# PSYC 259:
# Principles of Data Science

## Week 3: Part 2

# Outline - Data Structure

1. Data types/factors
2. Data wrangling with *dplyr*
3. Data wrangling tutorial

# Data types and factors

# Data types

- Why have pre-defined types?
  - Allows software to efficiently store data in memory
    - If a value is an integer (1, 2, 3, 4) storing it as an integer makes calculations easier compared to storing it as a double (2.3454248042462408 6)
  - Allows software to implement rules about transformations
    - Addition/subtraction for a date follows different rules compared with integers/double
    - “Less than” makes sense when comparing numbers, but not when comparing strings

# Common data types in R reflect how values are stored

- Numeric
  - integer - 1, 2, 3
  - double - 1.12124, 5.235235
- Character - “hello”
- Logical - T/F (TRUE/FALSE)
- Date/time
- Factor
- Use typeof() to check type

# Logical statements in R

- Comparisons evaluate as T or F
  - 1 > 0 #TRUE
  - 1 == 1 #TRUE
  - 1 != 1 #FALSE
  - “s” == “S” #FALSE
  - 1 > 0 | 0 > 1 #TRUE
  - 1 >0 & 0 > 1 #FALSE
  - !(1 == 1) #FALSE

# Other helpful logical functions

- ifelse(logical, if_true, if_false)
  - x <- c(-1, 0, 1)
  - ifelse(x > 0, “positive”, “negative”)
  - returns: “positive”, “positive”, “negative”
- is.na() checks if a value is NA
  - x <- c(1, 2, NA)
  - is.na(x) returns: FALSE, FALSE, TRUE
- Any logical with NA returns NA

# Checking/converting types

- as.factor, as.numeric, as.Date, as.character take a value and coerce it to that type
  - as.numeric(“1”) returns 1
- is.factor, is.numeric, is.character check if something is a particular type
  - is.numeric(1) #TRUE
  - is.character(as.numeric(“1”)) #FALSE

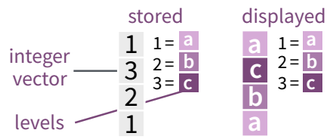
# Factors in R represent categories

- x <- factor(x, levels = c(1,2,3), labels = c("rarely", "neutral", "frequently")
- levels restrict the possible set of values
- x[2] <-  4       #error, will be stored as NA
- levels are *ordered*, which carries forward to output, modeling, graphics, etc.
- factors work as dummy codes; use as.numeric(factor) to treat as a continuous variable in models
- labels will display throughout R, which is lovely

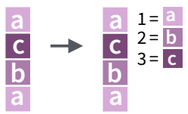
# Factors with forcats : : **CHEAT SHEET**

The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

## Factors



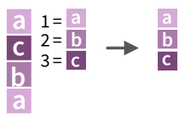

R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the values associated with them.



*Create a factor with factor()*

**factor**(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA) Convert a vector to a factor. Also **as_factor**.

```
f <- factor(c("a", "c", "b", "a"),
            levels = c("a", "b", "c"))
```

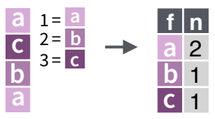


*Return its levels with levels()*

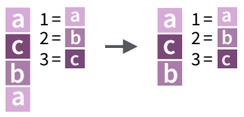
**levels**(x) Return/set the levels of a factor. *levels(f); levels(f) <- c("x","y","z")*

*Use unclass() to see its structure*

## Inspect Factors

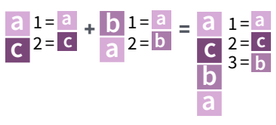


**fct_count**(f, sort = FALSE) Count the number of values with each level. *fct_count(f)*



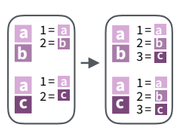
**fct_unique**(f) Return the unique values, removing duplicates. *fct_unique(f)*

## Combine Factors



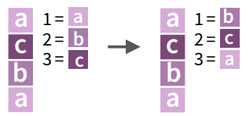
**fct_c**(…) Combine factors with different levels.

```
f1 <- factor(c("a", "c"))
f2 <- factor(c("b", "a"))
fct_c(f1, f2)
```

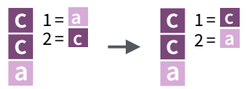


**fct_unify**(fs, levels = lvls_union(fs)) Standardize levels across a list of factors. *fct_unify(list(f2, f1))*

## Change the order of levels

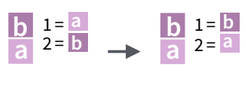


**fct_relevel**(.f, ..., after = 0L) Manually reorder factor levels. *fct_relevel(f, c("b", "c", "a"))*

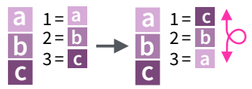


**fct_infreq**(f, ordered = NA) Reorder levels by the frequency in which they appear in the data (highest frequency first).

```
f3 <- factor(c("c", "c", "a"))
fct_infreq(f3)
```

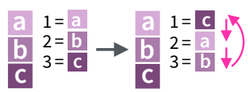


**fct_inorder**(f, ordered = NA) Reorder levels by order in which they appear in the data. *fct_inorder(f2)*

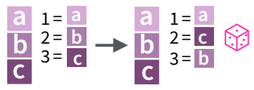


**fct_rev**(f) Reverse level order.

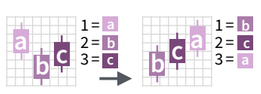
```
f4 <- factor(c("a","b","c"))
fct_rev(f4)
```



**fct_shift**(f) Shift levels to left or right, wrapping around end. *fct_shift(f4)*

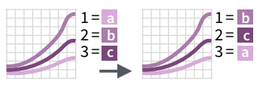


**fct_shuffle**(f, n = 1L) Randomly permute order of factor levels. *fct_shuffle(f4)*



**fct_reorder**(.f, .x, .fun=median, ..., .desc = FALSE) Reorder levels by their relationship with another variable.

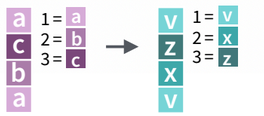
```
boxplot(data = iris, Sepal.Width ~
  fct_reorder(Species, Sepal.Width))
```



**fct_reorder2**(.f, .x, .y, .fun = last2, ..., .desc = TRUE) Reorder levels by their final values when plotted with two other variables.

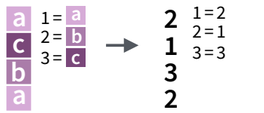
```
ggplot(data = iris,
  aes(Sepal.Width, Sepal.Length,
    color = fct_reorder2(Species,
    Sepal.Width, Sepal.Length))) +
  geom_smooth()
```

## Change the value of levels

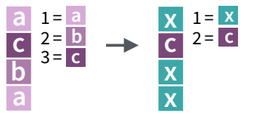


**fct_recode**(.f, ...) Manually change levels. Also **fct_relabel** which obeys purrr::map syntax to apply a function or expression to each level.

```
fct_recode(f, v = "a", x = "b", z = "c")
fct_relabel(f, ~ paste0("x", .x))
```

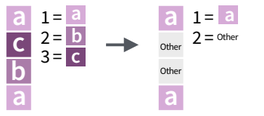


**fct_anon**(f, prefix = "")) Anonymize levels with random integers. *fct_anon(f)*



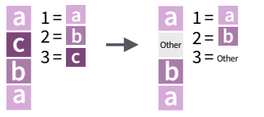
**fct_collapse**(.f, ...) Collapse levels into manually defined groups.

```
fct_collapse(f, x = c("a", "b"))
```



**fct_lump**(f, n, prop, w = NULL, other_level = "Other", ties.method = c("min", "average", "first", "last", "random", "max")) Lump together least/most common levels into a single level. Also **fct_lump_min**.

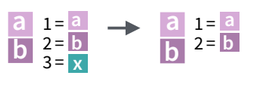
```
fct_lump(f, n = 1)
```



**fct_other**(f, keep, drop, other_level = "Other") Replace levels with "other."

```
fct_other(f, keep = c("a", "b"))
```

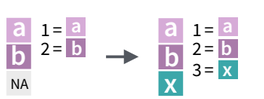
## Add or drop levels



**fct_drop**(f, only) Drop unused levels.

```
f5 <- factor(c("a","b"),c("a","b","x"))
f6 <- fct_drop(f5)
```

**fct_expand**(f, …) Add levels to a factor. *fct_expand(f6, "x")*



**fct_explicit_na**(f, na_level="(Missing)") Assigns a level to NAs to ensure they appear in plots, etc.

```
fct_explicit_na(factor(c("a", "b", NA)))
```

# Useful forcats functions

- fct_count = count # of each factor level
- fct_relevel, fct_rev = reorder levels
- fct_recode, fct_collapse = reassign or combine factor levels

# Data wrangling with dplyr

# General dplyr notes

- All dplyr transformations take a data argument (first argument, or piped in)
  - select(data, id)
  - ds %>% select(id)
- All dplyr transformations are temporary unless saved back to the dataset
  - ds <- ds %>% select(id)

# General dplyr notes

- All dplyr transformations can be chained with pipes
  - ds %>% filter(id > 0) %>% select(id:por_x) %>% mutate(por_x = por_x + 5) %>% arrange(id)
- All dplyr transformations have loads of powerful options that you might want
  - Read the documentation and examples

# Using dplyr helps to avoid inflexible, inefficient code

- Instead of “hard” coding based on position (such as ds[1, 2]), filter and select by name and logical conditions
- Instead of saving multiple subsets of data to calculate summaries, use group_by to summarize within groups
- Instead of typing out long lists of column names and functions, helper functions let you select columns in a variety of ways and apply multiple transformations to selected functions at once

# Data Transformation with dplyr : : CHEAT SHEET

**dplyr** functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

**pipes**

**x %>% f(y)** becomes **f(x, y)**

## Summarise Cases

These apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

**summary function**

**summarise**(.data, …)
Compute table of summaries.
*summarise(mtcars, avg = mean(mpg))*

**count**(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in … Also **tally**().
*count(iris, Species)*

### VARIATIONS

**summarise_all()** - Apply funs to every column.
**summarise_at()** - Apply funs to specific columns.
**summarise_if()** - Apply funs to all cols of one type.

## Group Cases

Use **group_by()** to create a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

```
mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))
```

**group_by**(.data, ..., add = FALSE)
Returns copy of table grouped by …
*g_iris <- group_by(iris, Species)*

**ungroup**(x, …)
Returns ungrouped copy of table.
*ungroup(g_iris)*

## Manipulate Cases

### EXTRACT CASES

Row functions return a subset of rows as a new table.

**filter**(.data, …) Extract rows that meet logical criteria. *filter(iris, Sepal.Length > 7)*

**distinct**(.data, ..., .keep_all = FALSE) Remove rows with duplicate values.
*distinct(iris, Species)*

**sample_frac**(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select fraction of rows.
*sample_frac(iris, 0.5, replace = TRUE)*

**sample_n**(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select size rows. *sample_n(iris, 10, replace = TRUE)*

**slice**(.data, …) Select rows by position.
*slice(iris, 10:15)*

**top_n**(x, n, wt) Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

**Logical and boolean operators to use with filter()**

| | | | | | |
|---|---|---|---|---|---|
| < | <= | is.na() | %in% | \| | xor() |
| > | >= | !is.na() | ! | & | |

See **?base::Logic** and **?Comparison** for help.

### ARRANGE CASES

**arrange**(.data, …) Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

### ADD CASES

**add_row**(.data, ..., .before = NULL, .after = NULL) Add one or more rows to a table.
*add_row(faithful, eruptions = 1, waiting = 1)*

## Manipulate Variables

### EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

**pull**(.data, var = -1) Extract column values as a vector. Choose by name or index.
*pull(iris, Sepal.Length)*

**select**(.data, …)
Extract columns as a table. Also **select_if()**.
*select(iris, Sepal.Length, Species)*

**Use these helpers with select (),**
*e.g. select(iris, starts_with("Sepal"))*

| | | |
|---|---|---|
| **contains**(match) | **num_range**(prefix, range) | **:**, e.g. mpg:cyl |
| **ends_with**(match) | **one_of**(…) | **-**, e.g, -Species |
| **matches**(match) | **starts_with**(match) | |

### MAKE NEW VARIABLES

These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).

**vectorized function**

**mutate**(.data, …)
Compute new column(s).
*mutate(mtcars, gpm = 1/mpg)*

**transmute**(.data, …)
Compute new column(s), drop others.
*transmute(mtcars, gpm = 1/mpg)*

**mutate_all**(.tbl, .funs, …) Apply funs to every column. Use with **funs()**. Also **mutate_if()**.
*mutate_all(faithful, funs(log(.), log2(.)))*
*mutate_if(iris, is.numeric, funs(log(.)))*

**mutate_at**(.tbl, .cols, .funs, …) Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for select().
*mutate_at(iris, vars( -Species), funs(log(.)))*

**add_column**(.data, ..., .before = NULL, .after = NULL) Add new column(s). Also **add_count()**, **add_tally()**. *add_column(mtcars, new = 1:32)*

**rename**(.data, …) Rename columns.
*rename(iris, Length = Sepal.Length)*

# Data wrangling with dplyr package **filter**, **select**, **pull,** and **arrange**

- Subsetting by rows (*filter*) or column (*select*)
  - ds %>% filter(id > 0): select rows id > 0
  - ds %>% select(id): select the column id
  - filter and select return tibbles
- *Pull* grabs values and returns as a vector
  - ds %>% filter(id > 0) %>% pull(id): returns a vector of ids (that are greater than 0)
- *Arrange* sorts by columns

# Data wrangling with dplyr package **rename** and **mutate**

- *Rename* changes column names
  - ds %>% rename(old_column = new_column)
- *Mutate* changes the values in a column
  - ds %>% mutate(id = id + 5)
- *Mutate* also creates new columns
  - ds %>% mutate(id_plus_5 = id + 5)

# Data wrangling with dplyr package **summarize** and **group_by**

- *summarize* collapses data down to a single row
  - ds %>% summarize(mean_porx = mean(por_x))
- *group_by* makes transformations apply within groups (e.g., factors)
  - ds %>% group_by(id, condition) %>% summarize(mean_porx = mean(por_x))

# tidy_select helper functions make it easy to select **columns**

- Powerful, flexible options for finding columns
  - starts_with, ends_with, contains a string
  - -variable (take everything else)
  - c(“var1”, “var2”, “var3”)
  - var1:var4 means from var1 to var4
  - where(is.factor)
- Don’t try to use these to select rows!

# across() is a powerful helper to use with mutate and summarize

- across(column_selection, function_to_apply)
  - summarize(across(var1:var4, mean)) will summarize by taking the mean of var1 to var4
  - saves you from typing: summarize(var1 = mean(var1), var2 = mean(var2), var3 = mean(var3), var4 = mean(var4)
  - can apply multiple functions: summarize(across(starts_with("item"), list(mean = mean, sd = sd))

# Data wrangling tutorial