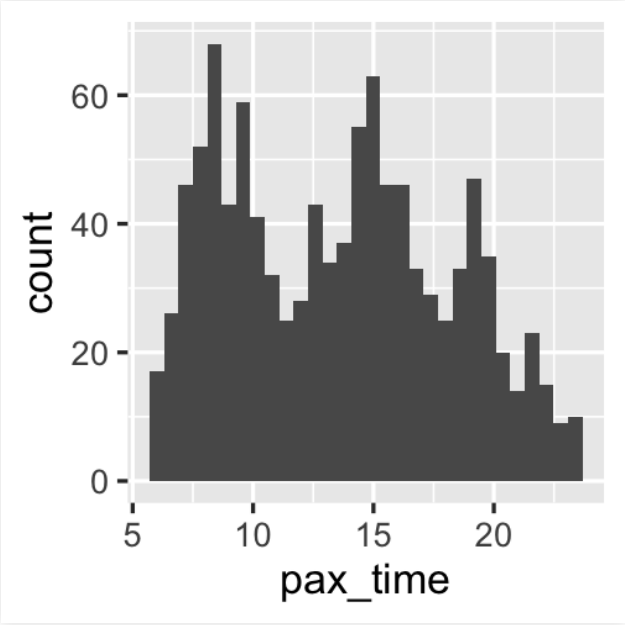
You take a look at the data and realize some of the difftimes are negative. What is going on here?

```
gtex_dates_clean6 %>%
  mutate(pax_time=pax_end_dt-pax_start_dt) %>%
  filter(pax_time < 0) %>%
  select(pax_start_dt, pax_end_dt, pax_time)
# A tibble: 2 x 3
  pax_start_dt        pax_end_dt          pax_time
  <dttm>              <dttm>              <drtn>
1 2013-12-31 07:38:00 2013-01-01 02:53:00 -8740.75 hours
2 2014-12-31 12:39:00 2014-01-01 02:03:00 -8746.60 hours
```

Plotting difftimes

We can also plot difftimes. Lets' look at the distribution of times in pax-genes fixative. What are the units here?

```
gtex_dates_clean6 %>% # look at the rest of the data
  mutate(pax_time=pax_end_dt-pax_start_dt) %>%
  filter(pax_time > 0) %>%
  ggplot(aes(x=pax_time))+
  geom_histogram()
Don't know how to automatically pick scale for object of type difftime. Defaulting
to continuous.
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Example using periods

To fix this we add a year to some of the ones with negative times. Which of these work and why?

```
gtex_dates_clean6 %>%
  select(contains("pax")) %>%
  mutate(pax_time=(pax_end_dt-pax_start_dt))  %>%
  mutate(neg_pax=(pax_time < 0)) %>%
  mutate(pax_time_adj=pax_time+years(neg_pax)) %>%
  filter(neg_pax)
```

```
gtex_dates_clean6 %>%
  select(contains("pax")) %>%
  mutate(pax_time=(pax_end_dt-pax_start_dt))  %>%
  mutate(neg_pax=(pax_time < 0)) %>%
  mutate(pax_end_dt_adj=pax_end_dt+years(neg_pax)) %>%
  filter(neg_pax)
```

```
gtex_dates_clean6 %>%
  select(contains("pax")) %>%
  mutate(pax_time_p=as.period(pax_end_dt-pax_start_dt))  %>%
  mutate(neg_pax=(pax_time_p < 0)) %>%
  mutate(pax_time_adj=pax_time_p+years(neg_pax)) %>%
  filter(neg_pax)
```

Intervals

- An `interval` is a specific span of time with a start and end date(-time)
- They can be defined with the `%--%` operator, which you can read as “from… until” as in “from July 4 1776 until today”

```
mdy("July 4 1776") %--% today()
[1] 1776-07-04 UTC--2021-07-19 UTC
```

- You can use %within% to see if a date or `dttm` falls in the interval

```
gtex_dates_clean6 %>%
  filter(iso_date %within% (mdy("feb 15 2013") %--% mdy("feb 14 2014"))) %>%
  head()
# A tibble: 6 x 6
  subject_id tissue iso_date   expr_date  pax_start_dt
  <chr>      <chr>  <date>     <date>     <dttm>
1 GTEX-11DXZ Blood  2013-05-21 2014-01-15 NA
2 GTEX-11DXZ Liver  2013-10-08 2014-02-09 2013-06-27 22:32:00
3 GTEX-11DXZ Adren… 2013-10-08 2014-02-09 2013-01-28 05:32:00
4 GTEX-11DXZ Heart  2013-10-08 2014-02-09 2013-07-01 12:33:00
5 GTEX-11DXZ Blood… 2013-10-11 2014-01-23 2013-02-14 01:52:00
6 GTEX-11DXZ Lung   2013-09-25 2014-03-22 2012-10-31 06:31:00
# … with 1 more variable: pax_end_dt <dttm>
```

Exercise: first days of the month

Create a vector of dates giving the first day of every month in 2015. Create a vector of dates giving the first day of every month in the current year.

lubridate cheat sheet

Data input/output

Rationale

- Sometimes R fails to read in the data from a file
- Or you will read data into R and find strange errors when you try to manipulate it
- This is often caused by type mismatches- e.g. you expected a column to have been read in as a factor, but it was actually read in as a logical.

```
file = readr_example("challenge.csv")
challenge = read_csv(file)
Parsed with column specification:
cols(
  x = col_double(),
  y = col_logical()
)
Warning: 1000 parsing failures.
 row col           expected     actual
file
1001   y 1/0/T/F/TRUE/FALSE 2015-01-16
'/Library/Frameworks/R.framework/Versions/3.6/Resources/library/readr/extdata/challenge.csv'

1002   y 1/0/T/F/TRUE/FALSE 2018-05-18
'/Library/Frameworks/R.framework/Versions/3.6/Resources/library/readr/extdata/challenge.csv'

1003   y 1/0/T/F/TRUE/FALSE 2015-09-05
'/Library/Frameworks/R.framework/Versions/3.6/Resources/library/readr/extdata/challenge.csv'

1004   y 1/0/T/F/TRUE/FALSE 2012-11-28
'/Library/Frameworks/R.framework/Versions/3.6/Resources/library/readr/extdata/challenge.csv'
```

Diagnosing intake errors

- Use `readr::problems()` on the returned object to learn more about the errors

```
problems(challenge)
# A tibble: 1,000 x 5
     row col   expected        actual   file
   <int> <chr> <chr>           <chr>    <chr>
 1  1001 y     1/0/T/F/TRUE/… 2015-01… '/Library/Frameworks/R.framework/Version…
 2  1002 y     1/0/T/F/TRUE/… 2018-05… '/Library/Frameworks/R.framework/Version…
 3  1003 y     1/0/T/F/TRUE/… 2015-09… '/Library/Frameworks/R.framework/Version…
 4  1004 y     1/0/T/F/TRUE/… 2012-11… '/Library/Frameworks/R.framework/Version…
 5  1005 y     1/0/T/F/TRUE/… 2020-01… '/Library/Frameworks/R.framework/Version…
 6  1006 y     1/0/T/F/TRUE/… 2016-04… '/Library/Frameworks/R.framework/Version…
 7  1007 y     1/0/T/F/TRUE/… 2011-05… '/Library/Frameworks/R.framework/Version…
 8  1008 y     1/0/T/F/TRUE/… 2020-07… '/Library/Frameworks/R.framework/Version…
 9  1009 y     1/0/T/F/TRUE/… 2011-04… '/Library/Frameworks/R.framework/Version…
10  1010 y     1/0/T/F/TRUE/… 2010-05… '/Library/Frameworks/R.framework/Version…
# … with 990 more rows
```

- This tells us that `read_csv()` was expecting the `y` column to be logical, but when we look at what was actually in the file at rows 1001+, there are what appear to be dates!
- This happens because `read_csv()` does not know what type of data are in the file- you haven't told it, so it has to guess.
- The way it guesses is by checking the first 1000 rows of each column and picking the most likely data type.
- You can tell `read_csv()` to check more rows before guessing by using the `guess_max` argument

Specifying data types

- In general, you may already know what types the columns should be, so you can provide those to `read_csv()`.

```
challenge = read_csv(file,
   col_types = cols(
     y = col_date()
   ))
head(challenge)
# A tibble: 6 x 2
      x y
  <dbl> <date>
1   404 NA
2  4172 NA
3  3004 NA
4   787 NA
5    37 NA
6  2332 NA
```

- This is a more robust solution than using more rows to guess
- Now we see that the problem was caused because the first 1000 rows of `y` are NAs
- Column types are provided to read_csv as named arguments to `cols()`, which itself is a named argument to `col_types`.
- You do not need to specify all columns (here we let it guess what `x` is) but it is often good practice to do so if possible

Specifying data types

- Note that `col_date()` guessed the formatting of the dates (correctly in this case).
- You can specify it using the `format` argument.

```
challenge = read_csv(file,
   col_types = cols(
     y = col_date(format="%Y-%m-%d")
   ))
tail(challenge)
# A tibble: 6 x 2
      x y
  <dbl> <date>
1 0.805 2019-11-21
2 0.164 2018-03-29
3 0.472 2014-08-04
4 0.718 2015-08-16
5 0.270 2020-02-04
6 0.608 2019-01-06
```

- You can also use a character string as a shortcut

```
challenge = read_csv(file, col_types = "dD")
```

- Each character stands for the datatype of the corresponding column.
  - In this case, `d` in position 1 means the first column is a double, `D` in position two says the second column is a date.

Specifying data types

- Factors can also be read in with a high level of control

```
df = readr_example("mtcars.csv")  %>%
read_csv(col_types = cols(
  cyl = col_factor(levels=c("4", "6", "8"))
))
```

- This will let you catch unexpected factor levels and set the proper order up-front!
- To allow all levels, don't use the `levels` argument

```
df = readr_example("mtcars.csv")  %>%
read_csv(col_types = cols(
  cyl = col_factor()
))
```

Non-csv flat files

- Besides .csv, you may find data in .tsv (tab-separated values) and other more exotic formats.
- Many of these are still delimited text files ("flat files"), which means that the data are stored as text with special characters between new lines and columns. This is an exmaple .csv:

```
toy_csv = "1,2,3\n4,5,6"
```

- This is an example .tsv

```
toy_tsv = "1\t2\t3\n4\t5\t6"
```

- The only difference is the **delimiter** which is the character that breaks up columns.

Non-csv flat files

- Both can be read in using `read_delim()`

```
read_delim("1,2,3\n4,5,6", delim=",", col_names = c("x","y","z"))
# A tibble: 2 x 3
      x     y     z
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5     6
read_delim("1\t2\t3\n4\t5\t6", delim="\t", col_names = c("x","y","z"))
# A tibble: 2 x 3
      x     y     z
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5     6
```

- `read_csv()` is just a shortcut to `read_delim()` that has `delim=","` hardcoded in.

Non-flat files

- There are also other packages that let you read in from other formats
  - `haven` reads in SPSS, Stata, and SAS files
  - `readxl` reads in `.xls` and `.xlsx`
  - `DBI` with a database backend (e.g. `odbc`) reads in from databases
  - `jsonlite` and `xml2` for heirarchical data
  - `rio` for more esoteric formats

Writing files

```
write_csv(challenge, "challenge.csv")
```

- metadata about column types is lost when writing to .csv
- use `write_rds()` (and `read_rds()`) to save to a binary R format that preserves column types

```
write_rds(challenge, "challenge.rds")
```

Exercise: file I/O

What will happen if I run this code?

```
file = readr_example("challenge.csv")
challenge = read_csv(file,
   col_types = cols(
     y = col_date() # the first 1000 rows of y are NA so need to specify it's a
date
   ))
write_csv(challenge, "challenge.csv")
challenge2 = read_csv("challenge.csv")
```

Understanding paths

When you read and write files from R, you will need to know where the file is located – either on your computer or somewhere on the internet. If you've noticed so far, we've only been reading files from URLs on GitHub. However, a lot of the time you will want to read from and write to files on your computer.

Finding out where you are

Figuring out where you are (or the “working directory”) is important for looking at where files are. The term “directory” means effectively the same thing as a “folder”.

You can get the path of working directory using the command

```
getwd()
```

Try this right now – what gets printed out? Where are you located?

You can also set the working directory using the command `setwd()` where you fill in the path you want as the argument, for example:

```
setwd("/Users/home/my_folder/")
```

(While we recommend you don't run this, it also shouldn't work because this path probably doesn't exist on your computer – more on that later.)

In general, you want to use `setwd()` very sparingly. It's not good practice to change where you are, and it can mess up your code uninentionally. However, there are some R package functions that require you to be in a particular directory (usually the same directory as the data) in order to run. For now, just know it exists and avoid using it if you can.

Relative and absolute paths

Paths describe where files are located, and there are two types of paths: relative and absolute. Relative paths describe the location of a file is relative to the working directory. Some examples of relative paths are:

```
"challenge.csv" # in the directory you're in
"data/challenge.csv" # in a directory called "data/" within your working directory
"../challenge.csv" # in the directory above you
"../../challenge.csv" # in the directory two above you
```

It's important to note that if your working directory changes, then the relative path to a file will no longer work.

While it's considered better coding practice to use relative paths (we'll talk about this in a later lecture), sometimes it is helpful to use an absolute path to make sure it works regardless of where you are.

Absolute paths list out the entire location of a file, such as:

```
"/home/users/Documents/my_file.csv" # path on Linux or Mac
"~/Desktop/my_file.csv" # "~" means the home directory
"C:\Users\me\projects\my_file.csv" # typical path on Windows
```

Writing and reading to local directories

Let's try reading and writing from your current directory.

```
file = readr_example("mtcars.csv")
print(file) # look at the path
[1]
"/Library/Frameworks/R.framework/Versions/3.6/Resources/library/readr/extdata/mtcars.csv"

mtcars = read_csv(file)
head(mtcars)
# A tibble: 6 x 11
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
6  18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

The `readr_example()` command grabs the path of an example file that is installed with tidyverse. Take a look at the path – this is where the file is located on your computer.

Now let's write it out:

```
write_csv(mtcars, "mtcars.csv")
```

The file should appear in the “Files” pane on the bottom right of RStudio after running this code. Does it?

Using the Files pane

Try reading it back in:

```
mtcars2 = read_csv("mtcars.csv")
```

Now use RStudio to create a folder called `data/` and move the `mtcars.csv` file to `data/`. Try reading it in from here now:

```
mtcars3 = read_csv("data/mtcars.csv")
```

The `tab` button is very helpful when specifying files - it should autocomplete the file name. Try this.

Exercise: Downloading a file and reading it into R

First, download this file from GitHub:

```
"https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-2021/data/poly.csv"
```

Go to the link, right click, and then “Save As” so it is on your local machine.

Now we have to figure out where the file is so we can read it in.

On a Windows machine, the path can be found by right-clicking the file icon, and selecting “Properties.” You'll see the file path, minus the name of the file, in the “General” tab, labeled “location.” The full file name will be this path plus the name of the file .

On a Mac, you can copy the full file path by right-clicking on the file or folder in the Mac Finder. While in the right-click menu, hold down the option key. The right-click menu will then reveal an option to “Copy "filename” as Pathname.“

Fill this in to `read_csv` and see if you can read it in! Think about how you would do this to read in a file from your computer.

Some of these materials were adapted from: Introduction to R Illinois University Library https://guides.library.illinois.edu/c.php?g=347944&p=2345554

readr cheat sheet