Functions, Attributes, and Class

Data Analysis with R and Python

Deepayan Sarkar

Functions

- Most useful things in R happen by calling functions
- Functions have one or more arguments
  - All arguments have names (available as variables inside the function)
  - Arguments may be compulsory or optional
  - Optional arguments usually have "default" values

Functions

- Most useful things in R happen by calling functions
- Functions have one or more arguments
  - All arguments have names (available as variables inside the function)
  - Arguments may be compulsory or optional
  - Optional arguments usually have "default" values
- Arguments may or may not be named when the function is called
  - Unnamed arguments are matched by position
  - Optional arguments are usually named
- Functions normally also have a useful "return" value

Example: Delhi Air Quality data

```
aqi <- read.csv("https://deepayan.github.io/BSDS/2024-01-DE/data/rkpuram-aqi.csv")
str(aqi)
```

```
'data.frame':	3930 obs. of  7 variables:
 $ date: chr  "2024/11/1" "2024/11/2" "2024/11/3" "2024/11/4" ...
 $ pm25: int  300 306 308 300 282 267 307 275 269 260 ...
 $ pm10: int  260 249 298 246 227 251 205 198 195 264 ...
 $ o3  : int  40 57 53 56 52 47 40 46 53 46 ...
 $ no2 : int  19 16 21 12 15 16 13 9 8 9 ...
 $ so2 : int  8 5 3 9 8 5 9 10 11 18 ...
 $ co  : int  17 19 17 24 22 19 21 22 19 32 ...
```

Example: Delhi Air Quality data

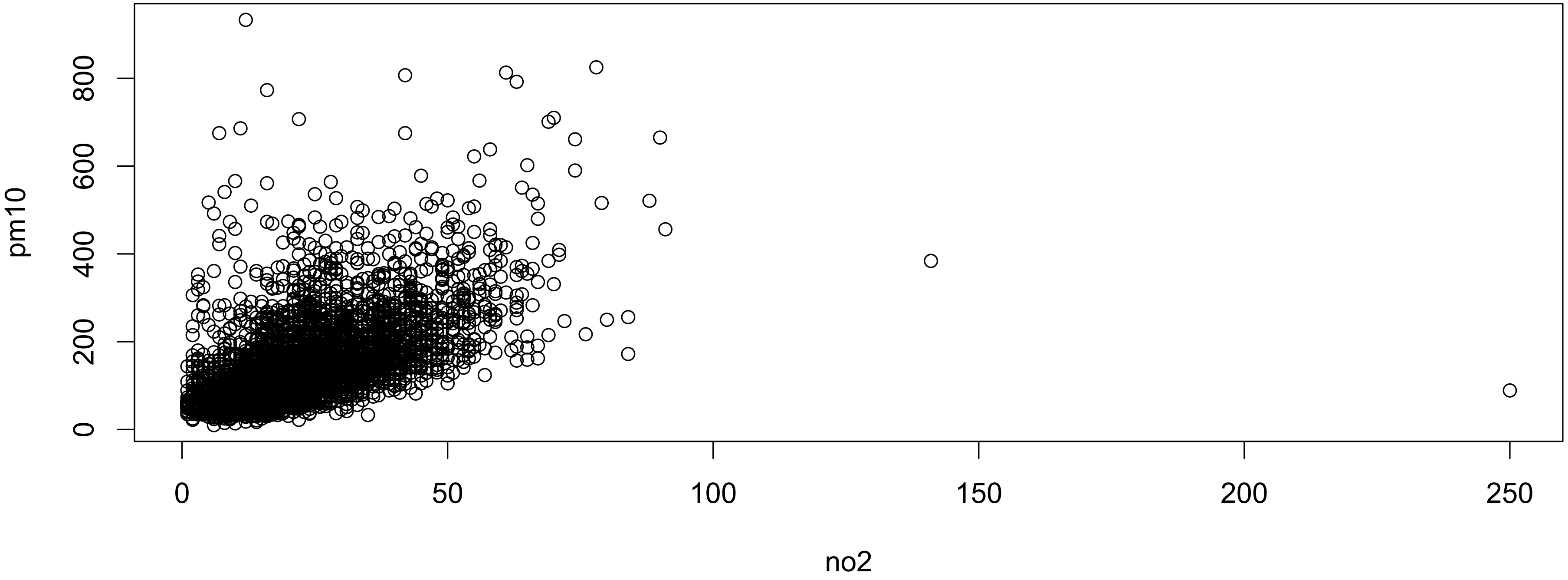
- How is AQI (PM10) related to `no2` (Nitrogen dioxide)?

Example: Delhi Air Quality data

- How is AQI (PM10) related to `no2` (Nitrogen dioxide)?
- Basic tools we are familiar with
  - Scatter plot
  - Linear Regression

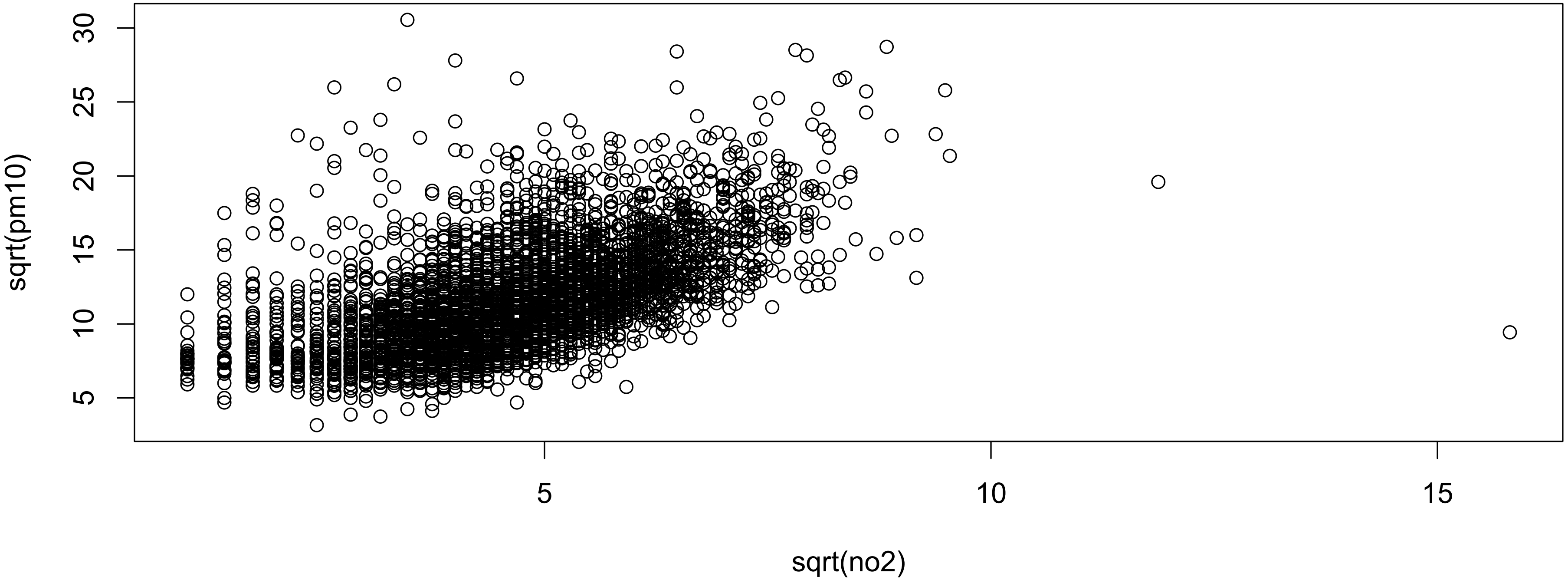
Example: Delhi Air Quality data

```r
plot(pm10 ~ no2, data = aqi)
```



Example: Delhi Air Quality data

```
plot(sqrt(pm10) ~ sqrt(no2), data = aqi)
```



Example: Delhi Air Quality data

- Formula for linear regression line

$$\hat{b} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

$$\hat{a} = \bar{y} - \hat{b}\bar{x}$$

Example: Delhi Air Quality data

- Calculation of linear regression line

```
x <- sqrt(aqi$no2)
y <- sqrt(aqi$pm10)
xbar <- mean(x)
ybar <- mean(y)
sxy <- sum( (x - xbar) * (y - ybar) )
sxx <- sum( (x - xbar)^2 )
```

Example: Delhi Air Quality data

- Calculation of linear regression line

```
x <- sqrt(aqi$no2)
y <- sqrt(aqi$pm10)
xbar <- mean(x)
ybar <- mean(y)
sxy <- sum( (x - xbar) * (y - ybar) )
sxx <- sum( (x - xbar)^2 )
```

```
c(xbar, ybar, sxx, sxy)
```

```
[1] NA NA NA NA
```

Example: Delhi Air Quality data

- Regression coefficients removing missing values

```
x <- sqrt(aqi$no2)
y <- sqrt(aqi$pm10)
xbar <- mean(x, na.rm = TRUE)
ybar <- mean(y, na.rm = TRUE)
sxy <- sum( (x - xbar) * (y - ybar), na.rm = TRUE)
sxx <- sum( (x - xbar)^2, na.rm = TRUE)
```

```
c(xbar, ybar, sxx, sxy)
```

```
[1]     4.703466    11.808792  7506.779092 11173.744082
```

Example: Delhi Air Quality data

- Even this may not be correct: should remove both if only one missing!

```
x <- sqrt(aqi$no2)
y <- sqrt(aqi$pm10)
ok <- is.finite(x) & is.finite(y)
xbar <- mean(x[ok]); ybar <- mean(y[ok])
sxy <- sum( (x[ok] - xbar) * (y[ok] - ybar))
sxx <- sum( (x[ok] - xbar)^2)
```

```
c(xbar, ybar, sxx, sxy)
```

```
[1]     4.700322    11.827206  7480.368670 11173.964771
```

Example: Delhi Air Quality data

- Coefficients of linear regression line

```
bhat <- sxy / sxx
ahat <- ybar - bhat * xbar
c(intercept = ahat, slope = bhat)
```

```
intercept     slope
 4.805996  1.493772
```

Example: Delhi Air Quality data

- Coefficients of linear regression line

```
bhat <- sxy / sxx
ahat <- ybar - bhat * xbar
c(intercept = ahat, slope = bhat)
```

```
intercept     slope 
 4.805996  1.493772 
```

- Standard R function to do this

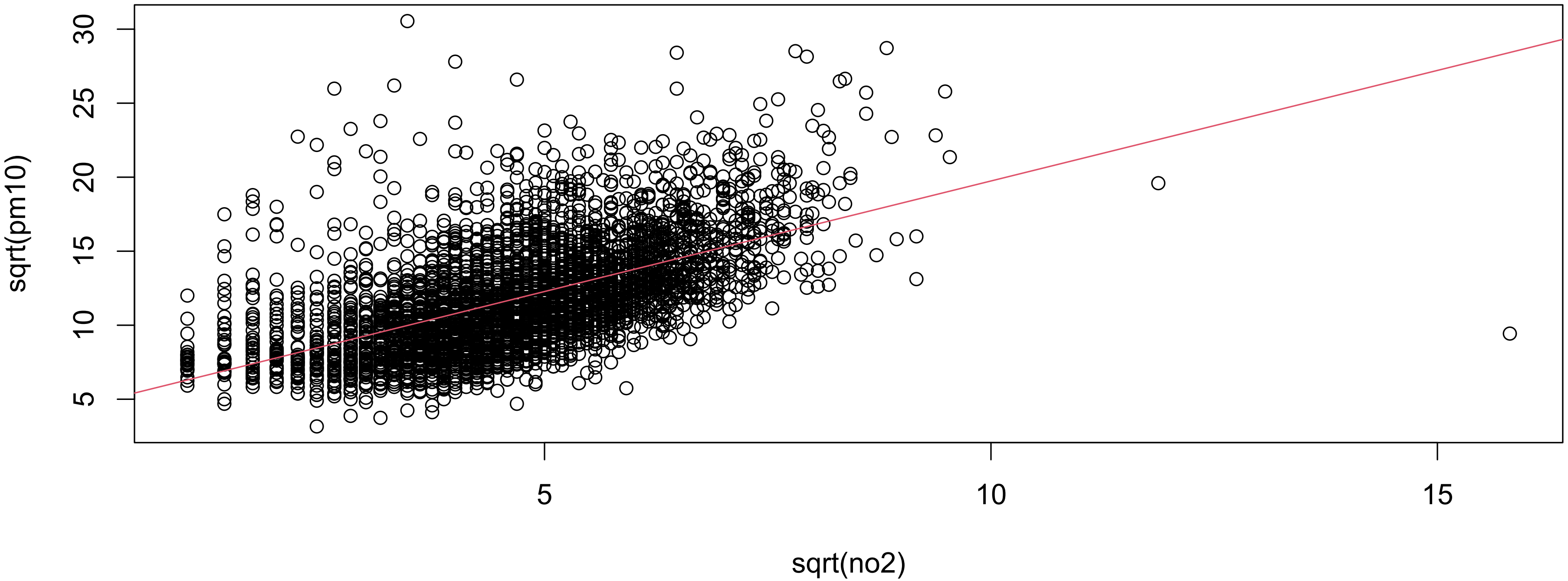
```
lm(sqrt(pm10) ~ sqrt(no2), data = aqi)
```

```

Call:
lm(formula = sqrt(pm10) ~ sqrt(no2), data = aqi)

Coefficients:
(Intercept)    sqrt(no2)  
      4.806        1.494  
```

Example: Delhi Air Quality data

- Add regression line to scatter plot

```
plot(sqrt(pm10) ~ sqrt(no2), data = aqi)
abline(ahat, bhat, col = 2)
```



Functions

- We have used several functions above
  - `sqrt()`, `mean()`, `sum()` — basic mathematical / summary functions
  - `plot()` — "high level" plotting function
  - `abline()` — "low level" plotting function
  - `lm()` — "high level" modeling function

Functions

- We will discuss graphics functions in more detail later

Functions

- We will discuss graphics functions in more detail later
- `lm()` is a good example to study the behaviour of modeling functions in R

Functions

- We will discuss graphics functions in more detail later
- `lm()` is a good example to study the behaviour of modeling functions in R
- Recall:
  - Functions have one or more arguments
  - All arguments have names (available as variables inside the function)
  - Arguments may be compulsory or optional
  - Optional arguments usually have "default" values

Functions

- `lm()` fits a more general class of models known as *linear models*

```
str(lm)
```

```
function (formula, data, subset, weights, na.action, method = "qr", model = TRUE,
    x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset,
    ...)
```

Functions

- `lm()` fits a more general class of models known as *linear models*

```
str(lm)
```

```
function (formula, data, subset, weights, na.action, method = "qr", model = TRUE,
    x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset,
    ...)
```

- These calls are equivalent

```
fm1 <- lm(pm10 ~ no2, aqi, (no2 < 100))
fm2 <- lm(pm10 ~ no2, data = aqi, method = "qr", subset = (no2 < 100))
coef(fm1)
```

```
(Intercept)         no2
  52.011396    4.228562
```

```
coef(fm2)
```

```
(Intercept)         no2
  52.011396    4.228562
```

Functions

- Rule: named arguments are matched by name, remaining by position
- Convention:
  - First few "standard" arguments are usually unnamed (matched by position)
  - Usually unnamed arguments are *not* used after named arguments

Functions

- Rule: named arguments are matched by name, remaining by position
- Convention:
  - First few "standard" arguments are usually unnamed (matched by position)
  - Usually unnamed arguments are *not* used after named arguments
- The following call is equivalent to previous two, but not recommended

```
fm3 <- lm(pm10 ~ no2, data = aqi, no2 < 100)
coef(fm3)
```

```
(Intercept)         no2
  52.011396    4.228562
```

Functions

- The return value of `lm()` is a list

```
str(fm2)
```

```
List of 13
 $ coefficients : Named num [1:2] 52.01 4.23
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "no2"
 $ residuals    : Named num [1:3810] 128 129 157 143 112 ...
  ..- attr(*, "names")= chr [1:3810] "1" "2" "3" "4" ...
 $ effects      : Named num [1:3810] -9466 3472 155 139 108 ...
  ..- attr(*, "names")= chr [1:3810] "(Intercept)" "no2" "" "" ...
 $ rank         : int 2
 $ fitted.values: Named num [1:3810] 132 120 141 103 115 ...
  ..- attr(*, "names")= chr [1:3810] "1" "2" "3" "4" ...
 $ assign       : int [1:2] 0 1
 $ qr           :List of 5
  ..$ qr    : num [1:3810, 1:2] -61.7252 0.0162 0.0162 0.0162 0.0162
```

Functions

- Individual elements can be extracted using list indexing

```
fm2$coefficients
```

```
(Intercept)         no2
  52.011396    4.228562
```

```
fm2$residuals
```

```
          1           2           3           4           5           6
127.64591729 129.33160471 157.18879234 143.24585460 111.56016718 131.33160471
          7           8           9          10          11          12
 98.01729213 107.93154203 109.16010450 173.93154203  98.61722945 414.61722945
         13          15          16          17          18          19
255.07435439 337.41735482 653.33160471  55.58885503 113.64591729  36.27454245
         20          21          22          23          24          25
 95.90316761 104.04597998  -2.49689508  73.58885503 -16.72545755  -3.95402002
         26          28          29          30          31          32
  9.27454245  17.70297955  19.47441708  27.93154203  26.16010450  11.38866697
         33          34          35          36          37          38
 13.93154203   9.93154203   3.70297955   4.93154203   1.70297955   4.56016718
         39          40          41          42          43          44
```

Attributes

- The names (derived from row names of the data) can be used as index

```
fm2$residuals["25"]
```

```
     25
-3.95402
```

- The names associated with a vector can be obtained using `names()`

```
names(fm2$residuals)
```

```
  [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"   "10"  "11"  "12"
 [13] "13"  "15"  "16"  "17"  "18"  "19"  "20"  "21"  "22"  "23"  "24"  "25"
 [25] "26"  "28"  "29"  "30"  "31"  "32"  "33"  "34"  "35"  "36"  "37"  "38"
 [37] "39"  "40"  "41"  "42"  "43"  "44"  "45"  "46"  "47"  "48"  "49"  "50"
 [49] "51"  "52"  "53"  "54"  "55"  "56"  "57"  "58"  "59"  "60"  "61"  "62"
 [61] "63"  "64"  "65"  "66"  "67"  "68"  "69"  "70"  "71"  "72"  "73"  "74"
 [73] "75"  "76"  "77"  "78"  "79"  "80"  "81"  "82"  "83"  "84"  "85"  "86"
 [85] "87"  "88"  "89"  "90"  "91"  "92"  "93"  "94"  "95"  "96"  "97"  "98"
 [97] "99"  "100" "101" "102" "103" "104" "105" "106" "107" "108" "109" "110"
[109] "111" "112" "113" "114" "115" "116" "117" "118" "119" "120" "121" "122"
[121] "123" "124" "125" "126" "127" "128" "129" "130" "131" "132" "133" "134"
```

Attributes

- These names are actually stored as an *attribute* called "names"

```
attr(fm2$residuals, "names")
```

```
  [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"   "10"  "11"  "12" 
 [13] "13"  "15"  "16"  "17"  "18"  "19"  "20"  "21"  "22"  "23"  "24"  "25" 
 [25] "26"  "28"  "29"  "30"  "31"  "32"  "33"  "34"  "35"  "36"  "37"  "38" 
 [37] "39"  "40"  "41"  "42"  "43"  "44"  "45"  "46"  "47"  "48"  "49"  "50" 
 [49] "51"  "52"  "53"  "54"  "55"  "56"  "57"  "58"  "59"  "60"  "61"  "62" 
 [61] "63"  "64"  "65"  "66"  "67"  "68"  "69"  "70"  "71"  "72"  "73"  "74" 
 [73] "75"  "76"  "77"  "78"  "79"  "80"  "81"  "82"  "83"  "84"  "85"  "86" 
 [85] "87"  "88"  "89"  "90"  "91"  "92"  "93"  "94"  "95"  "96"  "97"  "98" 
 [97] "99"  "100" "101" "102" "103" "104" "105" "106" "107" "108" "109" "110"
[109] "111" "112" "113" "114" "115" "116" "117" "118" "119" "120" "121" "122"
[121] "123" "124" "125" "126" "127" "128" "129" "130" "131" "132" "133" "134"
[133] "135" "136" "137" "138" "139" "140" "141" "142" "143" "144" "145" "146"
[145] "147" "148" "149" "150" "151" "152" "153" "154" "155" "156" "157" "158"
[157] "159" "160" "161" "162" "163" "164" "165" "167" "168" "169" "170" "171"
[169] "172" "173" "174" "175" "176" "177" "178" "179" "180" "181" "182" "183"
[181] "184" "185" "186" "187" "188" "189" "190" "191" "192" "193" "194" "195"
[193] "196" "197" "198" "199" "200" "201" "202" "203" "204" "205" "206" "207"
[205] "208" "209" "210" "211" "212" "213" "214" "215" "216" "217" "218" "219"
```

Attributes

- This is true for all vector objects, including lists

```
attr(fm2, "names")
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values"
 [6] "assign"        "qr"            "df.residual"   "na.action"     "xlevels"
[11] "call"          "terms"         "model"
```

Attributes

- Attributes are arbitrary R objects that can be attached to any other object
- Typically used for programming convenience, normally not seen by users
- However, some attributes are "special"

Attributes

- The "names" attribute can be extracted using the function `names()`
- `dimnames()` similarly gives row / column names for matrices and arrays

```
dimnames(Titanic)
```

```
$Class
[1] "1st"  "2nd"  "3rd"  "Crew"

$Sex
[1] "Male"   "Female"

$Age
[1] "Child" "Adult"

$Survived
[1] "No"  "Yes"
```

Attributes

- The "names" attribute can be extracted using the function `names()`
- `dimnames()` similarly gives row / column names for matrices and arrays

```
attr(Titanic, "dimnames")
```

```
$Class
[1] "1st"  "2nd"  "3rd"  "Crew"

$Sex
[1] "Male"   "Female"

$Age
[1] "Child" "Adult"

$Survived
[1] "No"  "Yes"
```

Attributes

- For example, column names can be obtained as

```
dimnames(Titanic)[[2]]
```

```
[1] "Male"   "Female"
```

- There are convenient shortcuts called `rownames()` and `colnames()`

```
colnames(Titanic)
```

```
[1] "Male"   "Female"
```

Attributes

- In fact, we can easily verify that this is what `colnames()` is doing by printing it

```
colnames
```

```
function (x, do.NULL = TRUE, prefix = "col")
{
    if (is.data.frame(x) && do.NULL)
        names(x)
    else dimnames(x)[[2L]] %||% if (do.NULL)
        NULL
    else {
        nc <- NCOL(x)
        if (nc > 0L)
            paste0(prefix, seq_len(nc))
        else character()
    }
}
<bytecode: 0x7fb403a14358>
<environment: namespace:base>
```

- All R functions can be easily inspected in this way

Attributes

- Another very important attribute is "class"
- For example, the return value of `lm()` has class "lm"

```
attr(fm2, "class")
```

```
[1] "lm"
```

- The class of an object can also be obtained using the function `class()`

```
class(fm2)
```

```
[1] "lm"
```

Attributes

- The class of an object can (usually) be "removed" by setting it to `NULL`
- This is not something you should actually do!

```
class(fm2) <- NULL
```

- Such objects will no longer have a "class" attribute

```
attr(fm2, "class")
```

```
NULL
```

- However, it will still have a class (implicitly)

```
class(fm2)
```

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

Class, generic functions, and methods

- The class of an object is fundamental to how R works
- Every object in R must have a class
- This is true even if the object does not have a class attribute

```
class(colnames)
```

```
[1] "function"
```

```
attr(colnames, "class")
```

```
NULL
```

```
class(Titanic)
```

```
[1] "table"
```

```
attr(Titanic, "class")
```

```
[1] "table"
```

Class, generic functions, and methods

- The main use of the class of an object is in how *generic functions* behave
- Generic functions are intended to perform general tasks, like
    - `print()`
    - `plot()`
    - `summary()`
- But details of what these functions should do depends on the input

Class, generic functions, and methods

```
print(Titanic[, , 1, 1])
```

```
      Sex
Class  Male Female
  1st     0      0
  2nd     0      0
  3rd    35     17
  Crew    0      0
```

```
fm1 <- lm(pm10 ~ no2, aqi, subset = (no2 < 100))
print(fm1)
```

```

Call:
lm(formula = pm10 ~ no2, data = aqi, subset = (no2 < 100))

Coefficients:
(Intercept)          no2
     52.011        4.229
```

Class, generic functions, and methods

```
summary(Titanic)
```

```
Number of cases in table: 2201
Number of factors: 4
Test for independence of all factors:
    Chisq = 1637.4, df = 25, p-value = 0
    Chi-squared approximation may be incorrect
```

Class, generic functions, and methods

```
summary(fm1)
```

```

Call:
lm(formula = pm10 ~ no2, data = aqi, subset = (no2 < 100))

Residuals:
    Min      1Q  Median      3Q     Max 
-235.21  -51.35  -17.91   28.72  830.25 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  52.0114     2.7786   18.72   <2e-16 ***
no2           4.2286     0.1014   41.71   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 83.24 on 3808 degrees of freedom
  (118 observations deleted due to missingness)
Multiple R-squared:  0.3136,	Adjusted R-squared:  0.3135 
F-statistic:  1740 on 1 and 3808 DF,  p-value: < 2.2e-16
```

Class, generic functions, and methods

- But suppose we make a copy of `fm1` and remove the class attribute from it

```
fm2 <- fm1
class(fm2) <- NULL
class(fm1)
```

```
[1] "lm"
```

```
class(fm2)
```

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

- `fm1` and `fm2` represent the same model fit
- But the different class means that `print()` and `summary()` behave differently

Class, generic functions, and methods

```
summary(fm2)
```

```
              Length Class      Mode
coefficients     2   -none-     numeric
residuals     3810   -none-     numeric
effects       3810   -none-     numeric
rank             1   -none-     numeric
fitted.values 3810   -none-     numeric
assign           2   -none-     numeric
qr               5   qr         list
df.residual      1   -none-     numeric
na.action      118   omit       numeric
xlevels          0   -none-     list
call             4   -none-     call
terms            3   terms      call
model            2   data.frame list
```

Class, generic functions, and methods

```
print(fm2)
```

```
$coefficients
(Intercept)         no2
  52.011396    4.228562

$residuals
           1            2            3            4            5            6
 127.64591729 129.33160471 157.18879234 143.24585460 111.56016718 131.33160471
           7            8            9           10           11           12
  98.01729213 107.93154203 109.16010450 173.93154203  98.61722945 414.61722945
          13           15           16           17           18           19
 255.07435439 337.41735482 653.33160471  55.58885503 113.64591729  36.27454245
          20           21           22           23           24           25
  95.90316761 104.04597998  -2.49689508  73.58885503 -16.72545755  -3.95402002
```

- This kind of customized output is achieved by *methods*

Class, generic functions, and methods

- Methods are specific implementations of a generic function customized to its input
- The appropriate method is chosen by looking at the *class* of the input argument

Class, generic functions, and methods

- Methods are specific implementations of a generic function customized to its input
- The appropriate method is chosen by looking at the *class* of the input argument
- The methods available for a generic function can be obtained using the `methods()` function

```
methods("summary")
```

```
 [1] summary.aov                    summary.aovlist*
 [3] summary.aspell*                summary.check_packages_in_dir*
 [5] summary.connection             summary.data.frame
 [7] summary.Date                   summary.default
 [9] summary.ecdf*                  summary.factor
[11] summary.glm                    summary.hcl_palettes*
[13] summary.infl*                  summary.lm
[15] summary.loess*                 summary.manova
[17] summary.matrix                 summary.mlm*
[19] summary.nls*                   summary.packageStatus*
[21] summary.POSIXct                summary.POSIXlt
[23] summary.ppr*                   summary.prcomp*
[25] summary.princomp*              summary.proc_time
```

Class, generic functions, and methods

```r
methods("print") # similar but much longer list
```

```
 [1] print.acf*
 [2] print.activeConcordance*
 [3] print.AES*
 [4] print.anova*
 [5] print.aov*
 [6] print.aovlist*
 [7] print.ar*
 [8] print.Arima*
 [9] print.arima0*
[10] print.AsIs
[11] print.aspell*
[12] print.aspell_inspect_context*
[13] print.bibentry*
[14] print.Bibtex*
[15] print.browseVignettes*
[16] print.by
```

Class, generic functions, and methods

- All available methods for a given class can be similarly obtained

```r
methods(class = "lm")
```

```
 [1] add1           alias          anova          case.names     coerce        
 [6] confint        cooks.distance deviance       dfbeta         dfbetas       
[11] drop1          dummy.coef     effects        extractAIC     family        
[16] formula        hatvalues      influence      initialize     kappa         
[21] labels         logLik         model.frame    model.matrix   nobs          
[26] plot           predict        print          proj           qr            
[31] residuals      rstandard      rstudent       show           simulate      
[36] slotsFromS3    summary        variable.names vcov          
see '?methods' for accessing help and source code
```

Class, generic functions, and methods

- The name of a specific method appears to have the form `generic.class`
- However, one should always call the generic function, not the method directly
- This is not OK:

```
summary.lm(fm1)
```

- Instead, use

```
summary(fm1)
```

- In fact, many methods cannot be called directly because they are "hidden"

Class, generic functions, and methods

- This is a form of *Object Oriented Programming* (OOP) in R
- Python also has OOP, but
  - Methods are usually tied to a class, not a *generic* function
  - One notable exception is the `__str__()` method, which is used by `print()`

Getting help

- R has an extensive collection of functions (even more if we include add-on packages)
- It is impossible for anyone to know them all, or remember details
- Fortunately, R also has an excellent help system

Getting help

- R has an extensive collection of functions (even more if we include add-on packages)
- It is impossible for anyone to know them all, or remember details
- Fortunately, R also has an excellent help system
- Every function and dataset in R (and add-on packages) must be documented
- The documentation can be accessed by `help(name)` or `?name`
- For example: `help(seq)`, `help(summary)`, etc.
- A more general (but limited) search can be performed using `help.search("search-string")`

Getting help

- How the help is shown depends on the *interface* being used
- RStudio has a separate help tab (which also allows searching)

Getting help

- How the help is shown depends on the *interface* being used
- RStudio has a separate help tab (which also allows searching)
- However, before using the help system, you should know how methods are documented

Help for generic functions and methods

- Generic functions and methods are distinct functions
- They often have different help pages
- In fact, many add-on packages define new methods for generics in another package
- These are always documented in a separate help page

Help for generic functions and methods

- To get help for the generic function `summary()`, type `help(summary)`
- To get help for the `summary()` method for "matrix" objects, type `help(summary.matrix)`
- To get help for the `summary()` method for "lm" objects, type `help(summary.lm)`
- The first two happen to be the same help page, but the third is different

Help for generic functions and methods

- To get help for the generic function `summary()`, type `help(summary)`
- To get help for the `summary()` method for "matrix" objects, type `help(summary.matrix)`
- To get help for the `summary()` method for "lm" objects, type `help(summary.lm)`
- The first two happen to be the same help page, but the third is different
- This is slightly confusing because you are **not** supposed to call `summary.lm()` directly
- More importantly, there may not actually be a `summary()` method for all classes
- For example, "list" objects are handled by a *fallback* method `summary.default()`

Help for generic functions and methods

- To get help for the generic function `summary()`, type `help(summary)`
- To get help for the `summary()` method for "matrix" objects, type `help(summary.matrix)`
- To get help for the `summary()` method for "lm" objects, type `help(summary.lm)`
- The first two happen to be the same help page, but the third is different
- This is slightly confusing because you are **not** supposed to call `summary.lm()` directly
- More importantly, there may not actually be a `summary()` method for all classes
- For example, "list" objects are handled by a *fallback* method `summary.default()`
- The list of available methods are obtained by `methods("summary")` as shown earlier
- All these should have a corresponding help page

Help for generic functions and methods

- The system we described is called "S3" (short for "S version 3")
- The documentation refers to specific methods implemented using this system as "S3 methods"
- To make things more complicated, there are other systems of defining classes and methods
- We will skip the details of these for now

Replacement Functions

- R functions are not allowed to modify its arguments

Replacement Functions

- R functions are not allowed to modify its arguments
- Consider a function that sets negative inputs to 0

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise.} \end{cases}$$

Replacement Functions

- R functions are not allowed to modify its arguments
- Consider a function that sets negative inputs to 0

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise.} \end{cases}$$

- We will refer to this as the *ReLU* function

The Scalar ReLU Function in R

```r
sReLU <- function(u) {
    if (u < 0) u = 0
    u
}
```

The Scalar ReLU Function in R

```r
sReLU <- function(u) {
    if (u < 0) u = 0
    u
}
```

```r
x <- -5
sReLU(x)
```

```
[1] 0
```

```r
x
```

```
[1] -5
```

The Scalar ReLU Function in R

```r
sReLU <- function(u) {
    if (u < 0) u = 0
    u
}
```

```r
x <- -5
y <- sReLU(x)
```

The Scalar ReLU Function in R

```r
sReLU <- function(u) {
    if (u < 0) u = 0
    u
}
```

```r
x <- -5
y <- sReLU(x)
```

```r
x
```

```
[1] -5
```

```r
y
```

```
[1] 0
```

The Scalar ReLU Function in Python

```python
def sReLU(u):
    if u < 0:
        u = 0
    return u
```

The Scalar ReLU Function in Python

```python
def sReLU(u):
    if u < 0:
        u = 0
    return u
```

```python
x = -5
y = sReLU(x)
```

The Scalar ReLU Function in Python

```python
def sReLU(u):
    if u < 0:
        u = 0
    return u
```

```python
x = -5
y = sReLU(x)
```

```python
x
```

```
-5
```

```python
y
```

```
0
```

The Vectorized ReLU Function in Python

```
def vReLU(u):
    for i in range(len(u)):
        if u[i] < 0:
            u[i] = 0
    return u
```

The Vectorized ReLU Function in Python

```
def vReLU(u):
    for i in range(len(u)):
        if u[i] < 0:
            u[i] = 0
    return u
```

```
from numpy import *
x = random.normal(size = 10)
y = x
x
```

```
array([-1.2993973 , -1.11465781, -0.31597779, -1.29471625,  0.38419193,
        0.31589083,  1.17404954,  0.06097466,  0.33193044,  0.62444228])
```

```
y
```

```
array([-1.2993973 , -1.11465781, -0.31597779, -1.29471625,  0.38419193,
        0.31589083,  1.17404954,  0.06097466,  0.33193044,  0.62444228])
```

The Vectorized ReLU Function in Python

```
z = vReLU(y)
```

The Vectorized ReLU Function in Python

```
z = vReLU(y)
```

```
z
```

```
array([0.        , 0.        , 0.        , 0.        , 0.38419193,
       0.31589083, 1.17404954, 0.06097466, 0.33193044, 0.62444228])
```

The Vectorized ReLU Function in Python

```
z = vReLU(y)
```

```
z
```

```
array([0.        , 0.        , 0.        , 0.        , 0.38419193,
       0.31589083, 1.17404954, 0.06097466, 0.33193044, 0.62444228])
```

```
y
```

```
array([0.        , 0.        , 0.        , 0.        , 0.38419193,
       0.31589083, 1.17404954, 0.06097466, 0.33193044, 0.62444228])
```

The Vectorized ReLU Function in Python

```
z = vReLU(y)
```

```
z
```

```
array([0.        , 0.        , 0.        , 0.        , 0.38419193,
       0.31589083, 1.17404954, 0.06097466, 0.33193044, 0.62444228])
```

```
y
```

```
array([0.        , 0.        , 0.        , 0.        , 0.38419193,
       0.31589083, 1.17404954, 0.06097466, 0.33193044, 0.62444228])
```

```
x
```

```
array([0.        , 0.        , 0.        , 0.        , 0.38419193,
       0.31589083, 1.17404954, 0.06097466, 0.33193044, 0.62444228])
```

The Vectorized ReLU Function in R

```
vReLU <- function(u) {
    for (i in seq_len(length(u))) {
        if (u[i] < 0)
            u[i] <- 0
    }
    return(u)
}
```

The Vectorized ReLU Function in R

```
vReLU <- function(u) {
    for (i in seq_len(length(u))) {
        if (u[i] < 0)
            u[i] <- 0
    }
    return(u)
}
```

```
x <- rnorm(10)
y <- x
```

The Vectorized ReLU Function in R

```
vReLU <- function(u) {
    for (i in seq_len(length(u))) {
        if (u[i] < 0)
            u[i] <- 0
    }
    return(u)
}
```

```
x <- rnorm(10)
y <- x
```

```
x
```

```
 [1] -0.5915339 -0.1111574  1.5757390 -1.5187443 -0.3405556  1.4324608  1.3858898
 [8]  0.8068037 -0.6983363 -0.8147241
```

```
y
```

```
 [1] -0.5915339 -0.1111574  1.5757390 -1.5187443 -0.3405556  1.4324608  1.3858898
 [8]  0.8068037 -0.6983363 -0.8147241
```

The Vectorized ReLU Function in R

```r
z <- vReLU(y)
z
```

```
[1] 0.0000000 0.0000000 1.5757390 0.0000000 0.0000000 1.4324608 1.3858898 0.8068037
[9] 0.0000000 0.0000000
```

The Vectorized ReLU Function in R

```r
z <- vReLU(y)
z
```

```
[1] 0.0000000 0.0000000 1.5757390 0.0000000 0.0000000 1.4324608 1.3858898 0.8068037
[9] 0.0000000 0.0000000
```

```r
y
```

```
[1] -0.5915339 -0.1111574  1.5757390 -1.5187443 -0.3405556  1.4324608  1.3858898
[8]  0.8068037 -0.6983363 -0.8147241
```

```r
x
```

```
[1] -0.5915339 -0.1111574  1.5757390 -1.5187443 -0.3405556  1.4324608  1.3858898
[8]  0.8068037 -0.6983363 -0.8147241
```

Alternative: Direct Assignment

- We can instead use direct assignment to a 'subset'

```
y[y < 0] <- 0
```

Alternative: Direct Assignment

- We can instead use direct assignment to a 'subset'

```r
y[y < 0] <- 0
```

```r
x
```

```
[1] -0.5915339 -0.1111574  1.5757390 -1.5187443 -0.3405556  1.4324608  1.3858898
[8]  0.8068037 -0.6983363 -0.8147241
```

```r
y
```

```
[1] 0.0000000 0.0000000 1.5757390 0.0000000 0.0000000 1.4324608 1.3858898 0.8068037
[9] 0.0000000 0.0000000
```

Functional programming

- R generally follows a *functional programming* paradigm
- Among other things, this says that functions should not modify its arguments

Functional programming

- R generally follows a *functional programming* paradigm
- Among other things, this says that functions should not modify its arguments
- This is a key difference between Python and R

Functional programming

- R generally follows a *functional programming* paradigm
- Among other things, this says that functions should not modify its arguments
- This is a key difference between Python and R
- One consequence: R needs an *unusual* approach when it needs to modify objects

Example: Modifying Names of a Data Frame

```r
d <- data.frame(1, rnorm(5), rexp(5))
names(d)
```

```
[1] "X1"       "rnorm.5." "rexp.5."
```

```r
d
```

```
  X1   rnorm.5.   rexp.5.
1  1  0.5137902 2.3629231
2  1 -1.2638761 0.3281384
3  1  2.4951138 2.8728996
4  1 -1.7135097 0.7648846
5  1  0.2733856 0.5005642
```

- Default names are not very nice

Example: Modifying Names of a Data Frame

- Want to change the names to `"Constant"`, `"Normal"`, `"Exponential"`
- Possible solution using the `setNames()` function

```
setNames(d, c("Constant", "Normal", "Exponential"))
```

```
  Constant      Normal Exponential
1        1   0.5137902   2.3629231
2        1  -1.2638761   0.3281384
3        1   2.4951138   2.8728996
4        1  -1.7135097   0.7648846
5        1   0.2733856   0.5005642
```

Example: Modifying Names of a Data Frame

- But names of `d` are not modified by this

```
d
```

```
  X1   rnorm.5.   rexp.5.
1  1  0.5137902 2.3629231
2  1 -1.2638761 0.3281384
3  1  2.4951138 2.8728996
4  1 -1.7135097 0.7648846
5  1  0.2733856 0.5005642
```

Example: Modifying Names of a Data Frame

- But names of `d` are not modified by this

```
d
```

```
  X1   rnorm.5.   rexp.5.
1  1  0.5137902 2.3629231
2  1 -1.2638761 0.3281384
3  1  2.4951138 2.8728996
4  1 -1.7135097 0.7648846
5  1  0.2733856 0.5005642
```

- Best we can hope for

```
d <- setNames(d, c("Constant", "Normal", "Exponential"))
```

Example: Modifying Names of a Data Frame

- In fact, the **dplyr** package has a more convenient version of this approach

```
d <- dplyr::rename(d, Constant = X1, Normal = rnorm.5., Exponential = rexp.5.)
d
```

```
  Constant     Normal Exponential
1        1  0.5137902   2.3629231
2        1 -1.2638761   0.3281384
3        1  2.4951138   2.8728996
4        1 -1.7135097   0.7648846
5        1  0.2733856   0.5005642
```

Example: Modifying Names of a Data Frame

- The classic R Alternative: Replacement Functions

```r
names(d) <- c("Constant", "Normal", "Exponential")
d
```

```
  Constant      Normal Exponential
1        1  0.5137902   2.3629231
2        1 -1.2638761   0.3281384
3        1  2.4951138   2.8728996
4        1 -1.7135097   0.7648846
5        1  0.2733856   0.5005642
```

Replacement Functions

- Other similar examples:

```
y[ y < 0 ] <- 0
class(x)  <- NULL
d$Normal[ d$Normal < 0 ] <- 0
attr(x, "name") <- value
```

- Common feature: "complex" expression on the LHS of the assignment