Session 7
Functions 1

R for Social Data Science

Jeffrey Ziegler, PhD

Assistant Professor in Political Science & Data Science
Trinity College Dublin

Fall 2022

Road map for today

Last time:

- Straight-line and branching programs
- Algorithms
- Conditional statements
- Loops and Iteration

This time:

- Decomposition and abstraction
- Arguments, environments
- Function definition and function call

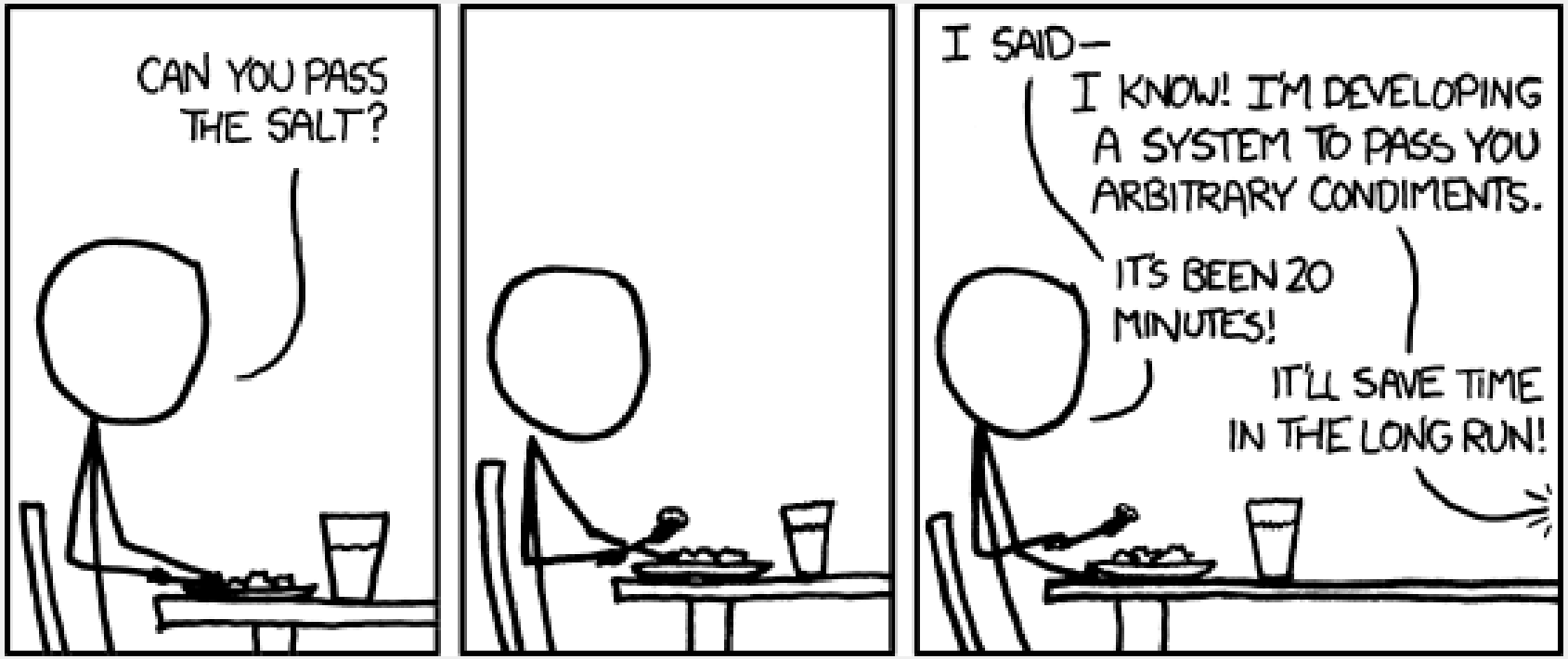
Decomposition and abstraction

Source: IKEA

Decomposition and abstraction

- So far:
  - built-in types
  - assignments
  - branching and looping constructs
- In principle, any problem can be solved just with those
- But a solution would be non-modual and hard-to-maintain
- Functions provide *decomposition* and *abstraction*

Functions



Source: xkcd

Functions in R

- Function call is the centerpiece of computation in R
- It involves function object and objects that are supplied as arguments
- Functions in R do not have side-effects (nonlocal modifications of input objects)
- In R we use function ‘function()‘ to create a function object
- Functions are also referred to as *closures* in some R documentation

```
<function_name> <- function(<arg_1>, <arg_2>, ..., <arg_n>) {
<function_body>
}
```

```
1 foo <- function(arg) {
2   # <function_body>
3 }
```

Function components

- Body (‘body()‘) - code inside the function
- List of arguments (‘formals()‘) - controls how function is called
- Environment/scope/namespace (‘environment()‘) - location of function’s definition and variables

Function components example

```
is_positive <- function(num) {
  if (num > 0) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}
body(is_positive)
```

```
{
if (num > 0) {
return(TRUE)
} else {
return(FALSE)
} }
```

```
formals(is_positive)
```

```
$num
```

```
environment(is_positive)
```

```
<environment: R_GlobalEnv>
```

Function call

- Function is executed until:
  - Either ‘return()‘ function is encountered
  - There are no more expressions to evaluate
- Function call always returns a value:
  - Argument of ‘return()‘ function call
  - Value of last expression if no ‘return()‘ (implicit return)
- Function can return only one object
  - But you can combine multiple R objects in a list

Function call example

```r
is_positive <- function(num){
  if (num > 0){
    res <- TRUE
  } else{
    res <- FALSE
  }
  return(res)
}
res_1 <- is_positive(5)
res_2 <- is_positive(-7)
print(res_1)
print(res_2)
```

```
[1] TRUE
[1] FALSE
```

Implicit return example

```
1  is_positive <- function(num){
2    if (num > 0){
3      res <- TRUE
4    } else{
5      res <- FALSE
6    }
7    res
8  }
9  res_1 <- is_positive(5)
10 res_2 <- is_positive(-7)
11 print(res_1)
12 print(res_2)
```

```
[1] TRUE
[1] FALSE
```

Implicit return example

```r
# While this function provides the same functionality as the two versions above
# This is an example of a bad programming style, return value is very unintuitive
is_positive <- function(num){
  if (num > 0){
    res <- TRUE
  } else{
    res <- FALSE
  }
}
res_1 <- is_positive(5)
res_2 <- is_positive(-7)
print(res_1)
print(res_2)
```

```
[1] TRUE
[1] FALSE
```

Function arguments

- *Arguments* provide a way of giving input to a function
- Arguments in function definition are *formal arguments*
- Arguments in function invocations are *actual arguments*
- When a function is invoked (called) arguments are matched and bound to local variable names
- R matches arguments in 3 ways:
  1. by exact name
  2. by partial name
  3. by position
- It is a good idea to only use unnamed (positional) for main (first one or two) arguments

Function arguments example

```
format_date <- function(day, month, year, reverse = TRUE){
  if (isTRUE(reverse)){
    formatted <- paste(as.character(year), as.character(month), as
    .character(day), sep = "-")
  } else {
    formatted <- paste(as.character(day), as.character(month), as.
    character(year), sep = "-")
  }
  return(formatted)
}
format_date(4,10,2021)
```

```
[1] "2021-10-4"
```

```
# Technically correct, but rather unintuitive
format_date(y = 2021, m = 10, d = 4)
```

```
[1] "2021-10-4"
```

```
format_date(y = 2021, m = 10, d = 4, FALSE)
```

```
[1] "2021-10-4"
```

Nested functions

```
which_integer <- function(num){
  even_or_odd <- function(num){
    if (num %% 2 == 0){
      return("even")
    } else {
      return("odd")
    }
  }
  eo <- even_or_odd(num)
  if (num > 0){
    return(pasteo("positive ", eo))
  } else if (num < 0){
    return(pasteo("negative ", eo))
  } else {
    return("zero")
  }
}
which_integer(-43)
```

```
[1] "negative odd"
```

```
even_or_odd(-43)
```

```
Error in even_or_odd(-43) : could not find function "even_or_odd"
```

R environment basics

- Variables (aka names) exist in an *environment* (aka namespace/scope in Python)
- The same R object can have different names
- Binding of objects to names (assignment) happens within a specific environment
- Most environments get created by function calls
- Approximate hierarchy of environments:
  - *Execution* environment of a function
  - *Global* environment of a script
  - *Package* environment of any loaded packages
  - *Base* environment of base R objects

R environment example

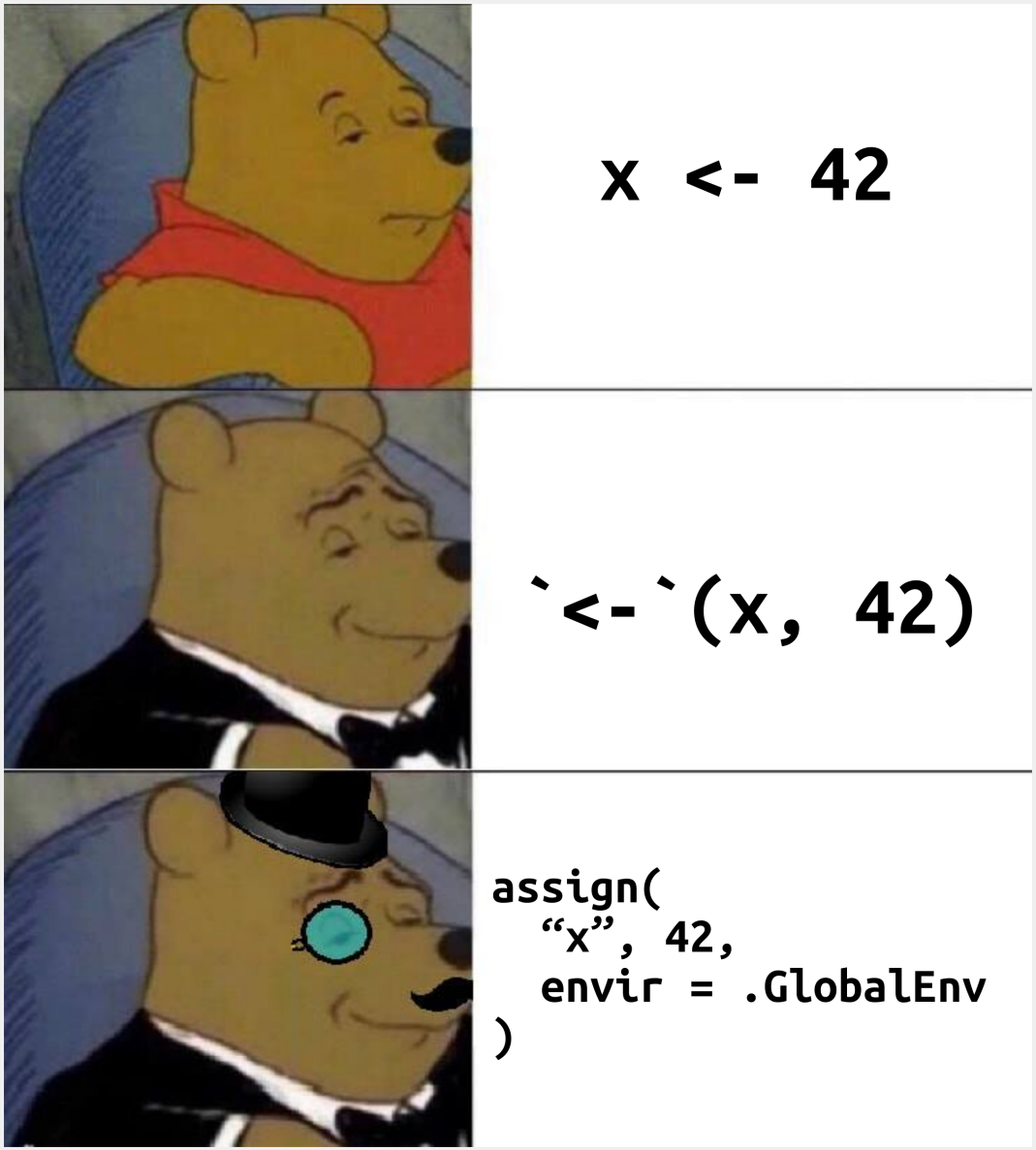
```r
x <- 42
# is equivalent to:
# Binding R object '42', double vector of length 1, to name 'x' in
    the global environment
assign("x", 42, envir = .GlobalEnv)
x
```

```
[1] 42
```

```r
x <- 5
foo <- function(){
  x <- 12
  return(x)
}
y <- foo()
print(y)
print(x)
```

```
[1] 12
[1] 5
```

R environments



Every operation is a function call

Matthew Kay @mjskay · Oct 31

remember kids everything in #rstats is a function

```
y = 9

for (x in 1:y) {
  if (x > 2) {
    xy = x * y
    cat(x, "*", y, "is", xy, "\n")
  }
}
#> 3 * 9 is 27
#> 4 * 9 is 36
#> 5 * 9 is 45
#> 6 * 9 is 54
#> 7 * 9 is 63
#> 8 * 9 is 72
#> 9 * 9 is 81
```

```
`=`(y, 9)

`for`(x, `:`(1, y), `{`(
  `if`(`>`(x, 2), `{`(
    `=`(xy, `*`(x, y)),
    cat(x, "*", y, "is", xy, "\n")
  ))
))
#> 3 * 9 is 27
#> 4 * 9 is 36
#> 5 * 9 is 45
#> 6 * 9 is 54
#> 7 * 9 is 63
#> 8 * 9 is 72
#> 9 * 9 is 81
```

11 38 377

Examples of operators as function calls

```
`+`(3, 2) # Equivalent to: 3 + 2
```

```
[1] 5
```

```
`<-`(x, c(10, 12, 14)) # x <- c(10, 12, 14)
```

```
[1]  10 12 14
```

```
`[`(x, 3) # x[3]
```

```
[1] 14
```

```
`>`(x, 10) # x > 10
```

```
[1] FALSE  TRUE  TRUE
```

Anonymous functions

- While R has no special syntax for creating anonymous (aka lambda in Python) function
- Note that the result of ‘function()‘ does not have to be assigned to a variable
- Thus function ‘function()‘ can be easily incorporate into other function calls

Anonymous functions

```
af <- add_five()
```

```
function(x) x + 5
<environment: 0x55d78232a7d8>
```

```
af # 'af' is just a function, which is yet to be invoked (called)
```

```
[1]  15
```

```
# Due to vectorized functions in R this example is an obvious
    overkill (seq(10) ^ 2 would do just fine)
# but it shows a general approach when we might need to apply a
    non-vectorized functions
sapply(seq(10), function(x) x ^ 2)
```

More on this next time!

Overview

This time:

- Decomposition and abstraction
- Arguments, environments
- Function definition and function call

Next time:

- Functionals
- Scoping in R