R 4 hackers

Hello World

… that is, *Data Science Hello World*.

We got some data...

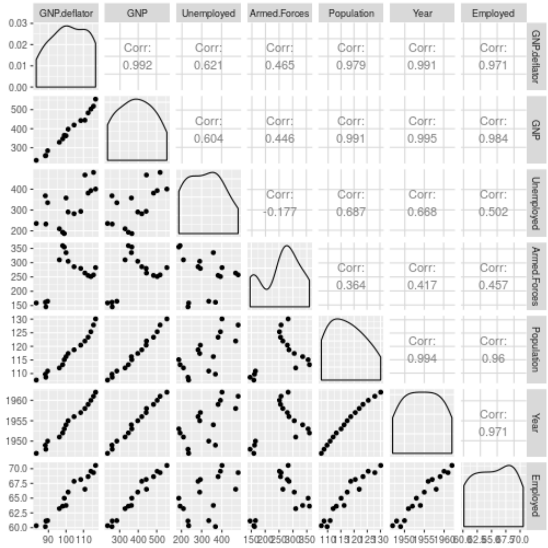
Sure, first we ALWAYS do some data exploration.

```
data(longley)
head(longley)
```

```
     GNP.deflator     GNP Unemployed Armed.Forces Population Year Employed
1947         83.0 234.289      235.6        159.0    107.608 1947   60.323
1948         88.5 259.426      232.5        145.6    108.632 1948   61.122
1949         88.2 258.054      368.2        161.6    109.773 1949   60.171
1950         89.5 284.599      335.1        165.0    110.929 1950   61.187
1951         96.2 328.975      209.9        309.9    112.075 1951   63.221
1952         98.1 346.999      193.2        359.4    113.270 1952   63.639
```

Sure, first we ALWAYS visualize...

```
ggpairs(longley)
```



But then: Hello World!

Linear models.

```
fit <- lm(Employed ~ GNP, longley)
```

Now what can we do with this thing returned by lm?

Print it

```
# equivalently: print(fit)
fit
```

```

Call:
lm(formula = Employed ~ GNP, data = longley)

Coefficients:
(Intercept)          GNP
   51.84359      0.03475
```

Output a summary

```
summary(fit)
```

```

Call:
lm(formula = Employed ~ GNP, data = longley)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.77958 -0.55440 -0.00944  0.34361  1.44594 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 51.843590   0.681372   76.09  < 2e-16 ***
GNP          0.034752   0.001706   20.37 8.36e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6566 on 14 degrees of freedom
Multiple R-squared:  0.9674,	Adjusted R-squared:  0.965 
F-statistic: 415.1 on 1 and 14 DF,  p-value: 8.363e-12
```

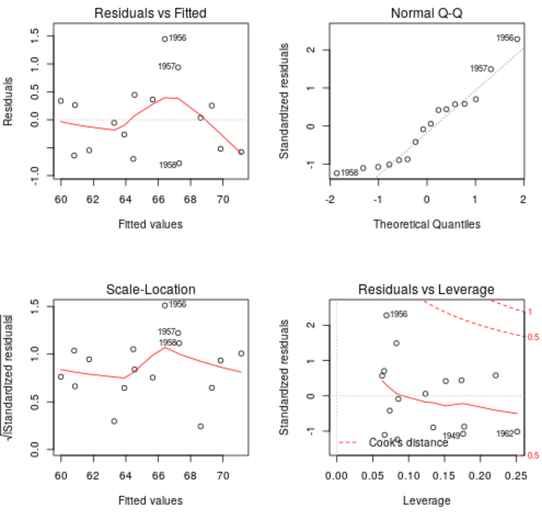
Use it to make predictions

```r
predict(fit, newdata = data.frame(GNP = c(222, 223)))
```

```
       1        2 
59.55860 59.59335 
```

We can plot it...

```
par(mfrow=c(2,2))
plot(fit)
```




```
dev.off()
```

```
null device
          1
```

And even do some fancy stuff.

Like extracting the Akaike Information Criterion …

```
extractAIC(fit)
```

```
[1]   2.00000 -11.59718
```

Or getting confidence intervals for coefficients.

```
confint(fit)
```

```
                 2.5 %      97.5 %
(Intercept) 50.38219297 53.30498660
GNP          0.03109391  0.03841068
```

Let's see what other objects we can print, plot, get a summary of!

Data frames (1)

```
df <- data.frame(x = 1:8, y = cumsum(rnorm(8)))
df
```

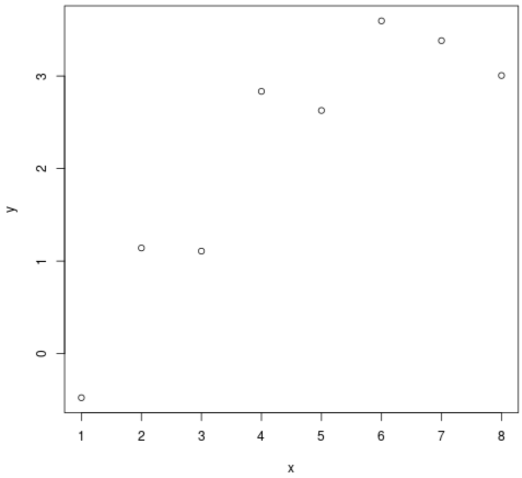
```
  x          y
1 1 -0.4763357
2 2  1.1415539
3 3  1.1076525
4 4  2.8341954
5 5  2.6276654
6 6  3.5959169
7 7  3.3822216
8 8  3.0054988
```

```
summary(df)
```

```
       x              y
 Min.   :1.00   Min.   :-0.4763
 1st Qu.:2.75   1st Qu.: 1.1331
 Median :4.50   Median : 2.7309
 Mean   :4.50   Mean   : 2.1523
 3rd Qu.:6.25   3rd Qu.: 3.0997
 Max.   :8.00   Max.   : 3.5959
```

Data frames (2)

```
plot(df)
```



Time series objects (1)

```r
ts <- ts(cumsum(round(rnorm(120), 2)), start = c(2004,12), frequency = 12)
ts
```

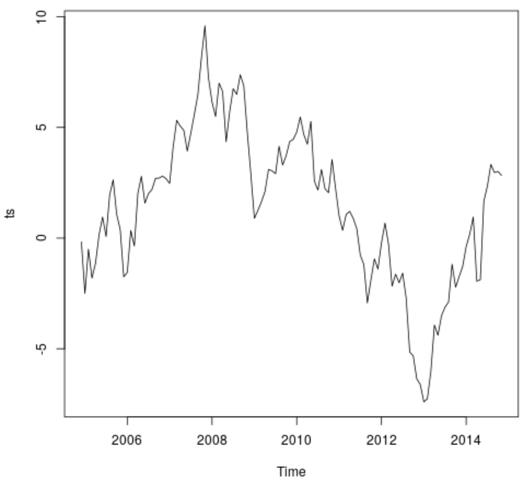
```
       Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov
2004
2005 -2.50 -0.50 -1.81 -1.14  0.14  0.95  0.07  1.94  2.63  1.07  0.38
2006 -1.55  0.34 -0.36  2.03  2.79  1.58  2.01  2.20  2.70  2.71  2.80
2007  2.47  4.13  5.32  5.06  4.87  3.93  4.74  5.59  6.49  8.15  9.59
2008  6.15  5.49  7.00  6.62  4.35  5.71  6.75  6.49  7.38  6.88  4.76
2009  0.89  1.25  1.64  2.10  3.10  3.04  2.91  4.15  3.30  3.72  4.36
2010  4.79  5.47  4.68  4.24  5.27  2.57  2.17  3.10  2.23  2.05  3.55
2011  1.00  0.35  1.07  1.21  0.89  0.43 -0.77 -1.19 -2.93 -1.90 -0.94
2012 -0.22  0.68 -0.28 -2.17 -1.63 -2.02 -1.58 -2.72 -5.15 -5.32 -6.36
2013 -7.40 -7.26 -6.01 -3.93 -4.39 -3.51 -3.12 -2.89 -1.18 -2.22 -1.74
2014 -0.38  0.18  0.95 -1.95 -1.88  1.67  2.35  3.33  2.96  3.01  2.83
       Dec
2004 -0.17
2005 -1.75
2006  2.68
2007  7.23
2008  2.90
2009  4.46
2010  2.24
2011 -1.40
2012 -6.62
2013 -1.28
2014
```

```r
summary(ts)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -7.400  -1.182   1.805   1.376   3.772   9.590
```

Time series objects (2)

```
plot(ts)
```



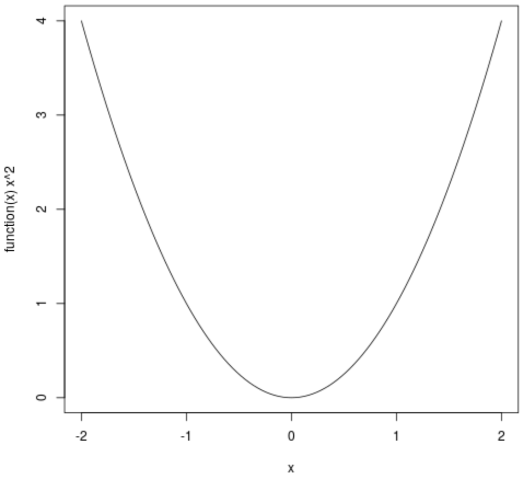
Things we can get a summary of (but not plot) …

```r
m <- matrix(1:10, nrow = 2)
summary(m)
```

```
       V1              V2              V3              V4       
 Min.   :1.00   Min.   :3.00   Min.   :5.00   Min.   :7.00  
 1st Qu.:1.25   1st Qu.:3.25   1st Qu.:5.25   1st Qu.:7.25  
 Median :1.50   Median :3.50   Median :5.50   Median :7.50  
 Mean   :1.50   Mean   :3.50   Mean   :5.50   Mean   :7.50  
 3rd Qu.:1.75   3rd Qu.:3.75   3rd Qu.:5.75   3rd Qu.:7.75  
 Max.   :2.00   Max.   :4.00   Max.   :6.00   Max.   :8.00  
       V5       
 Min.   : 9.00  
 1st Qu.: 9.25  
 Median : 9.50  
 Mean   : 9.50  
 3rd Qu.: 9.75  
 Max.   :10.00  
```

Things we can plot (but not get a summary of)

```
plot(function(x) x^2, from = -2, to = 2)
```



What is going on?

- R has several OO systems (on top of base (internal) objects)
- the oldest and most widely used is S3

S3

- generic function OO (instead of the more common message-passing OO)
- methods belong to (generic) functions, not to classes!
- *UseMethod()* performs method dispatch

```
print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x5618623538c0>
<environment: namespace:base>
```

```
UseMethod
```

```
function (generic, object)  .Primitive("UseMethod")
```

In S3, there are no formal class definitions

```r
# a bike constructor
bike <- function(type, color) {structure (list(type = type, color = color), class = 'bike')}

# create an instance of class bike
mybike <- bike('cyclocross', 'green')
class(mybike)
```

```
[1] "bike"
```

```r
# still prints like a list
mybike
```

```
$type
[1] "cyclocross"

$color
[1] "green"

attr(,"class")
[1] "bike"
```

Define a print method for bikes

```
# methods are called <funcname>.<classname>
print.bike <- function(b) {print(paste0('a ', b$color, ' ', b$type, ' bike'))}
mybike
```

```
[1] "a green cyclocross bike"
```

With S3, you could shoot yourself in the foot...

... or just ... don't.

```
# what if we just change the class
class(mybike) <- 'lm'
# and then print it
mybike
```

```

Call:
NULL

No coefficients
```

```
# let's undo this ASAP ;-)
class(mybike) <- 'bike'
# and nothing's broken
mybike
```

```
[1] "a green cyclocross bike"
```

S3 ”inheritance” is informal as well

```
ebike <- function(type, color) {
  parent <- bike(type, color)
  structure (c(unclass(parent), motor = TRUE), class = c('ebike', class(parent)))}

theotherpersonsbike <-ebike('mountain', 'red')
class(theotherpersonsbike)
```

```
[1] "ebike" "bike"
```

```
theotherpersonsbike
```

```
[1] "a red mountain bike"
```

```
print.ebike <- function(b) {
  ptext <- NextMethod()
  print('... with a motor!')
}
theotherpersonsbike
```

```
[1] "a red mountain bike"
[1] "... with a motor!"
```

S3: Create your own generic

```r
# create a generic function that calls UseMethod to do the dispatching
speed_up <- function(object, ...) UseMethod("speed_up")

# create an implementation for our bike class
speed_up.bike <- function(object, target_speed) {
  accelerate_until_at(target_speed)
}

speed_up(mybike, target_speed = 33)
```

```
[1] "Now accelerating to 33 km/h"
```

```r
# also of course create an implementation for the e-bike
speed_up.ebike <- function(object, target_speed) {
  adjusted_speed <- ifelse(target_speed <= 25, target_speed, 25) # ... you've seen that coming
  accelerate_until_at(adjusted_speed)
}

speed_up(theotherpersonsbike, target_speed = 33)
```

```
[1] "Now accelerating to 25 km/h"
```

S3: What happens if there is no implementation for a class?

The default method for a function will be used.

Remember confint() from above?

```
# these implementations exist for confint:
methods('confint')
```

```
[1] confint.default   confint.fracdiff* confint.glm*      confint.lm
[5] confint.multinom* confint.nls*
see '?methods' for accessing help and source code
```

```
data("lynx")
fit <- auto.arima(lynx)
# same as explicitly calling confint.default
confint(fit)
```

```
                 2.5 %        97.5 %
ar1          1.1491419    1.53501010
ar2         -0.8307363   -0.51688060
ma1         -0.4498853    0.04440014
ma2         -0.4713138   -0.04147972
intercept 1285.8372685 1802.97062435
```

OO wrap-up: Other systems

- S4:
  - more formal than S3 (formal class definitions)
  - but methods still belong to functions, not classes
- Reference classes (RC):
  - methods belong to objects, not functions
  - objects are mutable (the usual R copy-on-modify semantics do not apply)

On to...

functional programming!

The Magic Three: map, fold, and filter

Magic Three in Haskell:

- map: map a function over a list of elements

```
λ: map (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

- $filter$: filter a list of elements according to some predicate

```
λ: filter even [1..10]
[2,4,6,8,10]
```

- $fold$: combine values recursively (a.k.a. $reduce$ (Clojure, Java, Python…), $apply$ (Scheme, …))

```
λ: foldl (+) 0 [1..10]
55
```

Mapping in R (1): meet the APPLY family

- apply, lapply, sapply, vapply, mapply, tapply … ough!
- Basic question: What data structure(s) am I working with?
  - one-dimensional?
  - more than one dimension?
  - more than one data structure?

The apply family (1): just … apply

Use with more-than-one-dimensional data structures: data.frame, matrix, array

```
m <- matrix(1:12, nrow = 3, ncol = 4)
m
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
# apply mean to the columns
apply(m, 2, mean)
```

```
[1]  2  5  8 11
```

```
# apply mean to the rows
apply(m, 1, mean)
```

```
[1] 5.5 6.5 7.5
```

The apply family (2): lapply and friends

Use with one-dimensional stuff (list, vector)

- lapply: outputs a list

```
mychars <- c("a", "b"); str(lapply(mychars, toupper))
```

```
List of 2
 $ : chr "A"
 $ : chr "B"
```

- sapply: simplifies the result

```
mychars <- c("a", "b"); str(sapply(mychars, toupper))
```

```
 Named chr [1:2] "A" "B"
 - attr(*, "names")= chr [1:2] "a" "b"
```

- vapply: returns requested type

```
mychars <- c("a", "b"); str(vapply(mychars, utf8ToInt, integer(1)))
```

```
 Named int [1:2] 97 98
 - attr(*, "names")= chr [1:2] "a" "b"
```

Aside: What's the problem with sapply? (1)

```
# a list of 3
l1 <-list(
  col1 = "a",
  col2 = "b",
  col3 = c("c", "d")
)
str(l1)
```

```
List of 3
 $ col1: chr "a"
 $ col2: chr "b"
 $ col3: chr [1:2] "c" "d"
```

```
# a list of 2
l2 <- l1[1:2]
str(l2)
```

```
List of 2
 $ col1: chr "a"
 $ col2: chr "b"
```

Aside: What's the problem with sapply? (2)

```
# upper case everything
u1 <- sapply(l1, toupper)
# result is still a list!
str(u1)
```

```
List of 3
 $ col1: chr "A"
 $ col2: chr "B"
 $ col3: chr [1:2] "C" "D"
```

```
u2 <- sapply(l2, toupper)
# result is a vector!
str(u2)
```

```
 Named chr [1:2] "A" "B"
 - attr(*, "names")= chr [1:2] "col1" "col2"
```

Mapping in R (2): Map

Yes, we have them in R, too:

- Map
- Filter
- Reduce
- (plus Find, Negate, and Position)

Redoing the Magic Three, in R

Map

```
# same as lapply(1:10, function(x) x+1), but see order of args!
m <- Map(function(x) x+1, 1:10)
```

Filter

```
# same as lapply(1:10, function(x) x+1), but see order of args!
Filter(function(x) x %% 2 == 0, 1:10)
```

```
[1]  2  4  6  8 10
```

Reduce

```
# same as lapply(1:10, function(x) x+1), but see order of args!
Reduce(`+`, 1:10)
```

```
[1] 55
```

Mapping in R (3): Typesafe mapping with purrr

```
m <- map_dbl(1:10, function(x) x+1)
# but
m2 <- map_chr(letters[1:3], toupper)
```

So … what is purrr?

- functional programming package for R, by Hadley Wickham
- not just the “big three”…

Functional programming with purrr (examples)

Too verbose?

```
m <- map_dbl(1:10, function(x) x+1)
```

How about partial application:

```
m <- map_dbl(1:10, partial(`+`,1))
```

Or, how about function composition?

```
inttolower <- compose(tolower, intToUtf8)
inttolower(65:68)
```

```
[1] "abcd"
```

OK. Time for the real internals ...

We've seen objects, we've seen functions.

But what's R *basically* made of?

Object types: class(), typeof(), mode() ... oh my!

Just wanna use R? Use *class()*:

```
myfunc <- function(x) x + 1
tests1 <- c(`<-`, `if`, `[`, length, c, sum, nrow, eval, myfunc)
sapply(tests1, class)
```

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

For the user, these are all *functions*. Even though they do such different things as

- assignment (x <- 1)
- constructing new objects (x <- c(1,2))
- branching (if)

typeof() tells about the internal object type:

```
tests1 <- c(`<-`, `if`, `[`, length, c, sum, nrow, eval, myfunc)
sapply(tests1, typeof)
```

```
[1] "special" "special" "special" "builtin" "builtin" "builtin" "closure"
[8] "closure" "closure"
```

So we have three different corresponding object types:

- specials,
- builtins, and
- closures.

Closures (1)

Every user-defined function is a closure.

With closures, we can conveniently print the source code on the console:

```
nrow
```

```
function (x)
dim(x)[1L]
<bytecode: 0x558eaeba6f60>
<environment: namespace:base>
```

Closures (2)

Closures have formals, a body, and an associated environment.

```
c(formals(myfunc), body(myfunc), environment(myfunc))
```

```
$x


[[2]]
x + 1

[[3]]
<environment: R_GlobalEnv>
```

Let's try this with eval!

(Remember, this was a closure, too.)

```
body(eval)
```

```
.Internal(eval(expr, envir, enclos))
```

Oops!

So, for the .Internals...

For .Internal and .Primitive functions (the “builtins” above),

$R_source/src/main/names.c

contains the mapping to the corresponding C function:

```
{"ls",          do_ls,          1,      11,     3,      {PP_FUNCALL, PREC_FN,   0}},
{"typeof",      do_typeof,      1,      11,     1,      {PP_FUNCALL, PREC_FN,   0}},
{"eval",        do_eval,        0,      211,    3,      {PP_FUNCALL, PREC_FN,   0}},
{"returnValue",   do_returnValue,0,     11,     1,      {PP_FUNCALL, PREC_FN,   0}},
{"sys.parent",  do_sys,         1,      11,     -1,     {PP_FUNCALL, PREC_FN,   0}},
```

And this is (the beginning of) do_eval

```
SEXP attribute_hidden do_eval(SEXP call, SEXP op, SEXP args, SEXP rho)
{
    SEXP encl, x, xptr;
    volatile SEXP expr, env, tmp;

    int frame;
    RCNTXT cntxt;

    checkArity(op, args);
    expr = CAR(args);
    env = CADR(args);
    encl = CADDR(args);
    SEXPTYPE tEncl = TYPEOF(encl);
```

Notice something?

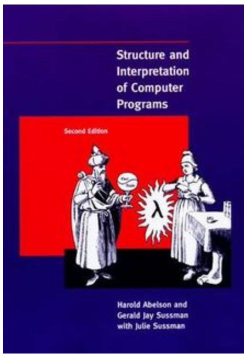
Yes. There's some LISP in there

Not just the CARs, CADRs, CADDRs…

… S-expressions…

… the whole idea of environments and closures in current R is modeled after Lisp.

(No time now and here, but there's always SICP to read up on environments etc.)



What's so special about specials?

Specials get their arguments passed in quoted and decide themselves when to evaluate what.

What do you think will happen here?

```
# no confint.bike defined for bikes -> confint.default will get called
# but confint.default needs some other methods that do not exist
if(1 > 0) 1 else confint(mybike)
```

```
[1] 1
```

Just so you believe me an error *would* get generated if confint(mybike) were called.

```
confint(mybike)
```

```
Error in UseMethod("vcov"): no applicable method for 'vcov' applied to an object of class "bike"
```

I promise you (1)

Here we have a user-defined function, f. What will happen?

```
f <- function(exp1, exp2) {
  exp1
}

f(confint(mybike), f)
```

```
Error in UseMethod("vcov"): no applicable method for 'vcov' applied to an object of class "bike"
```

```
f(123, confint(mybike))
```

```
[1] 123
```

Closures evaluate their arguments lazily (by need).

I promise you (2)

As users, we can create *promises*, too:

```
# normal assignment - this can't work
x <- confint(mybike)
```

```
Error in UseMethod("vcov"): no applicable method for 'vcov' applied to an object of class "bike"
```

```
# promise to evaluate when needed
# this works without error
delayedAssign('x', confint(mybike))

# here it gets evaluated
x
```

```
Error in UseMethod("vcov"): no applicable method for 'vcov' applied to an object of class "bike"
```

I promise you (3)

… this could go on for quite some time … but ;-)

Thanks a lot for your attention!

:-)