STA 141A: Fundamentals of Statistical Data Science

Course Material Summary *Part 1*

University of California at Davis

Last Edit Date: 06/23/2022

STA 141A Summary Part 1

I. R Basics

1. Working directory

```
getwd()
setwd("/path to working directory")
```

2. Arithmetic Operations

+ (addition), - (subtraction), * (multiplication), / (division), %% (remainder), %/% (integer division), ^ (exponent), log(x, base), factorial(x), exp(x), sin(), cos(), tan(), sqrt(), max(), min(), sum(), mean(), median(), quantile(x, prob), round(x, n), rank(), signif(x, n), var(), sd(), cor().

3. Assign value

We can use "=" or "<-" to assign value to a variable.

4. Print variable

```
print(x)
```

```
Output: 1
```

5. Print and assign

```
(x <- 2)
```

```
Output: 2
```

6. Workspace functions

Use ls() or objects() to show all the objects in the workspace.

```
ls()
objects()
```

rm(y, z) removes variables y, z, rm(list=ls()) removes everything.

```
rm(y,z)
rm(list=ls())
```

7. R is case sensitive.

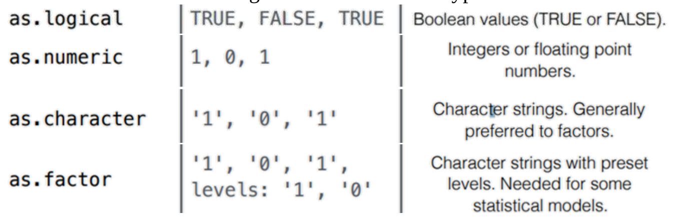
II. Vectors

1. Create vectors

`c(2,4,6)`	`2 4 6`	Join elements into a vector
`2:6`	`2 3 4 5 6`	An integer sequence
`seq(2,3,by=0.5)`	`2.0 2.5 3.0`	A complex sequence
`rep(1:2, times=3)`	`1 2 1 2 1 2`	Repeat a vector
`rep(1:2, each=3)`	`1 1 1 2 2 2`	Repeat elements of a vector

2. Types

We can use class() or mode() to check the type of a vector. Note that a vector has to have elements that are the same mode. The vector is coerced to the most flexible type. Character is the most flexible.
We can also use the following command to convert types:



`as.logical`	`TRUE, FALSE, TRUE`	Boolean values (TRUE or FALSE).
`as.numeric`	`1, 0, 1`	Integers or floating point numbers.
`as.character`	`'1', '0', '1'`	Character strings. Generally preferred to factors.
`as.factor`	`'1', '0', '1', levels: '1', '0'`	Character strings with preset levels. Needed for some statistical models.

3. Vector structure

```
length(x) #return the length of the vector x
head(x) #return the first 6 elements of vector x
head(x, n) #return the first n elements of vector x
tail(x) #return the last 6 elements of vector x
tail(x, n) #return the last n elements of vector x
```

4. Sort vector

```
sort(x) #sort increasingly
sort(x, decreasing = TRUE) #sort decreasingly
```

5. Sampling

```
sample(x, size, replace = T/F)
```

6. Recycle

Shorter vectors are recycled.

```
z = 1:8
x = 1:4
x+z
```

```
Output:
2 4 6 8 6 8 10 12
```

7. Logical vectors

OR: |

```
x <- 11:20
x<15|x>18
```

```
Output: TRUE TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE TRUE TRUE
```

AND: &

```
x<18&x>15
```

```
Output: FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE
```

NOT: !

```
x != 15
```

```
Output: TRUE  TRUE  TRUE  TRUE FALSE TRUE TRUE  TRUE  TRUE  TRUE
```

```
!(x<18&x>15)
```

```
Output: TRUE  TRUE  TRUE  TRUE FALSE TRUE TRUE  TRUE  TRUE  TRUE
```

%in%: This operator is used to identify if an element belongs to a vector.

```
15 %in% x
```

```
Output: TRUE
```

8. Searching for elements in the vector

which takes a logical vector, and returns an integer vector of positions, the position of the element/elements that satisfy a condition.

```
which(x<16|x>18)
```

```
Output: 1  2  3  4  5  9 10
```

```
which(x<16)
```

```
Output: 1 2 3 4 5
```

9. Calculate sum and mean of logical vector

```
sum(x<15)
```

```
Output: 4
```

```
mean(x<15)
```

```
Output: 0.4
```

10. Subset

1) Subset using a logical vector

```
x <- c(1,2,3,4,5)
x[c(TRUE, TRUE, FALSE, FALSE, TRUE)] #Output: 1,2,5
x[x<3] #Output: 1,2
x[x<3|x>4] #Output: 1,2,5
```

2) Subset using elements position

```
x <- c(1,2,3,4,5)
x[2] #Output: 2
x[-2] #Output: 1,3,4,5
x[c(1,5)] #Output: 1,5
x[-c(1,5)] #Output: 2,3,4
```

3) Subset using the name

```
x <- c(time1 = 2.5, time2 = 3.2)
x['time1'] #Output: ## time1 ## 2.5
```

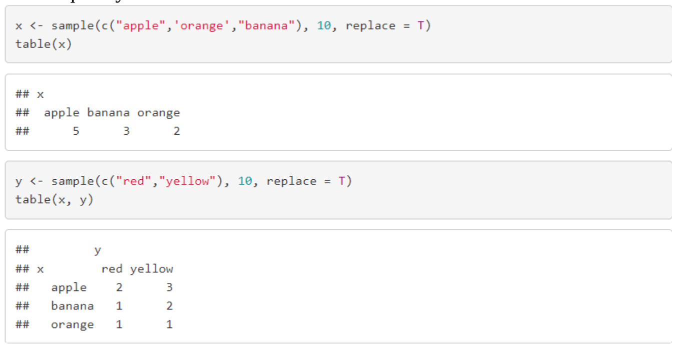
11. Equality of vectors

```
x <- 1:3
y <- c(1,2,4)
x==y #Output: TRUE  TRUE  FALSE
x[x==y] #Output: 1 2
```

12. Removing NA

```
x<-c(1,2,NA,10,7)
sum(x,na.rm=TRUE) #Output: 20
mean(x, na.rm=TRUE) #Output: 5
is.na(x) #Output: FALSE FALSE TRUE FALSE FALSE
x[!is.na(x)] #Output: 1 2 10 7
```

13. Frequency table



```
x <- sample(c("apple",'orange',"banana"), 10, replace = T)
table(x)
```

```
## x
##  apple banana orange
##      5      3      2
```

```
y <- sample(c("red","yellow"), 10, replace = T)
table(x, y)
```

```
##         y
## x         red yellow
##   apple    2      3
##   banana   1      2
##   orange   1      1
```

14. Factor

A factor is created using the factor() function.

```
education <- c('low', 'mid', 'high', 'high', 'mid')
educationf <- factor(education)
```

```
## [1] low  mid  high high mid
## Levels: high low mid
```

To simply print the levels of a factor the function levels() can be used. Factors can be useful to “filter out” typos or unacceptable values.

III. Matrix

1. Matrix Basics

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)
ncol() #shows the number of columns
nrow() #shows the number of rows
dim() #shows nrow first, ncol second
cbind() #add new vector of the same length of row
rbind() #add new vector of the same length of col
attributes() #shows the dimension
colnames() #add colnames
rownames() #add rownames
```

When adding two matrices that have the same dimension, they will add each other in the same position. Same for element-wise multiplication.

2. Matrix operations

```
t(A)%*%B # Matrix multiplication and transpose
sum(A)# add all elements
prod(A)
colSums(A) # sum columns
rowSums(A) # sum rows
```

3. Deal with NA

```
colSums(A, na.rm =T)# sum cols
rowSums(A, na.rm=T)# sum rows
colMeans(A, na.rm = T)# mean columns
rowMeans(A, na.rm = T)# mean rows
```

4. Using apply

```
apply(M, 1, sum, na.rm=T) # row sum without NA
apply(M, 2, sum, na.rm=T) # col sum without NA
```

5. Subset

Using the elements' position

```
M[n] # the n-th number in the matrix
M[n,] # the n-th row of the matrix
M[,n] # the n-th col of the matrix
M[n,m] # the n-th row, m-th col of the matrix
```

Using a logical vector

```
M[c(TRUE, FALSE, FALSE),c(TRUE, FALSE, FALSE)]
```

Using column names or row names

```
colnames(A) <- c("First","Second","Third")
rownames(A) <- c("a","b","C")
A["b","First"]
A[2,"First"]
```

6. Filtering

```
A <- matrix(100:105, ncol = 3, nrow = 2)
which(A==105) # return the index of element is 105
which(A==105,arr.ind=TRUE) # return 105 row and col
```

IV. List

1. List Basics

```
lst <- list(name="Fred", wife="Mary",
            no.children=3,
            child.ages=c(4,7,9), resident=TRUE)
str(lst)
```

```
## List of 5
##  $ name       : chr "Fred"
##  $ wife       : chr "Mary"
##  $ no.children: num 3
##  $ child.ages : num [1:3] 4 7 9
##  $ resident   : logi TRUE
```

```
class(lst)
```

```
## [1] "list"
```

```
attributes(lst)
```

```
## $names
## [1] "name"        "wife"        "no.children" "child.ages"  "resident"
```

2. Subset

```
class(lst[[1]])
```

```
## [1] "character"
```

```
class(lst[1])
```

```
## [1] "list"
```

```
lst[["name"]]
```

```
## [1] "Fred"
```

```
lst[["child.ages"]]
```

```
## [1] 4 7 9
```

3. Extract the first element of the fourth element of the list

```
lst[[4]][1]
```

```
## [1] 4
```

4. Select a sublist

```
lst[c(1,2)]
```

```
## $name
## [1] "Fred"
##
## $wife
## [1] "Mary"
```

```
lst[c("name","wife")]
```

```
## $name
## [1] "Fred"
##
## $wife
## [1] "Mary"
```

V. Arrays

1. One vector argument to describe all dimensions

```
A<-matrix(1:6,2,3)
B<-array(1:18,c(2,3,3))
C<-array(2,c(2,3,3))
dim(B)
```

```
## [1] 2 3 3
```

2. Subseting an array is like subseting a matrix but you will have more dimensions to specify

```
B[1,,]
```

```
##      [,1] [,2] [,3]
## [1,]    1    7   13
## [2,]    3    9   15
## [3,]    5   11   17
```

3. Select the last row of the third matrix

```
B[,,3][2,]
```

```
## [1] 14 16 18
```

4. Select the first element of the third matrix

```
B[,,3][1,1]
```

```
## [1] 13
```

VI. apply(), lapply(), sapply(), and tapply()

1. apply()

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

X: an array, including a matrix.
MARGIN: a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.

FUN: the function to be applied: see ‘Details’. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted.
...: optional arguments to FUN.
simplify: a logical indicating whether results should be simplified if possible.

2. lapply()

```
lapply(X, FUN, ...)
```

lapply returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

```
lst <- list(group1 = sample(seq(0, 10, 0.1), 10),
            group2 = sample(seq(5, 10, 0.1), 10),
            group3 = sample(seq(10, 15, 0.1), 10))
lapply(lst, mean)
```

```
## $group1
## [1] 4.56
##
## $group2
## [1] 7.44
##
## $group3
## [1] 12.87
```

3. sapply()

```
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

USE.NAMES: logical; if TRUE and if X is character, use X as names for the result unless it had names already. Since this argument follows ... its name cannot be abbreviated.
sapply is a user-friendly version and wrapper of lapply by default returning a vector, matrix or, if simplify = "array", an array if appropriate, by applying simplify2array(). sapply(x, f, simplify = FALSE, USE.NAMES = FALSE) is the same as lapply(x, f).

```
sapply(lst, mean)
```

```
## group1 group2 group3
##   4.56   7.44  12.87
```

4. tapply()

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify
= TRUE)
```

```
tapply(lst[[1]], lst[[2]], mean)
```

```
## 5.1 5.7 6.2 6.7 7.2   8 8.2 8.8 8.9 9.6
## 9.9 0.5 1.5 4.5 7.0 3.6 5.9 5.6 0.6 6.5
```

VII. Splitting according to a factor

```
str(ToothGrowth)
```

```
## 'data.frame':    60 obs. of  3 variables:
##  $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
##  $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
##  $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

```
# split() creates a list, in this case it separates the vector len into subgroups
# based on the variable dose and stores the new split vectors into a list
split(ToothGrowth$len,ToothGrowth$dose)
```

```
## $`0.5`
##  [1]  4.2 11.5  7.3  5.8  6.4 10.0 11.2 11.2  5.2  7.0 15.2 21.5 17.6  9.7 14.5
## [16] 10.0  8.2  9.4 16.5  9.7
##
## $`1`
##  [1] 16.5 16.5 15.2 17.3 22.5 17.3 13.6 14.5 18.8 15.5 19.7 23.3 23.6 26.4 20.0
## [16] 25.2 25.8 21.2 14.5 27.3
##
## $`2`
##  [1] 23.6 18.5 33.9 25.5 26.4 32.5 26.7 21.5 23.3 29.5 25.5 26.4 22.4 24.5 24.8
## [16] 30.9 26.4 27.3 29.4 23.0
```

```
class(split(ToothGrowth$len,ToothGrowth$dose))
```

```
## [1] "list"
```

The lapply() function applies a function to a list, returns a list in this case applies mean to len by does

```
lapply(split(ToothGrowth$len,ToothGrowth$dose),mean)
```

```
## $`0.5`
## [1] 10.605
##
## $`1`
## [1] 19.735
##
## $`2`
## [1] 26.1
```

sapply() is a user friendly version of lapply(), it returns a vector

```
sapply(split(ToothGrowth$len,ToothGrowth$dose),mean)
```

```
##    0.5      1      2
## 10.605 19.735 26.100
```

tapply() apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

```
tapply(ToothGrowth$len,ToothGrowth$dose,mean)
```

```
##    0.5      1      2
## 10.605 19.735 26.100
```

two way cross classification

```
sapply(split(ToothGrowth$len,
             list(ToothGrowth$supp,ToothGrowth$dose)),mean)
```

```
## OJ.0.5 VC.0.5   OJ.1   VC.1   OJ.2   VC.2
##  13.23   7.98  22.70  16.77  26.06  26.14
```

```
tapply(ToothGrowth$len,
       list(ToothGrowth$supp,ToothGrowth$dose),mean)
```

```
##      0.5     1     2
## OJ 13.23 22.70 26.06
## VC  7.98 16.77 26.14
```

VIII. Control structures: if, for, repeat, while

1. If statement

```
if (CON) {ACTION1} else {ACTION2}
if (CON) ACTION1 else ACTION2
ifelse(CON, ACTION1, ACTION2)
```

2. Nesting if statements

```
ifelse(CON1, ACTION1,
       ifelse(CON2, ACTION2,
              ACTION3))
```

3. For loop

```
for (variable in sequence){

   Do something

}
```

```
for (i in 1:4){
   j <- i + 10
   print(j)
}
```

4.While loop

```
while (condition){

   Do something

}
```

```
while (i < 5){
   print(i)
   i <- i + 1
}
```

5. Repeat

```
x <- 1
repeat
{
  print(x)
  x<-x+1
  if (x==10) break
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

IX. Function

```
function_name <- function(var){

   Do something

   return(new_variable)
}
```

```
square <- function(x){
   squared <- x*x
   return(squared)
}
```

Data structures: Vectors

- Manual creation: c(element1, element2, ...)
- Functions like seq(), rep(), rnorm(), rbinom(), sample(), etc, return a vector.
- Vectors contain elements of the same type (coercion to same type).
- Numeric vectors.
- Character vectors: insert string value using “” or ‘’
- Logical vectors: F, T, or FALSE, TRUE
  - They are coerced to 0, 1 when used in functions like sum(), mean(), etc
- Factors: vectors with the ‘levels’ attribute: created using factor()
  - Levels can be sorted and labelled.
- Sub-setting vectors:
  - using [],
  - position (also negative), logical vector, or name (in named vectors),
- Vectors recycling.

Data structures: matrices

- Creation from scratch: matrix(x, ncol, nrow).
- Creation by combining vectors:
  - cbind(vector1, vector2, …)
  - rbind(vector1, vector2, …)
- All elements must be of the same type (e. g. all numeric, all character, etc).
- Sub-setting:
  - Using [ ] + Position of rows and columns (, indicates all), logical vector, or name (in named matrices)
- Adding names:
  - colnames() <- c(name1, name2, …)
  - rownames() <- c(name1, name2, …)

Data structures: Lists

- Creation from scratch: list(element1, element2, …)
- Elements can be of the different type and/or different class.
- Adding elements:
  - list_name[['new_element_name']] <- new_element_values
  - list_name[[new_element_number]] <- new_element_values
- Sub-setting:
  - Using [ ] + elements position or elements name or logical vector >> returns a list.
  - Using [[ ]] + element position or element name >> returns the element with its class.
  - Using $ + element name >> returns the element with its class.

Data objects: Data frames

- Creation from scratch: data.frame(col1, col2, …).
- Columns are vectors that can be of the different type.
- Adding elements:
  - df_name[['new_element_name']] <- new_element_values
  - df_name$new_element_number <- new_element_values
- Sub-setting:
  - [ ] + columns position (and rows position) >> may return a data frame or a vector
  - [ ] + logical vector as long as number of columns (or number of rows) >> returns a data frame
  - [ ] + name of columns (and/or rows) >> returns a data frame
  - [[ ]] + column position or column name >> returns a data frame
  - $ + column name >> returns a vector

	Vector	Matrix	List	Dataframe	Tibble
Elements type	Same	Same	Different	Different	Different
Elements length	Same	Same	Different	Same	Same
Dimensions	Number of elements	Number of columns (elements) and rows.	Number of elements, length of each element	Number of columns (elements) and rows.	Number of columns (elements) and rows.
Sub-setting	[]	[]	[] [[]] $	[] [[]] $	[] [[]] $ Always returns tibbles
Printing method	Prints everything	Prints everything	Prints everything	Prints everything	Only prints the first 10 rows.
Recycling	-	Yes	-	Yes	No

General functions for objects

- is functions
- as functions
- length()
- dim()
- class()
- str()
- attributes()
- names(), colnames(), rownames()

Control flows: Repetitive execution

- apply
  - apply(x, MARGIN, fun)
  - X is a matrix
- sapply()
  - sapply(x, fun)
  - X is a list
  - Applies a function to all the elements of a list
- lapply()
  - lapply(x, fun)
  - X is a list
  - Applies a function to all the elements of a list
- tapply()
  - tapply(vec1, factor, function)

- Loops:
  - for loops

```
for (variable in vector) {
    commands to be repeated
}
```

  - while loop

```
while(condition) {
    commands to be repeated
}
```

Control flows: conditional execution

- Conditional execution: if statements
  - if (condition) {

```
    expression1
 } else {
    expression2
 }
```

- Vectorized version:

  ifelse(condition, expression1, expression2)

Advanced topics in Writing functions

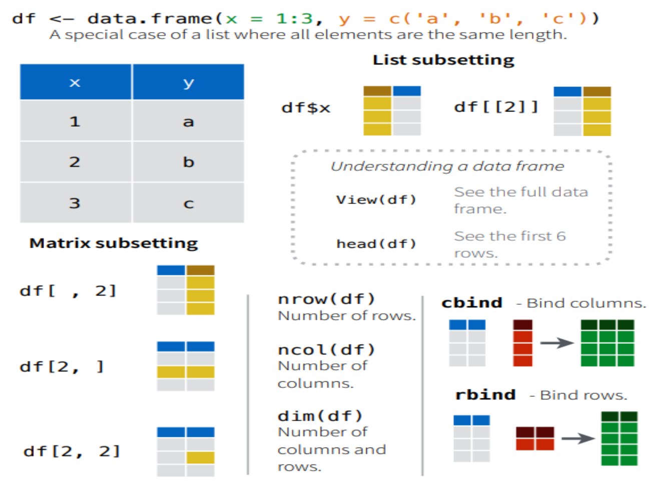
- Name masking.
  - Search path
- Returning multiple values:
  - The return() function only takes one argument, thus can only return one object. If more objects need to be returned, they need to be embedded in a list
- Default values:
  - Arguments can be set equal to default values.
- The `…' argument:
  - For passing any admissible argument.
- The stop() function:
  - Stops functions execution and prints a message.
- The switch() function:
  - For conditional execution.

Data visualization with ggplot

- First ggplot() function.
- Then geom functions with aesthetics > to map variables from the data frame to the plot.

ggplot(data = <DATA>) +
<GEOM FUNCTION>(mapping = aes(<MAPPINGS>))

- Local vs global mappings.
- Plotting subsets of the dataset.
- Plotting more than two variables (colour, size, group, facets).
- Graphics for communication.



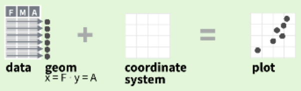
```
df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))
```

A special case of a list where all elements are the same length.

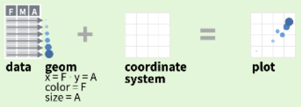
Data Visualization with ggplot2 : : CHEAT SHEET

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and **geoms**—visual marks that represent data points.



To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.



Complete the template below to build a graph.

ggplot (data = <DATA>) +
<GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
stat = <STAT>, position = <POSITION>) +
<COORDINATE_FUNCTION> +
<FACET_FUNCTION> +
<SCALE_FUNCTION> +
<THEME_FUNCTION>

required

Not required, sensible defaults supplied

ggplot(data = mpg, **aes(**x = cty, y = hwy**))** Begins a plot that you finish by adding layers to. Add one geom function per layer.

last_plot() Returns the last plot.

ggsave("plot.png", width = 5, height = 5**)** Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Aes

Common aesthetic values.

color and **fill** - string ("red", "#RRGGBB")

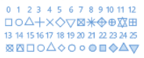
linetype - integer or string (0 = "blank", 1 = "solid", 2 = "dashed", 3 = "dotted", 4 = "dotdash", 5 = "longdash", 6 = "twodash")

lineend - string ("round", "butt", or "square")

linejoin - string ("round", "mitre", or "bevel")

size - integer (line width in mm)

shape - integer/shape name or a single character ("a")



Geoms

Use a geom function to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

GRAPHICAL PRIMITIVES

a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))

a + geom_blank() and **a + expand_limits()**
Ensure limits include values across all plots.

b + geom_curve(aes(yend = lat + 1, xend = long + 1), curvature = 1**)** - x, xend, y, yend, alpha, angle, color, curvature, linetype, size

a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1**)**
x, y, alpha, color, group, linetype, size

a + geom_polygon(aes(alpha = 50)**)** - x, y, alpha, color, fill, group, subgroup, linetype, size

b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1)**)** - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)**)** - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS

common aesthetics: x, y, alpha, color, linetype, size

b + geom_abline(aes(intercept = 0, slope = 1)**)**
b + geom_hline(aes(yintercept = lat)**)**
b + geom_vline(aes(xintercept = long)**)**

b + geom_segment(aes(yend = lat + 1, xend = long + 1)**)**
b + geom_spoke(aes(angle = 1:1155, radius = 1)**)**

ONE VARIABLE continuous

c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)

c + geom_area(stat = "bin"**)**
x, y, alpha, color, fill, linetype, size

c + geom_density(kernel = "gaussian"**)**
x, y, alpha, color, fill, group, linetype, size, weight

c + geom_dotplot()
x, y, alpha, color, fill

c + geom_freqpoly()
x, y, alpha, color, group, linetype, size

c + geom_histogram(binwidth = 5**)**
x, y, alpha, color, fill, linetype, size, weight

c2 + geom_qq(aes(sample = hwy)**)**
x, y, alpha, color, fill, linetype, size, weight

discrete

d <- ggplot(mpg, aes(fl))

d + geom_bar()
x, alpha, color, fill, linetype, size, weight

TWO VARIABLES

both continuous

e <- ggplot(mpg, aes(cty, hwy))

e + geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1**)** - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

e + geom_point()
x, y, alpha, color, fill, shape, size, stroke

e + geom_quantile()
x, y, alpha, color, group, linetype, size, weight

e + geom_rug(sides = "bl"**)**
x, y, alpha, color, linetype, size

e + geom_smooth(method = lm**)**
x, y, alpha, color, fill, group, linetype, size, weight

e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1**)** - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

one discrete, one continuous

f <- ggplot(mpg, aes(class, hwy))

f + geom_col()
x, y, alpha, color, fill, group, linetype, size

f + geom_boxplot()
x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

f + geom_dotplot(binaxis = "y", stackdir = "center"**)**
x, y, alpha, color, fill, group

f + geom_violin(scale = "area"**)**
x, y, alpha, color, fill, group, linetype, size, weight

both discrete

g <- ggplot(diamonds, aes(cut, color))

g + geom_count()
x, y, alpha, color, fill, shape, size, stroke

e + geom_jitter(height = 2, width = 2**)**
x, y, alpha, color, fill, shape, size

continuous bivariate distribution

h <- ggplot(diamonds, aes(carat, price))

h + geom_bin2d(binwidth = c(0.25, 500)**)**
x, y, alpha, color, fill, linetype, size, weight

h + geom_density_2d()
x, y, alpha, color, group, linetype, size

h + geom_hex()
x, y, alpha, color, fill, size

continuous function

i <- ggplot(economics, aes(date, unemploy))

i + geom_area()
x, y, alpha, color, fill, linetype, size

i + geom_line()
x, y, alpha, color, group, linetype, size

i + geom_step(direction = "hv"**)**
x, y, alpha, color, group, linetype, size

visualizing error

df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))

j + geom_crossbar(fatten = 2**)** - x, y, ymax, ymin, alpha, color, fill, group, linetype, size

j + geom_errorbar() - x, ymax, ymin, alpha, color, group, linetype, size, width
Also **geom_errorbarh()**.

j + geom_linerange()
x, ymin, ymax, alpha, color, group, linetype, size

j + geom_pointrange() - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

maps

data <- data.frame(murder = USArrests$Murder,
state = tolower(rownames(USArrests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))

k + geom_map(aes(map_id = state), map = map**)**
+ expand_limits(x = map$long, y = map$lat**)**
map_id, alpha, color, fill, linetype, size

THREE VARIABLES

seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))

l + geom_contour(aes(z = z)**)**
x, y, z, alpha, color, group, linetype, size, weight

l + geom_contour_filled(aes(fill = z)**)**
x, y, alpha, color, fill, group, linetype, size, subgroup

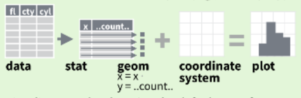
l + geom_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE**)**
x, y, alpha, fill

l + geom_tile(aes(fill = z)**)**
x, y, alpha, color, fill, linetype, size, width

Stats

An alternative way to build a layer.

A stat builds new variables to plot (e.g., count, prop).



Visualize a stat by changing the default stat of a geom function, **geom_bar(stat="count")** or by using a stat function, **stat_count(geom="bar")**, which calls a default geom to make a layer (equivalent to a geom function). Use **..name..** syntax to map stat variables to aesthetics.

geom to use · stat function · geommappings

i + stat_density_2d(aes(fill = ..level..),
geom = "polygon"**)**

variable created by stat

c + stat_bin(binwidth = 1, boundary = 10**)**
x, y | ..count.., ..ncount.., ..density.., ..ndensity..

c + stat_count(width = 1**) x, y** | ..count.., ..prop..

c + stat_density(adjust = 1, kernel = "gaussian"**)**
x, y | ..count.., ..density.., ..scaled..

e + stat_bin_2d(bins = 30, drop = T**)**
x, y, fill | ..count.., ..density..

e + stat_bin_hex(bins = 30**) x, y, fill** | ..count.., ..density..

e + stat_density_2d(contour = TRUE, n = 100**)**
x, y, color, size | ..level..

e + stat_ellipse(level = 0.95, segments = 51, type = "t"**)**

l + stat_contour(aes(z = z)**) x, y, z, order** | ..level..

l + stat_summary_hex(aes(z = z), bins = 30, fun = max**)**
x, y, z, fill | ..value..

l + stat_summary_2d(aes(z = z), bins = 30, fun = mean**)**
x, y, z, fill | ..value..

f + stat_boxplot(coef = 1.5**)**
x, y | ..lower.., ..middle.., ..upper.., ..width.. , ..ymin.., ..ymax..

f + stat_ydensity(kernel = "gaussian", scale = "area"**) x, y**
| ..density.., ..scaled.., ..count.., ..n.., ..violinwidth.., ..width..

e + stat_ecdf(n = 40**) x, y** | ..x.., ..y..

e + stat_quantile(quantiles = c(0.1, 0.9), formula = y ~ log(x), method = "rq"**) x, y** | ..quantile..

e + stat_smooth(method = "lm", formula = y ~ x, se = T, level = 0.95**) x, y** | ..se.., ..x.., ..y.., ..ymin.., ..ymax..

ggplot() + xlim(-5, 5**) + stat_function(**fun = dnorm, n = 20, geom = "point"**) x** | ..x.., ..y..

ggplot() + stat_qq(aes(sample = 1:100)**)**
x, y, sample | ..sample.., ..theoretical..

e + stat_sum() x, y, size | ..n.., ..prop..

e + stat_summary(fun.data = "mean_cl_boot"**)**

h + stat_summary_bin(fun = "mean", geom = "bar"**)**

e + stat_identity()

e + stat_unique()

Scales

Override defaults with **scales** package.

Scales map data values to the visual values of an aesthetic. To change a mapping, add a new scale.

n <- d + geom_bar(aes(fill = fl))

scale_ · aesthetic to adjust · prepackaged scale to use · scale-specific arguments

n + scale_fill_manual(
values = c("skyblue", "royalblue", "blue", "navy"),
limits = c("d", "e", "p", "r"), breaks =c("d", "e", "p", "r"),
name = "fuel", labels = c("D", "E", "P", "R")**)**

range of values to include in mapping · title to use in legend/axis · labels to use in legend/axis · breaks to use in legend/axis

GENERAL PURPOSE SCALES

Use with most aesthetics

scale_*_continuous() - Map cont' values to visual ones.
scale_*_discrete() - Map discrete values to visual ones.
scale_*_binned() - Map continuous values to discrete bins.
scale_*_identity() - Use data values as visual ones.
scale_*_manual(values = c()**)** - Map discrete values to manually chosen visual ones.
scale_*_date(date_labels = "%m/%d"), date_breaks = "2 weeks"**)** - Treat data values as dates.
scale_*_datetime() - Treat data values as date times. Same as scale_*_date(). See ?strptime for label formats.

X & Y LOCATION SCALES

Use with x or y aesthetics (x shown here)

scale_x_log10() - Plot x on log10 scale.
scale_x_reverse() - Reverse the direction of the x axis.
scale_x_sqrt() - Plot x on square root scale.

COLOR AND FILL SCALES (DISCRETE)

n + scale_fill_brewer(palette = "Blues"**)**
For palette choices:
RColorBrewer::display.brewer.all()

n + scale_fill_grey(start = 0.2, end = 0.8, na.value = "red"**)**

COLOR AND FILL SCALES (CONTINUOUS)

o <- c + geom_dotplot(aes(fill = ..x..))

o + scale_fill_distiller(palette = "Blues"**)**

o + scale_fill_gradient(low="red", high="yellow"**)**

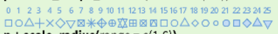
o + scale_fill_gradient2(low = "red", high = "blue", mid = "white", midpoint = 25**)**

o + scale_fill_gradientn(colors = topo.colors(6)**)**
Also: rainbow(), heat.colors(), terrain.colors(), cm.colors(), RColorBrewer::brewer.pal()

SHAPE AND SIZE SCALES

p <- e + geom_point(aes(shape = fl, size = cyl))

p + scale_shape() + scale_size()
p + scale_shape_manual(values = c(3:7)**)**



p + scale_radius(range = c(1,6)**)**
p + scale_size_area(max_size = 6**)**

Coordinate Systems

r <- d + geom_bar()

r + coord_cartesian(xlim = c(0, 5)**)** - xlim, ylim
The default cartesian coordinate system.

r + coord_fixed(ratio = 1/2**)**
ratio, xlim, ylim - Cartesian coordinates with fixed aspect ratio between x and y units.

ggplot(mpg, aes(y = fl)**) + geom_bar()**
Flip cartesian coordinates by switching x and y aesthetic mappings.

r + coord_polar(theta = "x", direction=1**)**
theta, start, direction - Polar coordinates.

r + coord_trans(y = "sqrt"**)** - x, y, xlim, ylim
Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.

π + coord_quickmap()
π + coord_map(projection = "ortho", orientation = c(41, -74, 0)**)** - projection, xlim, ylim
Map projections from the mapproj package (mercator (default), azequalarea, lagrange, etc.).

Position Adjustments

Position adjustments determine how to arrange geoms that would otherwise occupy the same space.

s <- ggplot(mpg, aes(fl, fill = drv))

s + geom_bar(position = "dodge")
Arrange elements side by side.

s + geom_bar(position = "fill")
Stack elements on top of one another, normalize height.

e + geom_point(position = "jitter")
Add random noise to X and Y position of each element to avoid overplotting.

e + geom_label(position = "nudge")
Nudge labels away from points.

s + geom_bar(position = "stack")
Stack elements on top of one another.

Each position adjustment can be recast as a function with manual **width** and **height** arguments:
s + geom_bar(position = position_dodge(width = 1))

Themes

r + theme_bw()
White background with grid lines.

r + theme_gray()
Grey background (default theme).

r + theme_dark()
Dark for contrast.

r + theme_classic()

r + theme_light()

r + theme_linedraw()

r + theme_minimal()
Minimal theme.

r + theme_void()
Empty theme.

r + theme() Customize aspects of the theme such as axis, legend, panel, and facet properties.
r + ggtitle("Title") + theme(plot.title.postion = "plot")
r + theme(panel.background = element_rect(fill = "blue"))

Faceting

Facets divide a plot into subplots based on the values of one or more discrete variables.

t <- ggplot(mpg, aes(cty, hwy)) + geom_point()

t + facet_grid(cols = vars(fl))
Facet into columns based on fl.

t + facet_grid(rows = vars(year))
Facet into rows based on year.

t + facet_grid(rows = vars(year), cols = vars(fl))
Facet into both rows and columns.

t + facet_wrap(vars(fl))
Wrap facets into a rectangular layout.

Set **scales** to let axis limits vary across facets.

t + facet_grid(rows = vars(drv), cols = vars(fl),
scales = "free")
x and y axis limits adjust to individual facets:
"free_x" - x axis limits adjust
"free_y" - y axis limits adjust

Set **labeller** to adjust facet label:

t + facet_grid(cols = vars(fl), labeller = label_both)

fl: c	fl: d	fl: e	fl: p	fl: r

t + facet_grid(rows = vars(fl),
labeller = label_bquote(alpha ^ .(fl)**))**

α^c	α^d	α^e	α^p	α^r

Labels and Legends

Use **labs()** to label the elements of your plot.

t + labs(x = "New x axis label", **y** = "New y axis label",
title ="Add a title above the plot",
subtitle = "Add a subtitle below title",
caption = "Add a caption below plot",
alt = "Add alt text to the plot",
<AES> = "New <AES> legend title"**)**

t + annotate(geom = "text", x = 8, y = 9, label = "A"**)**
Places a geom with manually selected aesthetics.

p + guides(x = guide_axis(n.dodge = 2)**)** Avoid crowded or overlapping labels with guide_axis(n.dodge or angle).

n + guides(fill = "none"**)** Set legend type for each aesthetic: colorbar, legend, or none (no legend).

n + theme(legend.position = "bottom"**)**
Place legend at "bottom", "top", "left", or "right".

n + scale_fill_discrete(name = "Title", labels = c("A", "B", "C", "D", "E")**)**
Set legend title and labels with a scale function.

Zooming

Without clipping (preferred):

t + coord_cartesian(xlim = c(0, 100), ylim = c(10, 20)**)**

With clipping (removes unseen data points):

t + xlim(0, 100**) + ylim(**10, 20**)**

t + scale_x_continuous(limits = c(0, 100)**) +**
scale_y_continuous(limits = c(0, 100)**)**