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)

<table>
<thead>
<tr>
<th>Year</th>
<th>GNP.deflator</th>
<th>GNP</th>
<th>Unemployed</th>
<th>Armed.Forces</th>
<th>Population</th>
<th>Year</th>
<th>Employed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1947</td>
<td>83.0</td>
<td>234.289</td>
<td>235.6</td>
<td>159.0</td>
<td>107.608</td>
<td>1947</td>
<td>60.323</td>
</tr>
<tr>
<td>1948</td>
<td>88.5</td>
<td>259.426</td>
<td>232.5</td>
<td>145.6</td>
<td>108.632</td>
<td>1948</td>
<td>61.322</td>
</tr>
<tr>
<td>1949</td>
<td>88.2</td>
<td>258.854</td>
<td>368.2</td>
<td>161.6</td>
<td>109.773</td>
<td>1949</td>
<td>60.171</td>
</tr>
<tr>
<td>1950</td>
<td>89.5</td>
<td>284.599</td>
<td>355.1</td>
<td>165.0</td>
<td>110.929</td>
<td>1950</td>
<td>61.187</td>
</tr>
<tr>
<td>1951</td>
<td>96.2</td>
<td>328.975</td>
<td>299.9</td>
<td>309.9</td>
<td>112.975</td>
<td>1951</td>
<td>63.221</td>
</tr>
<tr>
<td>1952</td>
<td>98.1</td>
<td>346.999</td>
<td>193.2</td>
<td>359.4</td>
<td>113.270</td>
<td>1952</td>
<td>63.639</td>
</tr>
</tbody>
</table>
Sure, first we ALWAYS visualize...
ggpairs(longley)
But then: Hello World!

Linear models.

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

Now what can we do with this thing returned by lm?
# equivalently: print(fit)

```r
fit
```

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

Coefficients:
```
              Estimate Std. Error t value Pr(>|t|) 
(Intercept)  51.84359    2.27805  22.883  < 2e-16 
         GNP  0.03475    0.01270   2.734  0.02077 *
```


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

1 2
59.55860 59.59335
We can plot it...

```r
par(mfrow=c(2,2))
plot(fit)
dev.off()
null device 1
```
And even do some fancy stuff.

Like extracting the Akaike Information Criterion ...

```r
eextractAIC(fit)
```

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

Or getting confidence intervals for coefficients.

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.4763357</td>
</tr>
<tr>
<td>2</td>
<td>1.1415539</td>
</tr>
<tr>
<td>3</td>
<td>1.1076525</td>
</tr>
<tr>
<td>4</td>
<td>2.8341954</td>
</tr>
<tr>
<td>5</td>
<td>2.6276654</td>
</tr>
<tr>
<td>6</td>
<td>3.5938169</td>
</tr>
<tr>
<td>7</td>
<td>3.3822216</td>
</tr>
<tr>
<td>8</td>
<td>3.0054988</td>
</tr>
</tbody>
</table>

summary(df)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>Min</td>
<td>Min.</td>
</tr>
<tr>
<td>1.00</td>
<td>-0.4763</td>
</tr>
<tr>
<td>1st Qu.</td>
<td>2.75</td>
</tr>
<tr>
<td>Median</td>
<td>4.50</td>
</tr>
<tr>
<td>Mean</td>
<td>4.50</td>
</tr>
<tr>
<td>3rd Qu.: 6.25</td>
<td></td>
</tr>
<tr>
<td>Max.</td>
<td>8.00</td>
</tr>
<tr>
<td>1.1331</td>
<td>3.5959</td>
</tr>
</tbody>
</table>

```
plot(df)
```r
# Time series objects

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

ts

summary(ts)
```

<table>
<thead>
<tr>
<th></th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>2.50</td>
<td>-0.50</td>
<td>-1.81</td>
<td>-1.14</td>
<td>0.14</td>
<td>0.95</td>
<td>0.07</td>
<td>1.94</td>
<td>2.63</td>
<td>1.07</td>
<td>0.38</td>
</tr>
<tr>
<td>2005</td>
<td>-1.55</td>
<td>0.34</td>
<td>-0.36</td>
<td>2.03</td>
<td>2.79</td>
<td>1.58</td>
<td>2.01</td>
<td>2.20</td>
<td>2.70</td>
<td>2.71</td>
<td>2.80</td>
</tr>
<tr>
<td>2006</td>
<td>2.47</td>
<td>4.13</td>
<td>5.32</td>
<td>5.06</td>
<td>4.87</td>
<td>3.93</td>
<td>4.74</td>
<td>5.59</td>
<td>6.49</td>
<td>8.15</td>
<td>9.59</td>
</tr>
<tr>
<td>2007</td>
<td>6.15</td>
<td>5.49</td>
<td>7.00</td>
<td>6.62</td>
<td>4.35</td>
<td>5.71</td>
<td>6.75</td>
<td>6.49</td>
<td>7.38</td>
<td>6.88</td>
<td>4.76</td>
</tr>
<tr>
<td>2008</td>
<td>0.89</td>
<td>1.25</td>
<td>1.64</td>
<td>2.10</td>
<td>3.10</td>
<td>3.04</td>
<td>2.91</td>
<td>4.15</td>
<td>3.30</td>
<td>3.72</td>
<td>4.36</td>
</tr>
<tr>
<td>2009</td>
<td>4.79</td>
<td>5.47</td>
<td>4.68</td>
<td>4.24</td>
<td>5.27</td>
<td>2.57</td>
<td>2.17</td>
<td>3.18</td>
<td>2.23</td>
<td>2.05</td>
<td>3.55</td>
</tr>
<tr>
<td>2010</td>
<td>1.00</td>
<td>0.35</td>
<td>1.97</td>
<td>1.21</td>
<td>0.89</td>
<td>0.43</td>
<td>-0.77</td>
<td>-1.19</td>
<td>-2.93</td>
<td>-1.90</td>
<td>-0.94</td>
</tr>
<tr>
<td>2011</td>
<td>-0.22</td>
<td>0.68</td>
<td>-0.28</td>
<td>-2.17</td>
<td>-1.63</td>
<td>-2.02</td>
<td>-1.58</td>
<td>-2.72</td>
<td>5.15</td>
<td>5.32</td>
<td>-6.36</td>
</tr>
<tr>
<td>2013</td>
<td>-0.38</td>
<td>0.18</td>
<td>0.95</td>
<td>-1.95</td>
<td>-1.88</td>
<td>1.67</td>
<td>2.35</td>
<td>3.33</td>
<td>2.96</td>
<td>3.01</td>
<td>2.83</td>
</tr>
<tr>
<td>2014</td>
<td>Dec</td>
<td>0.17</td>
<td>0.15</td>
<td>0.75</td>
<td>2.68</td>
<td>7.23</td>
<td>2.90</td>
<td>4.46</td>
<td>2.24</td>
<td>1.40</td>
<td>6.62</td>
</tr>
</tbody>
</table>

```
summary(ts)
```

<table>
<thead>
<tr>
<th></th>
<th>Min.</th>
<th>1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-7.400</td>
<td>-1.182</td>
<td>1.805</td>
<td>1.376</td>
<td>3.772</td>
<td>9.590</td>
</tr>
</tbody>
</table>
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)
```

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min.</td>
<td>1.00</td>
<td>3.00</td>
<td>5.00</td>
<td>7.00</td>
</tr>
<tr>
<td>1st Qu.</td>
<td>1.25</td>
<td>3.25</td>
<td>5.25</td>
<td>7.25</td>
</tr>
<tr>
<td>Median</td>
<td>1.50</td>
<td>3.50</td>
<td>5.50</td>
<td>7.50</td>
</tr>
<tr>
<td>Mean</td>
<td>1.50</td>
<td>3.50</td>
<td>5.50</td>
<td>7.50</td>
</tr>
<tr>
<td>3rd Qu.</td>
<td>1.75</td>
<td>3.75</td>
<td>5.75</td>
<td>7.75</td>
</tr>
<tr>
<td>Max.</td>
<td>2.00</td>
<td>4.00</td>
<td>6.00</td>
<td>8.00</td>
</tr>
</tbody>
</table>

V5

<table>
<thead>
<tr>
<th></th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min.</td>
<td>9.00</td>
</tr>
<tr>
<td>1st Qu.</td>
<td>9.25</td>
</tr>
<tr>
<td>Median</td>
<td>9.50</td>
</tr>
<tr>
<td>Mean</td>
<td>9.50</td>
</tr>
<tr>
<td>3rd Qu.</td>
<td>9.75</td>
</tr>
<tr>
<td>Max.</td>
<td>10.00</td>
</tr>
</tbody>
</table>
Things we can plot (but not get a summary of)

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

```r
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"

# still prints like a list
mybike

$type
[1] "cyclocross"

$color
[1] "green"

attr(,"class")
[1] "bike"
```
Define a print method for bikes

```r
# 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.

```r
# 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"
```
```
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"

class(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

# 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"

# 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"
The default method for a function will be used.

Remember `confint()` from above?

```r
# 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
```
• 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
  
  ```
  \lambda: \text{map} (+1) \left[1..10\right]
  
  \left[2,3,4,5,6,7,8,9,10,11\right]
  ```

- **filter**: filter a list of elements according to some predicate
  
  ```
  \lambda: \text{filter even} \left[1..10\right]
  
  \left[2,4,6,8,10\right]
  ```

- **fold**: combine values recursively (a.k.a. *reduce* (Clojure, Java, Python...), *apply* (Scheme, ...))
  
  ```
  \lambda: \text{foldl} (+) 0 \left[1..10\right]
  
  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?
Use with more-than-one-dimensional data structures: data.frame, matrix, array

```r
m <- matrix(1:12, nrow = 3, ncol = 4)
m
```
```
[1,]   1   4   7  10
[2,]   2   5   8  11
[3,]   3   6   9  12
```

# apply mean to the columns
```r
apply(m, 2, mean)
```
```
[1]  2  5  8 11
```

# apply mean to the rows
```r
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
  ```r
  mychars <- c("a", "b"); str(lapply(mychars, toupper))
  List of 2
  $ : chr "A"
  $ : chr "B"
  ```

- `sapply`: simplifies the result
  ```r
  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
  ```r
  mychars <- c("a", "b"); str(vapply(mychars, utf8ToInt, integer(1)))
  Named int [1:2] 97 98
  - attr(*, "names")= chr [1:2] "a" "b"
  ```
# 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)

```r
# 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"
```
Yes, we have them in R, too:

- Map
- Filter
- Reduce
- (plus Find, Negate, and Position)
Redoing the Magic Three, in R

### Map

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

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

### Filter

```r
# 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

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

Reduce(`+`, 1:10)
```

```
[1] 55
```
So ... what is purrr?

- functional programming package for R, by Hadley Wickham
- not just the “big three”...
Too verbose?

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

How about partial application:

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

Or, how about function composition?

```r
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()`:

```r
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:

```r
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.
Every user-defined function is a closure.

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

```r
nrow

function (x)
dim(x)[1L]
<bytecode: 0x558eaeba6f60>
<environment: namespace:base>
```
Closures have formals, a body, and an associated environment.

```r
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.)

<table>
<thead>
<tr>
<th>body(eval)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Internal(eval(expr, envir, enclos))</td>
</tr>
</tbody>
</table>

Oops!
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

```c
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);
    SEXP_TYPE 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?

```r
# 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.

```r
confint(mybike)
```

```
Error in UseMethod("vcov"): no applicable method for 'vcov' applied to an object of class "bike"
```
Here we have a user-defined function, \( f \). What will happen?

```r
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"
```

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

```
[1] 123
```

Closures evaluate their arguments lazily (by need).
As users, we can create *promises*, too:

```r
# 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!

:-)