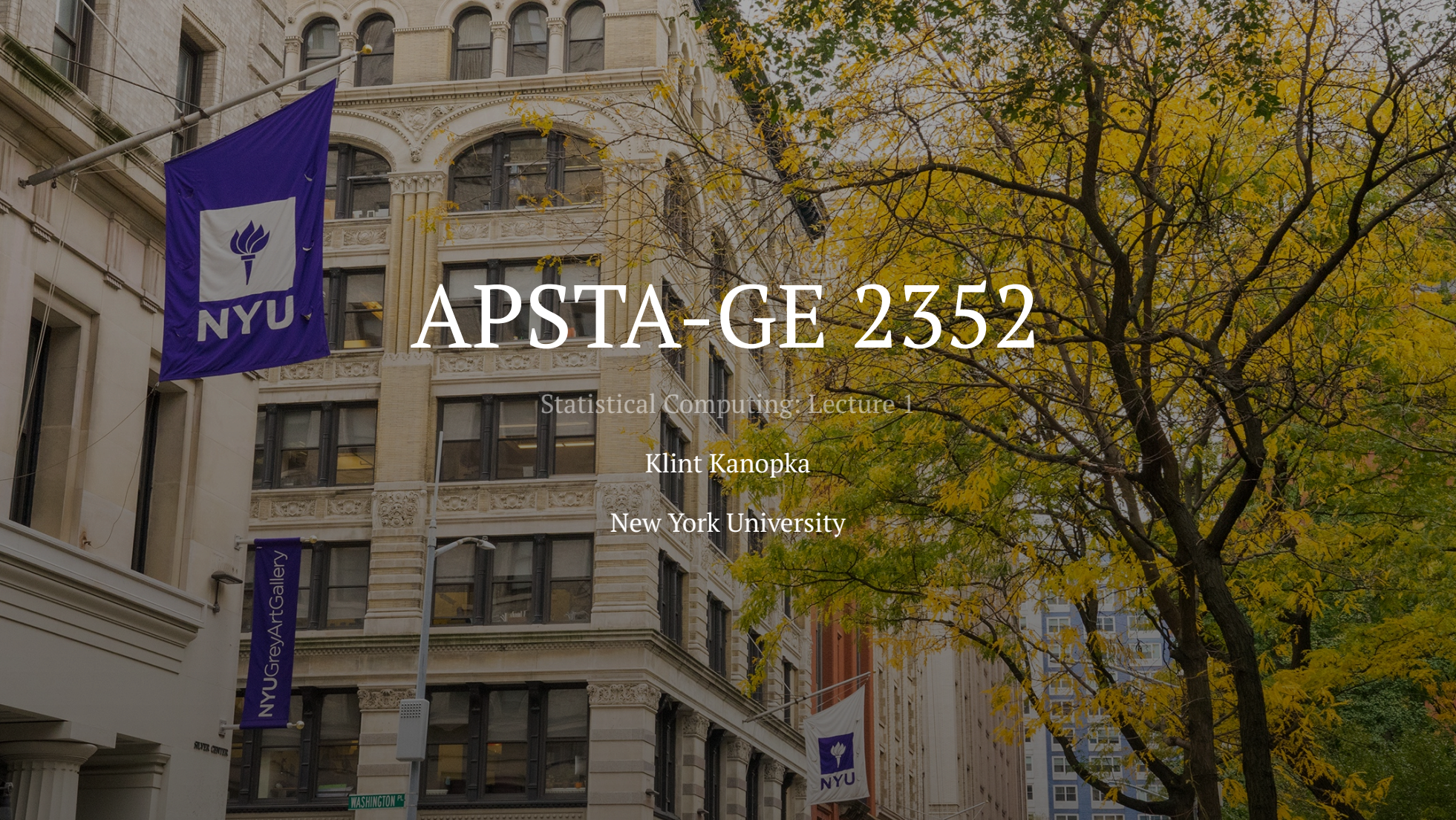


APSTA-GE 2352

Statistical Computing: Lecture 1

Klint Kanopka

New York University

Table of Contents

Welcome

Your Instructor

Important facts about me

- Please call me Klint if you feel comfortable doing so
- Assistant professor of applied statistics
- I really like to drink (black) coffee
- I'm *mostly* a psychometrician, but I have a background in computational physics, machine learning, and AI
- My dog's name is Onions
- Email me: klint.kanopka@nyu.edu
- My office: Kimball Hall, Rm 205W
- Office Hours: T 2-3p @ PRIISM Lobby (Kimball Hall, 2nd Floor)

More About Your Instructor

Less important, but possibly more useful

- I have a lot of idiosyncratic coding preferences and habits
- I'm a dedicated `ggplot()` user
- I *love* the `tidyverse` for cleaning data
- I *hate* the `tidyverse` for absolutely everything else
- My problem sets have a pretty clear "style"
- I am going to be really picky about some things and then not picky at all about basically everything else (I'll try my best to warn you)
- Questions?

Course Business

Course Description

This course will introduce the student to statistical programming and simulation using `R`. Students will first understand variables, data structures, program flow (e.g., conditional execution, looping) and functional programming, then apply these skills to answer interesting statistical questions involving the comparison of groups. Most statistical analysis will be motivated via simulations, rather than mathematical theory. The course content (programming and data analysis) requires significant outside reading and programming.

Some Clarification

- This course is designed to treat `R` as a programming language
- Programming skills are taught through the analysis, design, and implementation of algorithms, with special attention paid to modern statistical algorithms (e.g., gradient descent, k-Means clustering, Markov Chain Monte Carlo)
- Time will also be spent on optimization techniques (e.g., vectorization and parallelization) useful for statistical analysis
- Everything done in this course will be implemented in base `R`, with three exceptions:
  1. `ggplot2` for visualization
  2. The use of `tidyverse` tools for reshaping data so you can feed it into `ggplot()`
  3. Libraries like `MASS` for sampling from distributions not in base `R`
- You will implement algorithms, write unit tests, and debug from scratch using the tools available in base `R`

Even More Clarification

- This course is focused on computing
- You won't learn too much about specifically cleaning or analyzing data here
- You won't even learn too much about statistics here (on purpose)
- You will learn how to be a much better programmer!
- You'll come out knowing a lot more about the tools `R` has to offer and how to solve your own problems if you need something that isn't built in directly
  - This will, by extension, make you better at cleaning and analyzing data!
  - Better at statistics, too! You'll be less reliant on other people's work
- You'll also learn about how statistical algorithms work, when they break, and how to implement and modify them
- I don't believe in paying for books unless you like them, all readings will be distributed as free .pdf files

Prerequisites

This course assumes some experience with the `R` programming language and probability. You may find prior experience with computer science fundamentals and `ggplot2` to be helpful. No previous exposure to the design and analysis of algorithms is assumed.

Student Learning Outcomes

1. Students will implement literate programming to produce coherent and reproducible code.
2. Students will verify code function through the implementation of unit tests.
3. Students will write more efficient code by applying optimization techniques (e.g., vectorization, parallelization).
4. Students will solve problems by implementing and modifying algorithms.
5. Students will answer statistical questions by implementing Monte Carlo simulations.

Meeting Times

- Lecture: Th 4.55-6.35p
- Lab (with Ruiting): W 3.45-4.35p
- Office Hours:
  - Klint: T 2-3p
  - Ruiting: W 9-10a

Course Grades

Category Breakdown

Category weights:

Category	p
Problem Sets	0.7
Final Exam	0.3

- Three-credit course
- Eight equally-weighted problem sets (PS0-PS7)
- Extra credit points are added directly to the point total for problem sets
- The Brightspace "final grade" is unlikely to be correct throughout the year, please compute your own expected final grade to monitor progress

Grading Scale

	G^-	G	G^+
A	$[.895, .945)$	$[.945, 1]$	
B	$[.795, .825)$	$[.825, .865)$	$[.865, .895)$
C	$[.695, .725)$	$[.725, .765)$	$[.765, .795)$
D	$[.600, .640)$	$[.640, .670)$	$[.670, .695)$
F		$[0, .600)$	

Problem Sets

- Released on (or before) Thursday
- Due on Thursdays before lecture @ 4.54p
- PS0 is a one week assignment
- PS1-PS7 are two week assignments
- Submit both a .qmd **and** compiled .pdf files on Brightspace
  - Assignments are distributed as a .qmd template and a compiled .pdf, please use these
  - If your document does not compile from your .qmd, we will not grade your assignment
- Do not call `install.packages()`
- For more specifics, see syllabus

Data types

Data types

- Numeric
  - Numbers with decimals
  - `R` only uses *double precision*
  - Numbers are stored with 64 bits of data
  - 1 bit stores the sign: $\pm$
  - 11 bits store the exponent: between -1022 and $+1023$
  - 52 bits store the significant: between 15-17 significant digits
- Integer
  - Numbers without decimals
  - Can be exactly represented from $-9,007,199,254,740,992$ to $9,007,199,254,740,992$
  - Larger integers are rounded

Data types

- Character
  - Used to store text
  - Also called strings
- Factor
  - Categorical variables
  - Text labels for each possible value called "levels"
  - Factors can be ordered or unordered
  - These can behave weirdly, often best to store data as character strings until you need to do something that requires factors
- Logical
  - Take the values `TRUE` and `FALSE`
  - Can be abbreviated `T` and `F`
  - Under the hood, these take the value `1` and `0`

Data types

- `class()` will tell you what type of thing an object is
- The `as.` functions will cast one object to an object of a different type

```
as.numeric()
as.character()
as.factor()
```

- The `is.` functions ask the question, "is the object this type?"
  - They return a logical object (i.e., `TRUE` or `FALSE`)

```
is.numeric()
is.character()
is.factor()
```

Vectors, lists, and data frames

- Vectors are combinations of objects stored in a single object
  - we make them using the function `c()` (think: *combine*)
  - In `R`, vectors have the property that everything in them must be the same object type
    - This not true of all programming languages
    - This means when you try to combine numbers and strings in a vector, everything is cast to a string
- Lists are *also* combinations of objects stored in a single object
  - Individual elements of a list can be named (and subsequently fished out using `$`)
  - Lists do not require that every object inside them is the same type
- Dataframes are *also* combinations of objects stored in a single object
  - Dataframes are, specifically, combinations of *vectors*
  - Dataframes are *also* lists, so each element can be named
  - Dataframes enforce that every element is the same length

Environments

Environments

- Environments associate (bind) *names* to *values*
- We typically work in the *global environment*
- Environments have *parents*
- Executing a function creates a new environment, with the global environment as the parent
- Loading a package adds that package as the parent of the global environment

Environments

- *Scoping* allows environments to inherit name-value bindings from their parents
  - When you ask for something by name, `R` will look in the current environment first
  - If it can't find the thing, it looks one environment up (function looks to global environment, then to the most recently loaded package, then the package before that, and so on)
  - If two packages have functions of the same name, calling it will use the most recently loaded version
- We can use the syntax `package::function()` to look in a specific environment for an object (like `dplyr::mutate()` or `mirt::fscores()`)

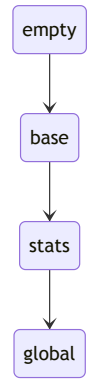
Environments

```
library(tidyverse)
library(ggplot2)
library(psych)

x <- mean(1:10)
y <- rnorm(10)
```

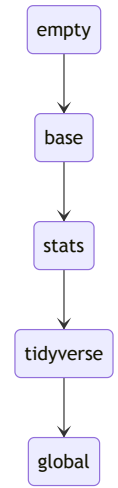
Environments

```
1  library(tidyverse)
2  library(ggplot2)
3  library(psych)
4
5  x <- mean(1:10)
6  y <- rnorm(10)
```



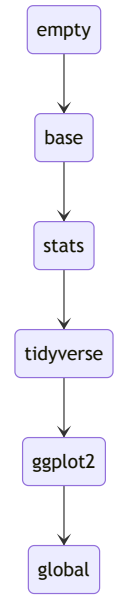
Environments

```
1  library(tidyverse)
2  library(ggplot2)
3  library(psych)
4
5  x <- mean(1:10)
6  y <- rnorm(10)
```



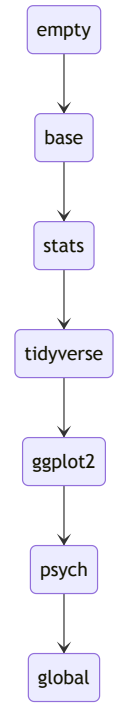
Environments

```
1  library(tidyverse)
2  library(ggplot2)
3  library(psych)
4
5  x <- mean(1:10)
6  y <- rnorm(10)
```



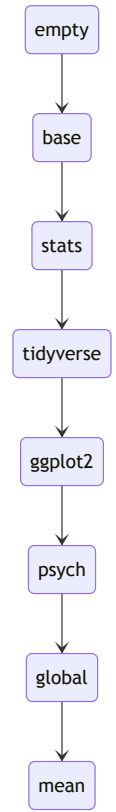
Environments

```
1  library(tidyverse)
2  library(ggplot2)
3  library(psych)
4
5  x <- mean(1:10)
6  y <- rnorm(10)
```



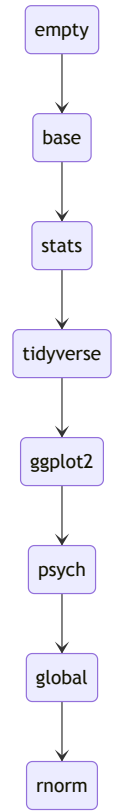
Environments

```
1  library(tidyverse)
2  library(ggplot2)
3  library(psych)
4
5  x <- mean(1:10)
6  y <- rnorm(10)
```



Environments

```
1  library(tidyverse)
2  library(ggplot2)
3  library(psych)
4
5  x <- mean(1:10)
6  y <- rnorm(10)
```



Built-in functions

Built-in functions

- Tons of very useful functions come pre-loaded in the `R` global environment
- These represent the capabilities `R` has without loading other packages
- Some to get you started:

```
mean()     # find the mean of a standard vector
sd()       # find the standard deviation of a numeric vector
median()   # find the median of a numeric vector
sort()     # return a copy of a vector with the elements sorted
library()  # load a package as the parent of the current environment
floor()    # round a number down to the nearest integer
round()    # round a number to a specific decimal place
read.csv() # read a .csv file from disk into memory
```

Random number generation

- `R` (and computers) are horrible at generating truly random numbers
- What they do is produce *pseudorandom* numbers
  - These are deterministic sequences
  - But they have the properties of random sequences when you look at subsets them!
- We want to sample from distributions, and there are four main functions we'll use in this course:

```
rbinom() # draws samples from binomial distributions (i.e., weighted coin flips)
runif()  # draws samples from uniform distributions (i.e., each result is equally likely)
rnorm()  # draws samples from normal distributions (i.e., results closer to the mean are more likely)
sample() # draws samples from a specified object with specified probabilities and with or without replacement
```

- All of these are built-in!

rbinom()

- To read the documentation:

```
?rbinom
```

- Usage:

```
1  out <- rbinom(10, 1, 0.5)
```

rbinom()

- To read the documentation:

```
?rbinom
```

- Usage:

```
out <- rbinom(
  n = 10,
  size = 1,
  prob = 0.5
)
```

rbinom()

- To read the documentation:

```
?rbinom
```

- Usage:

```
out <- rbinom(
  n = 10,      # number of observations
  size = 1,
  prob = 0.5
)
```

rbinom()

- To read the documentation:

```
?rbinom
```

- Usage:

```
out <- rbinom(
  n = 10,      # number of observations
  size = 1,    # number of draws per observation
  prob = 0.5
)
```

rbinom()

- To read the documentation:

```
?rbinom
```

- Usage:

```
out <- rbinom(
  n = 10,      # number of observations
  size = 1,    # number of draws per observation
  prob = 0.5   # probability of success on an individual draw
)
```

`rbinom()`

- To read the documentation:

```
?rbinom
```

- Usage:

```
out <- rbinom(
  n = 10,      # number of observations
  size = 1,    # number of draws per observation
  prob = 0.5   # probability of success on an individual draw
)
```

- Output:
  - A vector of length `n` with each element being the total number of successful draws out of `size`

`runif()`

- To read the documentation:

```
?runif
```

- Usage:

```
1  out <- runif(10)
```

runif()

- To read the documentation:

```
?runif
```

- Usage:

```
1  out <- runif(10, 0, 1)
```

runif()

- To read the documentation:

```
?runif
```

- Usage:

```
out <- runif(
  n = 10,
  min = 0,
  max = 1
)
```

runif()

- To read the documentation:

```
?runif
```

- Usage:

```
out <- runif(
  n = 10,   # number of observations
  min = 0,
  max = 1
)
```

runif()

- To read the documentation:

```
?runif
```

- Usage:

```
out <- runif(
  n = 10,   # number of observations
  min = 0,  # minimum value
  max = 1
)
```

runif()

- To read the documentation:

```
?runif
```

- Usage:

```
out <- runif(
  n = 10,   # number of observations
  min = 0,  # minimum value
  max = 1   # maximum value
)
```

`runif()`

- To read the documentation:

```
?runif
```

- Usage:

```
out <- runif(
  n = 10,   # number of observations
  min = 0,  # minimum value
  max = 1   # maximum value
)
```

- Output:
  - A vector of length `n` with numbers uniformly distributed between `min` and `max`

rnorm()

- To read the documentation:

```
?rnorm
```

- Usage:

```
1  out <- rnorm(10)
```

rnorm()

- To read the documentation:

```
?rnorm
```

- Usage:

```
1  out <- rnorm(10, 0, 1)
```

`rnorm()`

- To read the documentation:

```
?rnorm
```

- Usage:

```
out <- rnorm(
  n = 10,
  mean = 0,
  sd = 1
)
```

rnorm()

- To read the documentation:

```
?rnorm
```

- Usage:

```
out <- rnorm(
  n = 10,     # number of observations
  mean = 0,
  sd = 1
)
```

rnorm()

- To read the documentation:

```
?rnorm
```

- Usage:

```
out <- rnorm(
  n = 10,     # number of observations
  mean = 0,   # mean of the distribution
  sd = 1
)
```

rnorm()

- To read the documentation:

```
?rnorm
```

- Usage:

```
out <- rnorm(
  n = 10,     # number of observations
  mean = 0,   # mean of the distribution
  sd = 1      # standard deviation of the distribution
)
```

`rnorm()`

- To read the documentation:

```
?rnorm
```

- Usage:

```
out <- rnorm(
  n = 10,     # number of observations
  mean = 0,   # mean of the distribution
  sd = 1      # standard deviation of the distribution
)
```

- Output:
  - A vector of length `n` with numbers drawn from a normal distribution with the specified mean and standard deviation

`set.seed()`

- Remember how I said random numbers aren't random?
- These functions produce deterministic sequences!
- `set.seed()` takes a number as input and tells `R` "where to start" when doing things that involve randomness
- Setting a seed allows you to run the same code twice and get the same results both times, even if it's generating random numbers!

Data manipulation with the `tidyverse`

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

What is the `tidyverse`?

- A family of packages designed for "data science"
- Designed to work on dataframes (or `tibbles`)
- Loading the package `tidyverse` is an alias for loading *seven* other packages:

```
library(tidyverse)   # the whole family bundled together

library(dplyr)       # data manipulation
library(tibble)      # an alternative to dataframes
library(ggplot2)     # visualization
library(tidyr)       # pivoting tools
library(stringr)     # string manipulation tools
library(lubridate)   # tools for working with dates as a datatype
library(purrr)       # functional programming tools; map() family
```

`dplyr` verbs

- Built around dataframes in a "tidy" format:
  - Each row is one observation
  - Each column is one variable
  - Each cell contains one single value (numeric or otherwise)
- We'll focus on six core `dplyr` verbs that end up doing most of the heavy lifting:

```
select()     # selects columns in a dataframe
filter()     # selects rows in a dataframe that meet a condition
mutate()     # creates new columns in a dataframe
group_by()   # adds one or more layers of grouping
ungroup()    # removes all layers of grouping
summarize()  # combines multiple rows together, respects grouping
```

- Each of these has a consistent structure
  - The first argument each verb takes is always the dataframe you want to operate on
  - The output is always the dataframe after the operation has been performed

```
output_data <- dplyr_verb(input_data, arg_1, arg_2, arg_3, ...)
```

`select()`

- Read the documentation:

```
library(tidyverse)
?select
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# specify the columns you'd like to keep:
data <- select(raw_data, var_1, var_2, var_3)

# alternatively, specify the columns you'd like to drop:
data <- select(raw_data, -var_4)
```

`select()`

- Read the documentation:

```
library(tidyverse)
?select
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# specify the columns you'd like to keep:
data <- select(raw_data, var_1, var_2, var_3)

# alternatively, specify the columns you'd like to drop:
data <- select(raw_data, -var_4)
```

`select()`

- Read the documentation:

```
library(tidyverse)
?select
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# specify the columns you'd like to keep:
data <- select(raw_data, var_1, var_2, var_3)

# alternatively, specify the columns you'd like to drop:
data <- select(raw_data, -var_4)
```

`select()`

- Read the documentation:

```
library(tidyverse)
?select
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# specify the columns you'd like to keep:
data <- select(raw_data, var_1, var_2, var_3)

# alternatively, specify the columns you'd like to drop:
data <- select(raw_data, -var_4)
```

`select()`

- Read the documentation:

```
library(tidyverse)
?select
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# specify the columns you'd like to keep:
data <- select(raw_data, var_1, var_2, var_3)

# alternatively, specify the columns you'd like to drop:
data <- select(raw_data, -var_4)
```

- Remember:
  - No quotes around variable names
  - There are helper functions you can use to select columns based on conditions
  - The first input and the output are always dataframes

filter()

- Read the documentation:

```
?filter
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# use a logical expression to specify what rows to keep
data <- filter(raw_data, var_1 > 0)

# you can also specify multiple conditions that all must be met
data <- filter(raw_data, var_2 <= 7, var_3 != 0)
```

filter()

- Read the documentation:

```
?filter
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# use a logical expression to specify what rows to keep
data <- filter(raw_data, var_1 > 0)

# you can also specify multiple conditions that all must be met
data <- filter(raw_data, var_2 <= 7, var_3 != 0)
```

filter()

- Read the documentation:

```
?filter
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# use a logical expression to specify what rows to keep
data <- filter(raw_data, var_1 > 0)

# you can also specify multiple conditions that all must be met
data <- filter(raw_data, var_2 <= 7, var_3 != 0)
```

filter()

- Read the documentation:

```
?filter
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# use a logical expression to specify what rows to keep
data <- filter(raw_data, var_1 > 0)

# you can also specify multiple conditions that all must be met
data <- filter(raw_data, var_2 <= 7, var_3 != 0)
```

`filter()`

- Read the documentation:

```
?filter
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# use a logical expression to specify what rows to keep
data <- filter(raw_data, var_1 > 0)

# you can also specify multiple conditions that all must be met
data <- filter(raw_data, var_2 <= 7, var_3 != 0)
```

- Remember:
  - Still no quotes around variable names
  - Multiple conditions can be in one `filter()` call or spread out across multiple calls
  - Rows where the condition is `TRUE` are kept, rows where the condition is `FALSE` are dropped
  - The first input and the output are always dataframes

mutate()

- Read the documentation:

```
?mutate
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create a new variable name and specify what it should be
data <- mutate(raw_data, mean_vars = (var_1 + var_2 + var_3)/3)

# you can also create multiple new variables at once
data <- mutate(
    raw_data,
    max_var = max(var_1, var_2, var_3),
    min_var = min(var_1, var_2, var_3)
    )
```

mutate()

- Read the documentation:

```
?mutate
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create a new variable name and specify what it should be
data <- mutate(raw_data, mean_vars = (var_1 + var_2 + var_3)/3)

# you can also create multiple new variables at once
data <- mutate(
    raw_data,
    max_var = max(var_1, var_2, var_3),
    min_var = min(var_1, var_2, var_3)
    )
```

mutate()

- Read the documentation:

```
?mutate
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create a new variable name and specify what it should be
data <- mutate(raw_data, mean_vars = (var_1 + var_2 + var_3)/3)

# you can also create multiple new variables at once
data <- mutate(
    raw_data,
    max_var = max(var_1, var_2, var_3),
    min_var = min(var_1, var_2, var_3)
    )
```

mutate()

- Read the documentation:

```
?mutate
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create a new variable name and specify what it should be
data <- mutate(raw_data, mean_vars = (var_1 + var_2 + var_3)/3)

# you can also create multiple new variables at once
data <- mutate(
    raw_data,
    max_var = max(var_1, var_2, var_3),
    min_var = min(var_1, var_2, var_3)
    )
```

mutate()

- Read the documentation:

```
?mutate
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create a new variable name and specify what it should be
data <- mutate(raw_data, mean_vars = (var_1 + var_2 + var_3)/3)

# you can also create multiple new variables at once
data <- mutate(
    raw_data,
    max_var = max(var_1, var_2, var_3),
    min_var = min(var_1, var_2, var_3)
    )
```

- Remember:
  - Adds a new column to the dataframe, computing a value for each row; existing data left untouched
  - The first input and the output are always dataframes

`group_by()` and `ungroup()`

- Read the documentation:

```
?group_by
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create groups of rows that have the same value for var_1
data <- group_by(raw_data, var_1)

# specifying multiple grouping variables
data <- group_by(raw_data, var_2, var_3)

# remove grouping
data <- ungroup(data)
```

`group_by()` and `ungroup()`

- Read the documentation:

```
?group_by
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create groups of rows that have the same value for var_1
data <- group_by(raw_data, var_1)

# specifying multiple grouping variables
data <- group_by(raw_data, var_2, var_3)

# remove grouping
data <- ungroup(data)
```

`group_by()` and `ungroup()`

- Read the documentation:

```
?group_by
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create groups of rows that have the same value for var_1
data <- group_by(raw_data, var_1)

# specifying multiple grouping variables
data <- group_by(raw_data, var_2, var_3)

# remove grouping
data <- ungroup(data)
```

`group_by()` and `ungroup()`

- Read the documentation:

```
?group_by
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create groups of rows that have the same value for var_1
data <- group_by(raw_data, var_1)

# specifying multiple grouping variables
data <- group_by(raw_data, var_2, var_3)

# remove grouping
data <- ungroup(data)
```

`group_by()` and `ungroup()`

- Read the documentation:

```
?group_by
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create groups of rows that have the same value for var_1
data <- group_by(raw_data, var_1)

# specifying multiple grouping variables
data <- group_by(raw_data, var_2, var_3)

# remove grouping
data <- ungroup(data)
```

`group_by()` and `ungroup()`

- Read the documentation:

```
?group_by
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# create groups of rows that have the same value for var_1
data <- group_by(raw_data, var_1)

# specifying multiple grouping variables
data <- group_by(raw_data, var_2, var_3)

# remove grouping
data <- ungroup(data)
```

- Remember:
  - Multiple grouping variables creates groups for each unique combination of values across those variables
  - `ungroup()` removes **all** grouping variables
  - The first input and the output are always dataframes

summarize()

- Read the documentation:

```
?summarize
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# most useful on grouped dataframes!
data <- group_by(raw_data, var_1)

# works like mutate, except it destroys all data except the grouping variables!
data <- summarize(data, mean_var_2 = mean(var_2), sd_var_2 = sd(var_2))

# don't forget to remove grouping if you don't want it anymore!
data <- ungroup(data)

# final data frame will have three columns: var_1, mean_var_2, sd_var_2 and one row per unique value of var_1
```

summarize()

- Read the documentation:

```
?summarize
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# most useful on grouped dataframes!
data <- group_by(raw_data, var_1)

# works like mutate, except it destroys all data except the grouping variables!
data <- summarize(data, mean_var_2 = mean(var_2), sd_var_2 = sd(var_2))

# don't forget to remove grouping if you don't want it anymore!
data <- ungroup(data)

# final data frame will have three columns: var_1, mean_var_2, sd_var_2 and one row per unique value of var_1
```

summarize()

- Read the documentation:

```
?summarize
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# most useful on grouped dataframes!
data <- group_by(raw_data, var_1)

# works like mutate, except it destroys all data except the grouping variables!
data <- summarize(data, mean_var_2 = mean(var_2), sd_var_2 = sd(var_2))

# don't forget to remove grouping if you don't want it anymore!
data <- ungroup(data)

# final data frame will have three columns: var_1, mean_var_2, sd_var_2 and one row per unique value of var_1
```

summarize()

- Read the documentation:

```
?summarize
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# most useful on grouped dataframes!
data <- group_by(raw_data, var_1)

# works like mutate, except it destroys all data except the grouping variables!
data <- summarize(data, mean_var_2 = mean(var_2), sd_var_2 = sd(var_2))

# don't forget to remove grouping if you don't want it anymore!
data <- ungroup(data)

# final data frame will have three columns: var_1, mean_var_2, sd_var_2 and one row per unique value of var_1
```

summarize()

- Read the documentation:

```
?summarize
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# most useful on grouped dataframes!
data <- group_by(raw_data, var_1)

# works like mutate, except it destroys all data except the grouping variables!
data <- summarize(data, mean_var_2 = mean(var_2), sd_var_2 = sd(var_2))

# don't forget to remove grouping if you don't want it anymore!
data <- ungroup(data)

# final data frame will have three columns: var_1, mean_var_2, sd_var_2 and one row per unique value of var_1
```

summarize()

- Read the documentation:

```
?summarize
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# most useful on grouped dataframes!
data <- group_by(raw_data, var_1)

# works like mutate, except it destroys all data except the grouping variables!
data <- summarize(data, mean_var_2 = mean(var_2), sd_var_2 = sd(var_2))

# don't forget to remove grouping if you don't want it anymore!
data <- ungroup(data)

# final data frame will have three columns: var_1, mean_var_2, sd_var_2 and one row per unique value of var_1
```

`summarize()`

- Read the documentation:

```
?summarize
```

- Usage:

```
raw_data <- read_csv('./file_path/data.csv')

# most useful on grouped dataframes!
data <- group_by(raw_data, var_1)

# works like mutate, except it destroys all data except the grouping variables!
data <- summarize(data, mean_var_2 = mean(var_2), sd_var_2 = sd(var_2))

# don't forget to remove grouping if you don't want it anymore!
data <- ungroup(data)

# final data frame will have three columns: var_1, mean_var_2, sd_var_2 and one row per unique value of var_1
```

- Remember:
  - `summarize()` combines rows together and is most often used on grouped dataframes
  - The first input and the output are always dataframes

The pipe

- Did you notice how all of the first inputs and outputs are always dataframes?
- Tidyverse functions are built to be connected with *pipes*
- There are two pipes in `R`
  - `%>%` is from a package called `magrittr`
  - `|>`, added later, is native `R` syntax
  - The native `R` pipe is faster and should be preferred
  - That said, the `magrittr` pipe has more features (I don't use them)
- The pipe takes the output on the left hand side and puts it into the first argument of the function on the right hand side

```
out <- mean(x)
```

The pipe

- Did you notice how all of the first inputs and outputs are always dataframes?
- Tidyverse functions are built to be connected with *pipes*
- There are two pipes in `R`
  - `%>%` is from a package called `magrittr`
  - `|>`, added later, is native `R` syntax
  - The native `R` pipe is faster and should be preferred
  - That said, the `magrittr` pipe has more features (I don't use them)
- The pipe takes the output on the left hand side and puts it into the first argument of the function on the right hand side

```
out <- x |>
    mean()
```

The pipe

- Did you notice how all of the first inputs and outputs are always dataframes?
- Tidyverse functions are built to be connected with *pipes*
- There are two pipes in `R`
  - `%>%` is from a package called `magrittr`
  - `|>`, added later, is native `R` syntax
  - The native `R` pipe is faster and should be preferred
  - That said, the `magrittr` pipe has more features (I don't use them)
- The pipe takes the output on the left hand side and puts it into the first argument of the function on the right hand side

```
out <- x |>
    mean()
```

- Okay, but why is this any good?

Rewriting `dplyr` code with pipes

```r
out <- read_csv('./file_path/data.csv')
out <- filter(out, var_3 != 0)
out <- group_by(out, var_1)
out <- summarize(out, mean_var = mean(var_2), sd_var = sd(var_2))
out <- ungroup(out)
out <- mutate(out, ci_lower = mean_var - 1.96*sd_var, ci_upper = mean_var + 1.96*sd_var)
```

Rewriting `dplyr` code with pipes

```
out <- read_csv('./file_path/data.csv') |>
    filter(var_3 != 0)
out <- group_by(out, var_1)
out <- summarize(out, mean_var = mean(var_2), sd_var = sd(var_2))
out <- ungroup(out)
out <- mutate(out, ci_lower = mean_var - 1.96*sd_var, ci_upper = mean_var + 1.96*sd_var)
```

Rewriting `dplyr` code with pipes

```
out <- read_csv('./file_path/data.csv') |>
    filter(var_3 != 0) |>
    group_by(var_1)
out <- summarize(out, mean_var = mean(var_2), sd_var = sd(var_2))
out <- ungroup(out)
out <- mutate(out, ci_lower = mean_var - 1.96*sd_var, ci_upper = mean_var + 1.96*sd_var)
```

Rewriting `dplyr` code with pipes

```
out <- read_csv('./file_path/data.csv') |>
    filter(var_3 != 0) |>
    group_by(var_1) |>
    summarize(mean_var = mean(var_2),
              sd_var = sd(var_2))
out <- ungroup(out)
out <- mutate(out, ci_lower = mean_var - 1.96*sd_var, ci_upper = mean_var + 1.96*sd_var)
```

Rewriting `dplyr` code with pipes

```r
out <- read_csv('./file_path/data.csv') |>
    filter(var_3 != 0) |>
    group_by(var_1) |>
    summarize(mean_var = mean(var_2),
              sd_var = sd(var_2)) |>
    ungroup()
out <- mutate(out, ci_lower = mean_var - 1.96*sd_var, ci_upper = mean_var + 1.96*sd_var)
```

Rewriting `dplyr` code with pipes

```
out <- read_csv('./file_path/data.csv') |>
    filter(var_3 != 0) |>
    group_by(var_1) |>
    summarize(mean_var = mean(var_2),
              sd_var = sd(var_2)) |>
    ungroup() |>
    mutate(ci_lower = mean_var - 1.96*sd_var,
           ci_upper = mean_var + 1.96*sd_var)
```

Rewriting `dplyr` code with pipes

```
out <- read_csv('./file_path/data.csv') |>
    filter(var_3 != 0) |>
    group_by(var_1) |>
    summarize(mean_var = mean(var_2),
              sd_var = sd(var_2)) |>
    ungroup() |>
    mutate(ci_lower = mean_var - 1.96*sd_var,
           ci_upper = mean_var + 1.96*sd_var)
```

- What you get is more readable code with fewer intermediate objects stored!

Long data and wide data

- We often think of dataframes being structured like spreadsheets
  - One row per individual, one column per variable
  - This is called *wide* data
- There is another way!
  - Multiple rows per individual, with each row associated with a specific variable or measurement!
  - One column for an `id` variable
  - One column to identify the measurement
  - One column to store the value of that measurement
  - This is called *long* data

Two versions of the same data

Wide

id	test_1	test_2
A	95	99
B	86	92
C	90	84

Long

id	test	score
A	1	95
A	2	99
B	1	86
B	2	92
C	1	90
C	2	84

`pivot_longer()`

- The job of `pivot_longer()` is to turn data from a wider form into a longer form!
- To read the documentation:

```
?pivot_longer
```

- Usage:

```
1  data_long <- pivot_longer()
```

`pivot_longer()`

- The job of `pivot_longer()` is to turn data from a wider form into a longer form!
- To read the documentation:

```
?pivot_longer
```

- Usage:

```
data_long <- pivot_longer(
  data,
  cols,
  names_to,
  values_to
)
```

`pivot_longer()`

- The job of `pivot_longer()` is to turn data from a wider form into a longer form!
- To read the documentation:

```
?pivot_longer
```

- Usage:

```
data_long <- pivot_longer(
  data = data_wide,         # the data you want to pivot
  cols,
  names_to,
  values_to
)
```

`pivot_longer()`

- The job of `pivot_longer()` is to turn data from a wider form into a longer form!
- To read the documentation:

```
?pivot_longer
```

- Usage:

```
data_long <- pivot_longer(
  data = data_wide,          # the data you want to pivot
  cols = c(test_1, test_2), # the (unquoted) columns you want to make longer
  names_to,
  values_to
)
```

`pivot_longer()`

- The job of `pivot_longer()` is to turn data from a wider form into a longer form!
- To read the documentation:

```
?pivot_longer
```

- Usage:

```
data_long <- pivot_longer(
  data = data_wide,          # the data you want to pivot
  cols = c(test_1, test_2), # the (unquoted) columns you want to make longer
  names_to = 'test',         # the name of the new column that identifies each measurement
  values_to
)
```

`pivot_longer()`

- The job of `pivot_longer()` is to turn data from a wider form into a longer form!
- To read the documentation:

```
?pivot_longer
```

- Usage:

```
data_long <- pivot_longer(
  data = data_wide,          # the data you want to pivot
  cols = c(test_1, test_2), # the (unquoted) columns you want to make longer
  names_to = 'test',         # the name of the new column that identifies each measurement
  values_to = 'score'        # the name of the new column that contains the values of each measurement
)
```

`pivot_wider()`

- The job of `pivot_longer()` is to turn data from a longer form into a wider form!
- To read the documentation:

```
?pivot_wider
```

- Usage:

```
1  data_wide <- pivot_wider()
```

`pivot_wider()`

- The job of `pivot_longer()` is to turn data from a longer form into a wider form!
- To read the documentation:

```
?pivot_wider
```

- Usage:

```
data_wide <- pivot_wider(
  data,
  id_cols,
  names_from,
  values_from
)
```

`pivot_wider()`

- The job of `pivot_longer()` is to turn data from a longer form into a wider form!
- To read the documentation:

```
?pivot_wider
```

- Usage:

```
data_wide <- pivot_wider(
  data = data_long,    # the data you want to pivot
  id_cols,
  names_from,
  values_from
)
```

`pivot_wider()`

- The job of `pivot_longer()` is to turn data from a longer form into a wider form!
- To read the documentation:

```
?pivot_wider
```

- Usage:

```
data_wide <- pivot_wider(
  data = data_long,     # the data you want to pivot
  id_cols = id,         # the column(s) that uniquely identify which row each measurement belongs to
  names_from,
  values_from
)
```

`pivot_wider()`

- The job of `pivot_longer()` is to turn data from a longer form into a wider form!
- To read the documentation:

```
?pivot_wider
```

- Usage:

```
data_wide <- pivot_wider(
  data = data_long,     # the data you want to pivot
  id_cols = id,         # the column(s) that uniquely identify which row each measurement belongs to
  names_from = test,    # the column that identifies the measurements
  values_from
)
```

`pivot_wider()`

- The job of `pivot_longer()` is to turn data from a longer form into a wider form!
- To read the documentation:

```
?pivot_wider
```

- Usage:

```
data_wide <- pivot_wider(
  data = data_long,     # the data you want to pivot
  id_cols = id,         # the column(s) that uniquely identify which row each measurement belongs to
  names_from = test,    # the column that identifies the measurements
  values_from = score   # the column that contains the values of the measurements
)
```

`pivot_wider()`

- The job of `pivot_longer()` is to turn data from a longer form into a wider form!
- To read the documentation:

```
?pivot_wider
```

- Usage:

```
data_wide <- pivot_wider(
  data = data_long,        # the data you want to pivot
  id_cols = id,            # the column(s) that uniquely identify which row each measurement belongs to
  names_from = test,       # the column that identifies the measurements
  values_from = score,     # the column that contains the values of the measurements
  names_prefix = 'test_'   # a prefix that goes in front of the new column names
)
```

Data visualization with `ggplot2`

The grammar of graphics

- Grammar is the syntax we used to combine words into sentences to express ideas
- Plots convey ideas with their own “grammatical structure”
  - Layers
    - Data
    - Mappings
    - Statistical transformations
    - Geometric objects
    - Position Adjustments
  - Scales
    - Numerical scales, color scales, etc
  - Coordinate Systems
    - Cartesian, Polar, etc
  - Facets
    - Panels

Layers

- Have five parts:
  1. Data (nouns, the subject of our plot)
     - The data we want to describe
  2. Mappings (verbs, the actions that assign our data to scales)
     - How we assign data to different scales
     - x-axis, y-axis, color, fill, size, transparency, etc.
  3. Statistical transformations (adjectives, modify data)
     - Log transforms, power transforms, adding variables together, etc
  4. Geometric objects (objects - mappings project data to scales)
     - Scatter plots, bars, lines, etc
  5. Position Adjustments (adjectives, modify objects)
     - Nudges, jitters, etc
- Added to plots using `geom_xxx()` functions
- Inherit data and mappings from the original `ggplot()` call

Scales

- Scales determine how data are transformed into visual elements
  - They can be used to translate values to colors
  - They can be used to translate values to numeric scales for positions
- Added using `scale_A_B()` calls
  - `A` is the aspect you're scaling (x, y, color, fill)
  - `B` is how you're scaling it (manually, `log10`, or something else)

Coordinate Systems

- Often not specified!
- The default is cartesian (an x, y plane)
- Could also be polar, 3d, or other specialized stuff
- Fancy coordinate systems are often harder to understand for readers

Facets

- Facets allow you to create different panels so you can compare similar plots of subsets of data side-by-side
- Added using `facet_grid()` or similar
- Facets share all the same layers, scales, and coordinate systems

What does `ggplot()` want from us?

- "Long form data"
  - Every variable you want to map to a scale has to have its own column
  - Makes every item drawn in the plot in its own row
- We often have "wide data"
  - Multiple observations are in the same row
  - Example: tracking steps for an office walking contest
    - Each row is a person
    - First column is their name
    - Each subsequent column is the number of steps they took on a given day
  - If you wanted to make line plots with a different line for each person, `ggplot()` can't handle this format!
    - It wants three columns: `name, day, steps`
    - You need to use `pivot_longer()`!
    - If you collect data for ten days, each person has ten rows

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
1 ggplot()
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
1 ggplot(data, aes())
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
ggplot(data, aes(x =, y =, color =))
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
1  ggplot(data, aes(x = day, y =, color =))
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
ggplot(data, aes(x = day, y = steps, color =))
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
1  ggplot(data, aes(x = day, y = steps, color = name))
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
ggplot(data, aes(x = day, y = steps, color = name)) +
  geom_line()
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
ggplot(data, aes(x = day, y = steps, color = name)) +
  geom_line() +
  scale_color_manual()
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
ggplot(data, aes(x = day, y = steps, color = name)) +
  geom_line() +
  scale_color_manual(values =)
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
ggplot(data, aes(x = day, y = steps, color = name)) +
  geom_line() +
  scale_color_manual(values = c('klint' = 'green',
                                'ravi' = 'blue',
                                'daphna' = 'red',
                                'alex' = 'purple')
```

Building a plot using `ggplot()`

- Recall the office walking contest:
  - We have a dataframe called `data`
  - In long form with one row per person-day
  - Three columns: `name, day, steps`

```
ggplot(data, aes(x = day, y = steps, color = name)) +
  geom_line() +
  scale_color_manual(values = c('klint' = 'green',
                                'ravi' = 'blue',
                                'daphna' = 'red',
                                'alex' = 'purple') +
  theme_minimal()
```

Wrapping up

- We covered a lot of kinda bland introductory material, with not a lot of context!
- Your lab on Wednesday is devoted to practicing `tidyverse` and `ggplot2`
- Lab is also a place where you'll go over homework-related problems and can ask questions. Use Ruiting as a resource!
- Ask questions on Slack!
- PS0 is posted and due before class next week
- PS1 will post tonight or tomorrow and be due in two weeks