More on functions

Modified from materials by Mark Hansen, STAT 202a

Functional programming

Functional programming takes inspiration from the work of Princeton mathematician Alonzo Church on the lambda calculus

While the lambda calculus is a bit off topic for today, suffice it to say that in a functional model, we equate a program (a description of a specific computation) with a mathematical function; that is, a rule that associates to each set of inputs a unique output, or, in classical notation y = f(x)

With a programming language, we distinguish function definition and function application; the former tells us how a function is to be computed using formal parameters, while the latter is a call to a declared function that uses actual values

In mathematics this distinction is often a little clouded; when we write the definition

$f(x) = x * x$

we think of x as a formal parameter; but when we want to find x such that f(x) = 2, we think of x as a variable

In mathematics, variables always stand for actual values -- While in the programming languages variables can be both memory locations as well as values
For example, in mathematical notation x = x + 1 doesn't make much sense (whereas in Python or R we would mean that we increment x)

With functional programming, you don’t have the notion of a variable, except as a name for a value -- The idea that a variable is a piece of memory that can be updated is not allowed

With this restriction, we further eliminate (re)assignment as an available operation (by this we mean that we can give a name to a value but we cannot change that value later) -- Functions can be defined without repeatedly modifying some internal “state”

And without reassignment, we eliminate the possibility of a loop -- By its very definition, a loop has to have a control variable that is reassigned as the loop executes

This might sound crazy at first, but it turns out that any computation can be described using function calls alone

Below we define a simple R function that “reverses” a string, but does so without assignment and without an explicit loop -- Instead, the function uses recursion, in effect maintaining state through the arguments passed to successive calls to itself

```
flip <- function(x){
  if(x == ""){
    return(x)
    } else {
      return(paste(flip(substring(x,2)),substring(x,1,1),sep=""))
    }}

> flip("flip it and reverse it")
[1] "ti esrever dna ti pilf”
```

It’s worth mentioning a couple other key principles of this coding paradigm -- One direct implication of adopting a mathematical interpretation of functions is that the only result of applying a function is its return value (the output)

Put another way, functions cannot produce “side effects” (by contrast, recall that in Python we could change the value of mutable objects passed to functions)

In addition, because mathematical functions depend only on the values assigned to their formal parameters (the inputs), we don’t have to worry about “external influences” when applying a function (by contrast, recall that Python had a set of rules for searching outside a function definition for names outside the local scope)

Because of these last two properties, functional programs are easy to test -- It’s almost a problem in experimental design

It is also felt that functional programs are easier to debug because they don’t depend on a lot of external, possibly ephemeral, conditions that could be hard to reproduce

Finally, functional programs can be reasoned about mathematically, that is, you can prove things about your code, like whether or not two different programs represent the same function -- this makes it possible to create systems that analyze your code for correctness and even propose a range of input values for testing

One final note, to deliver on these promises, a functional programming language must be able to manipulate functions in arbitrary ways -- That is, functions must be general language objects

In particular, functions must be viewed as values themselves, which can be computed by other functions and which can also be parameters to functions

Functional programming and R

The view of functional programming on the previous slides is often referred to as “pure” -- Many languages that are functional in style break from this strict doctrine (we’ve seen many times in this class how some of the most powerful tools strike a balance between strict formalism and ease of use)

R, for example, allows for both local and non-local assignments, and some functions have wide-ranging side effects

Still, functions play an important and deep role in R -- The style of programming advocated by Chambers and others makes liberal use of functions for reasons like readability and testing

With that in mind, we’ll now shift gears and consider functions -- Now that you have written a few, we can talk a bit about what goes on “under the hood”...

Last time we saw that we create our own functions by evaluating an expression of the form

```
function(formal arguments) body
```

The resulting object has class `function` and it consists of three pieces:

- A function's **formal arguments** are contained in a comma-separated list of names followed optionally by the corresponding default expression (glued together with an '=' which is not to be confused with the assignment operator)

- The **body of a function** can be any complete R expression, but is typically a sequence of expressions (a block surrounded by {}'s}

- The **environment of a function** is important in determining what objects are visible in a call to this function; when a function is created by evaluating its expression (it's definition), the current environment is recorded as the function's environment

Your Turn

In the Console, print the code for `mad` (the mean absolute deviation). What are its formal arguments? Its body? Its environment?

```
> mad
function (x, center = median(x), constant = 1.4826, na.rm = FALSE,
    low = FALSE, high = FALSE)
{
    if (na.rm)
        x <- x[!is.na(x)]
    n <- length(x)
    constant * if ((low || high) && n%%2 == 0) {
        if (low && high)
            stop("'low' and 'high' cannot be both TRUE")
        n2 <- n%/%2 + as.integer(high)
        sort(abs(x - center), partial = n2)[n2]
    }
    else median(abs(x - center))
}
<bytecode: 0x10da27c40>
<environment: namespace:stats>
```

```
> body(mad)
{
    if (na.rm)
        x <- x[!is.na(x)]
    n <- length(x)
    constant * if ((low || high) && n%%2 == 0) {
        if (low && high)
            stop("'low' and 'high' cannot be both TRUE")
        n2 <- n%/%2 + as.integer(high)
        sort(abs(x - center), partial = n2)[n2]
    }
    else median(abs(x - center))
}
```

```
> formals(mad)
$x


$center
median(x)

$constant
[1] 1.4826

$na.rm
[1] FALSE

$low
[1] FALSE

$high
[1] FALSE
```

```
> environment(mad)
<environment: namespace:stats>
```

Notice that the environment for `mad` is the namespace for the package `stats` while `paste` and `substring` are associated with the package `base` -- Environments for function objects are important in that they determine **what other objects are visible from within a call to the function**

```
> environment(substring) <environment: namespace:base>
> environment(paste) <environment: namespace:base>
```

We will return to this idea in a moment -- Notice, however, if we create a function during an R session, **its environment is that of our workspace (`.GlobalEnv`)**, the top level global environment

```
# when we define our own functions on the command line, their environment
# is that of our workspace
> f <- function(x) x^2
> environment(f)
<environment: R_GlobalEnv>
> environment(mad)
<environment: namespace:stats>
> environment(paste)
<environment: namespace:base>
> search()
[1] ".GlobalEnv"        "package:reprex"    "package:Hmisc"     "package:Formula"
[5] "package:survival"  "package:lattice"   "package:skimr"     "package:bindrcpp"
[9] "package:forcats"   "package:ggplot2"   "package:tidyr"     "package:janitor"
[13] "package:readr"     "package:dplyr"     "tools:rstudio"     "package:stats"
[17] "package:graphics"  "package:grDevices" "package:utils"     "package:datasets"
[21] "package:methods"   "Autoloads"         "package:base"
```

Scoping rules

It is essential to be able to refer to objects that we create (or make use of objects and methods that others have created) -- The procedures languages follow for matching names to objects are known as scoping rules

We have seen R’s scoping rules in action already...

```
# the command search() presents our search path; when we type a name into
# R, it runs along the path, looking in each “environment” (here, the packages you
# have attached) until it finds an object associated with the name you requested
> search()
[1] ".GlobalEnv"        "package:reprex"    "package:Hmisc"     "package:Formula"
[5] "package:survival"  "package:lattice"   "package:skimr"     "package:bindrcpp"
[9] "package:forcats"   "package:ggplot2"   "package:tidyr"     "package:janitor"
[13] "package:readr"     "package:dplyr"     "tools:rstudio"     "package:stats"
[17] "package:graphics"  "package:grDevices" "package:utils"     "package:datasets"
[21] "package:methods"   "Autoloads"         "package:base"

# let’s look for pi
> pi
[1] 3.141593
> find("pi")
[1] “package:base"

# and let’s create our own
> pi <- 10
> find("pi")
[1] ".GlobalEnv"   "package:base"
> pi
[1] 10

# a way to get the value of pi we want
> base::pi
[1] 3.141593
> rm(pi)
```

Scoping rules

Essentially the same problem exists when you are executing the body of a function and R encounters a name that is not one of the formal arguments and has not been defined in the body of the function itself

Consider the following...

```
# first, a simple function that does nothing particularly interesting...
# x and y are the formal arguments, z is a local variable
> test <- function(x,y) {
   z <- 3
   return(x*y+z)
 }
> test(2,3)
[1] 9
# now, rewrite the function, but do not define z in the body
> test <- function(x,y) {
   return(x*y+z)
 }
> test(2,3)
Error in test(2, 3) : object 'z' not found

# there is no object z for it to find; for example, there's no z in our workspace
> ls(pattern="^z") character(0)
Error: unexpected symbol in "ls(pattern="^z") character"

# ... but if we make z... > z <- 5
> test(2,3)
Error in test(2, 3) : object 'z' not found

> ls(pattern="^z")
character(0)
```

Scoping rules

In R, the fundamental reference to an object consists of a pair -- The combination of **a name** (a character string) and **an environment** (a context) in which the name is evaluated

An environment is a class of object in R that consists of pairs of names and objects -- It also contains **a reference to another environment**, **its parent** and this simple structure creates a search hierarchy that R follows to names to objects

As you work through an R session, you will encounter a number of environments (well, they will be tapped into existence for you, whether you are directly aware of their presence or not)

Your **workspace or the so-called global environment** is one example; an environment is created when you **evaluate a function** (as we will see shortly); and **environments associated with packages contain objects exported to the session** or, in the package's namespace, the objects visible to functions in the package

Environments

Whether we are hunting for `pi` at the command line, or `z` within a function body, R uses environments to specify the search process

The chain of environments for an R session depends on what packages and other environments are attached; now we see that the function `search()` returns the names of these environments in a search list

```
> search()
 [1] ".GlobalEnv"        "package:MASS"      "package:reprex"    "package:Hmisc"
 [5] "package:Formula"   "package:survival"  "package:lattice"   "package:skimr"
 [9] "package:bindrcpp"  "package:forcats"   "package:ggplot2"   "package:tidyr"
[13] "package:janitor"   "package:readr"     "package:dplyr"     "tools:rstudio"
[17] "package:stats"     "package:graphics"  "package:grDevices" "package:utils"
[21] "package:datasets"  "package:methods"   "Autoloads"         "package:base"
> ev <- parent.env(.GlobalEnv)
> environmentName(ev)
[1] "package:MASS"
> class(ev)
[1] "environment"
> ev2 <- parent.env(ev)
> environmentName(ev2)
[1] "package:reprex"
> ev2
<environment: package:reprex>
attr(,"name")
[1] "package:reprex"
attr(,"path")
[1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/reprex"
```

```
> library(MASS)
> search()
 [1] ".GlobalEnv"        "package:MASS"      "package:reprex"    "package:Hmisc"
 [5] "package:Formula"   "package:survival"  "package:lattice"   "package:skimr"
 [9] "package:bindrcpp"  "package:forcats"   "package:ggplot2"   "package:tidyr"
[13] "package:janitor"   "package:readr"     "package:dplyr"     "tools:rstudio"
[17] "package:stats"     "package:graphics"  "package:grDevices" "package:utils"
[21] "package:datasets"  "package:methods"   "Autoloads"         "package:base"
> ev <- parent.env(.GlobalEnv)
> environmentName(ev)
[1] "package:MASS"
```

Environments

Notice that loading a package changes the order of your search path and there might be conflicts with names

Both mgcv and gam both have a function gam(), for example, and the version R will find depends on which of library(mgcv) and library(gam) was called last -- It is possible to refer uniquely to the function you want with the operator `::`, or mgcv::gam() and gam::gam()

(We will see how to rectify all this when we learn how to create our own packages)

Environments and functions

Recall that a function definition consists of three pieces: A function's formal arguments, the expressions that comprise its body, and an environment

When a function is created, it gets a reference to the environment in which its defining expression was evaluated -- So, if we define a function at the R prompt, it's environment is our workspace or the global environment

Calls to R functions

So we're now in a position to talk about how functions are evaluated in R; there are basically three steps

1. The argument expressions in the call (the actual arguments) are matched to the formal arguments in the function definition

2. A new environment is created and the data you provided in the call are copied to this new workspace (with defaults for the remaining arguments) -- the parent environment is the environment of the function object

3. The body of the function is evaluated in the new environment and the result is returned as the value of the function call

Calls to R functions

Remember, that assignments (like the one mentioned in step 2) **create copies of objects** -- In this way, we can be sure that whatever we pass to a function is not changed by that function

Also, recall that the expressions you pass as actual arguments are evaluated in this special environment; they are **evaluated the first time they are needed** (this is so-called lazy evaluation)

This is why we can define `center=median(x)` meaningfully as a default value in the function definition of `mad`

Calls to R functions: Scoping

R uses what is commonly referred to as static scoping or **“lexical scoping”**; the term “lexical” coming from the fact that if you were to sit down with the code, you could see (read) where each variable gets its value (This is in contrast to so-called dynamic scoping, which we won’t really get into this quarter)

When R evaluates the expressions in the function’s body, **it uses values associated with names in its evaluation environment** -- If the function body references a name not contained in this environment, it starts looking...

The hunt begins with the **parent environment of the function call, or rather the environment associated with the function object** -- In the simplest case, you create a function in your workspace, and this becomes its parent environment and R will search there for names

If it can’t be found in the function object’s environment, R starts searching, moving up the chain of parent environments we explored earlier