Functions

Functions in R

We have seen several examples of existing R functions, and done some tiny examples of defining our own functions, e.g. in `apply` or `integrate`.

```
apply(A, 2, function(x) {sum(x)})
```

```
integrate(function(x){x*dnorm(x)},
          lower = -Inf, upper = Inf)
```

Functions are key to writing **generalizable** code. If there's code we want to repeatedly apply to very or even just slightly different data, we want to avoid copying and pasting that code and instead store the code in a function that can be called again.

Components of A Function

All functions have some "body" or content comprised of lines of code they call.

```
apply(A, 2, function(x) {sum(x)})
```

```
integrate(function(x){x*dnorm(x)},
          lower = -Inf, upper = Inf)
```

Most functions have inputs.

Some functions have outputs.

Defining a Function

Most functions have names, which come first when defining a function.

Let's define a function called `emptyfunction`.

```
emptyfunction <- function() {
}
```

We call this function by typing `emptyfunction()`.

Defining a Function

Now let's actually define a function that does something.

```
ex <- function(x) {
  x*dnorm(x)
}
```

This function takes an **input** `x` and returns an **output** `x*dnorm(x)`.

By default, the function will return the output of the last line of code it contains.

Output of a Function

Not all functions return output.

What does the variation below return?

```
ex <- function(x) {
  v <- x*dnorm(x)
}
```

Output of Functions

All of these functions produce the same output.

```
ex <- function(x) {
  x*dnorm(x)
}
```

```
ex <- function(x) {
  return(x*dnorm(x))
}
```

```
ex <- function(x) {
  v <- x*dnorm(x)
  v
}
```

```
ex <- function(x) {
  v <- x*dnorm(x)
  return(v)
}
```

Scope of a Function

We just saw an example of a new object being defined within a function.

```
ex <- function(x) {
  v <- x*dnorm(x)
  return(v)
}
```

Objects defined within a function exist within the function but not outside of it unless you replace the assignment operators <- or = with <<-. **DON'T DO THIS!!!**

This is a very convenient aspect of functions, it can help us keep our workspace clean by temporarily creating objects as needed and then throwing them away for us once we're done with them.

Try it out!

Scope of a Function - Side Effects

A special feature of functions in R is that not only do objects defined within a function only exist within that function, but also objects defined within a function do not overwrite objects outside a function with the same name.

```
v <- 2
ex <- function(x) {
  v <- x*dnorm(x)
  return(v)
}
a <- ex(1)
```

What happens to `v` after we run `a <- ex(1)`?

Scope of a Function - Side Effects

To better drive this point home...,

```
toy <- function(x) {
  x <- x/2
  return(x)
}
x <- 2
toy(x)
```

```
[1] 1
```

```
x
```

```
[1] 2
```

Scope of a Function - Looking Out

Objects defined outside of a function exist inside of a function (providing they are defined before it). This can be dangerous and is somewhat unusual behavior for a programming language.

```
z <- 1
ex <- function(x) {
  v <- z*dnorm(z)
  return(v)
}
```

Try it out!

How can I tell what is in the scope?

Typing `ls()` returns a character vector of the names of all objects that are in scope.

```
ls()
```

```
[1] "a"   "ex"  "toy" "v"   "x"   "z"
```

Note: Functions are objects!

```
toy <- function(x) {
  x <- x/2
  print(ls())
  return(x)
}
```

```
toy(1)
```

```
[1] "x"
```

```
[1] 0.5
```

Order Matters

```
toy <- function(x) {
  x <- x/2
  return(x)
  print(ls())
}
```

```
toy(1)
```

```
[1] 0.5
```

Order Matters

```
toy <- function(x) {
  x <- x/2
  x
  print(ls())
}
```

```
toy(1)
```

```
[1] "x"
```

Do we need `return`?

Practically, not really. It might even slow things down a bit.

For readability, it's nice. And it does help to ensure that our functions return what we want them to return.

A Function Can Have Multiple Arguments

In practice, we will often want to allow a function to have multiple arguments.

For the `ex` function, we may want to change the mean and standard deviation of the normal density.

```
ex <- function(x, mean, sd) {
  x*dnorm(x, mean = mean, sd = sd)
}
```

Try it out!

What happens if we don't specify `mean` and `sd`?

We Can Specify Default Arguments

If some arguments have typical `default` values, we can specify those when defining the function.

```
ex <- function(x, mean = 0, sd = 1) {
  x*dnorm(x, mean = mean, sd = sd)
}
```

If values for the arguments `mean` or `sd` are not specified, the default values `0` or `1` will be used.

Try it out!

What can functions return?

Functions can return any type of object, including a function (although this is a rare way to use functions, and it is much more common for functions to return a vector, matrix, or nothing at all).

If we want our function to return **multiple** objects, even if they are of the same type, we need to return a **list**.

We won't talk about this just yet, but soon!

Making Functions Available

User defined functions are the primary, most useful contents of R packages.

Examining Others' Functions

Typing a function name into the console will return code used to define a function.

```
library(gnorm)
dgnorm
```

```
function (x, mu = 0, alpha = 1, beta = 1, log = FALSE)
{
    if (alpha <= 0 | beta <= 0) {
        cat("Not defined for negative values of alpha and/o
        return(rep(NaN, length(x)))
    }
    if (!log) {
        return(exp(-(abs(x - mu)/alpha)^beta) * beta/(2 * a
            gamma(1/beta)))
    }
    else {
        return(-(abs(x - mu)/alpha)^beta + log(beta) - (log
            log(alpha) + log(gamma(1/beta))))
```

Activities

Write a function that creates a square AR-1 covariance matrix of arbitrary size and correlation parameter ρ, with elements $c_{ij} = \rho^{|i-j|}$.

Write a function that smooths a vector by averaging sequences of k points, where k is provided by the user.