Functions are objects, and can be arguments to other functions

Example: `curve`

Example: `gradient` and `gradient.descent`

**Optional Recommended Reading:** Chapter 3 of Chambers

**Merely Useful Reading:** Chapter 3 of the textbook

**Code from this Lecture:** At class website, with comments
Functions as Objects

In R, functions are objects, just like everything else. This means that they can be passed to functions as arguments, and returned by functions as arguments as well. Today we’ll look at using functions as arguments. Next lecture will cover returning functions. Both ideas can be understood from your experience with calculus.
You already know these very well!
Maximum, and location of the maximum: takes $f$, gives number

$$\max f(x), \text{argmax} f(x)$$

Derivative of $f$ at $x_0$: takes a function and a point, gives a number

$$\frac{df}{dx}(x_0) \equiv \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

Definite integral of $f$ over $[a, b]$: takes a function and two points, gives a number

$$\int_a^b f(x)dx \equiv \lim_{n \to \infty} \sum_{i=0}^{n-1} \left( \frac{b-a}{n} \right) f \left( a + i \frac{b-a}{n} \right)$$
Functions of functions which return numbers sometimes are sometimes called \textbf{functionals}, e.g., expectation values:

$$\mathbb{E}[f(X)] \equiv \int_{\text{all } x} f(x)p(x)dx$$

$\nabla f(x_0)$ takes $f$ and $x_0$, gives vector: not strictly a functional

$\nabla f$ is another, vector-valued function

$\nabla$ takes a function and returns a function

$\nabla$ is an \textbf{operator}, not a functional

operators are for next time
We often want to do very similar things to many different functions. The procedure is the same, only the function we’re working with changes.

.: Write one function to do the job, and pass the function as an argument.

Because R treats functions as objects like any other, we can do this simply.

We have already seen an example: `apply` takes a function as one of its arguments.
Some R Syntax Facts About Functions

A call to `function` returns a function object
Typing a function’s name at the prompt gives the code
`formals(foo)` gives the list of arguments of `foo`: names are argument names, values are expressions for defaults (if any)
`body(foo)` gives the body of the definition
`environment(foo)` gives the environment in which it was defined

Functions can be put into lists or arrays
User-defined and built-in R functions are both of class `function`
User-defined functions are of class `closure`, built-ins are either `builtin` or `special` (don’t ask)
You learned to use `curve` in the first week (because you did all of the assigned reading, including section 2.3.3 of the textbook)
A call to `curve` looks like this:

```
curve(expr, from = a, to = b, ...)```

`expr` is some expression involving a variable called `x` which is swept from the value `a` to the value `b`
`...` are other plot-control arguments
`curve` presumes that the expression can take a vector of `x` values and return a vector of numerical values, e.g.,

```
curve(x^2 * sin(x))```

is fine
If we have defined a function already, we can use it in `curve`:

```r
psi <- function(x, c=1) {ifelse(abs(x)>c, c*abs(x), x^2)}
curve(psi(x, c=10), from=-20, to=20)

Try this! Also try

```r
curve(psi(x=10, c=x), from=-20, to=20)
```

and explain it to yourself
If our function doesn’t take vectors to vectors, `curve` becomes unhappy

```r
> mse <- function(y0,a,Y=gmp$pcgmp,N=gmp$pop) {
+     mean((Y - y0*(N^a))^2)
+ }
> curve(mse(a=x,y0=6611),from=0.10,to=0.15)
Error in curve(mse(a = x, y0 = 6611), from = 0.1, to = 0.15) : 
  ‘expr’ did not evaluate to an object of length ‘n’
In addition: Warning message:
In N^a : longer object length is not a multiple of shorter object length
```

How do we solve this?
Remember that `apply` applies the same function to every row or column of an array. `sapply` applies the same function to every element of an array or vector, and tries to simplify the result down to an array.

```r
c > sapply(seq(from=0.10, to=0.15, by=0.01),mse,y0=6611)
[1] 154701953 102322975 68755655 64529167 104079528 207057513

c > mse(6611,0.10)
[1] 154701953
```

Now (try it!):

```r
define mse.plottable <- function(a,...){sapply(a,mse,...)}
curve(mse.plottable(a=x), from=0.10, to=0.15)
curve(mse.plottable(a=x, y0=5100), from=0.10, to=0.20)
```

Next week, we will see many more related tricks for splitting up problems and applying the same function repeatedly.
Lots of statistical problems come down to optimization
Lots of optimization problems require finding the gradient of some objective function
We do the same thing to get the gradient of $f$ at $x$ no matter what $f$ is:

find the partial derivative of $f$ with respect to each component of $x$
return the vector of partial derivatives

It makes no sense to re-write this every time we change $f$!
Solution: write a function to calculate the gradient of an arbitrary function

```
gradient <- function(f, x, deriv.steps) {
  # not real code
  evaluate the function at x and at x+deriv.steps
  take slopes to get partial derivatives
  return the vector of partial derivatives
}
```
A naive implementation would use a for loop

\[
\text{gradient} \leftarrow \text{function}(f, x, \text{deriv.steps}, \ldots) \{
\begin{align*}
p & \leftarrow \text{length}(x) \\
\text{stopifnot}(\text{length}(\text{deriv.steps}) == p) \\
f.\text{old} & \leftarrow f(x, \ldots) \\
\text{gradient} & \leftarrow \text{vector(length}=p) \\
\text{for (coordinate in 1:p) \{} \\
& \quad x.\text{new} \leftarrow x \\
& \quad x.\text{new}[\text{coordinate}] \leftarrow x.\text{new}[\text{coordinate}] + \text{deriv.steps}[\text{coordinate}] \\
& \quad f.\text{new} \leftarrow f(x.\text{new}, \ldots) \\
& \quad \text{gradient}[\text{coordinate}] \leftarrow (f.\text{new} - f.\text{old})/\text{deriv.steps}[\text{coordinate}] \\
\} \\
\text{return}(\text{gradient}) \\
\}
\]

Works, but it’s so repetitive!
Better: use matrix manipulation and apply

```r
gradient <- function(f, x, deriv.steps, ...) {
  p <- length(x)
  stopifnot(length(deriv.steps) == p)
  f.old <- f(x, ...)
  x.new <- matrix(rep(x, times=p), nrow=p) + diag(deriv.steps, nrow=p)
  f.new <- apply(x.new, 2, f, ...)
  gradient <- (f.new - f.old) / deriv.steps
  return(gradient)
}
```
(clearer, and half as long)
Presumes that \( f \) takes a vector and returns a single number
Any extra arguments to `gradient` will get passed to `f`
Check: Does this work when \( f \) is a function of a single number?
How can gradient be improved?

- Acts badly if $f$ is only defined on a limited domain and we ask for the gradient somewhere near a boundary
- Forces the user to choose `deriv.steps`
- Uses the same `deriv.steps` everywhere, imagine $f(x) = x^2 \sin x$

...and so on through much of a first course in numerical analysis (or at least §5.7 of *Numerical Recipes*)

If it really matters, use the `grad` function in the `numDeriv` package
Now we can use this as a piece of a larger machine:

```r
gradient.descent <- function(f, initial.x, max.iterations, step.scale, stopping.deriv,...) {
  x <- initial.x
  for (iteration in 1:max.iterations) {
    grad <- gradient(f,x,...)
    x <- x - step.scale*grad
    if(all(abs(grad) < stopping.deriv)) { break() }
  }
  fit <- list(argmin=x, final.gradient=grad, iterations=iteration)
  return(fit)
}
```

(As written, we need to specify `deriv.steps` when calling this, but that’s not an argument. (How can you tell? Why make this choice?))

Works equally well whether $f$ is mean squared error of a regression, $\psi$ error of a regression, (negative log) likelihood, cost of a production plan, ...
gradient.descent presumes f takes a vector
mse takes two scalars

What to do?

1. Put a wrapper around mse:

```r
mse.for.optimization <- function(param,...) {
  return(mse(y0=param[1], a=param[2], ...))
}
gradient.descent(f=mse.for.optimization, blah blah blah)
```

2. Use an anonymous function:

```r
gradient.descent(f=function(param,...) {mse(y0=param[1],
  a=param[2],...)},blah blah blah)
```

(in fact the f= is optional here)

Anonymous functions work because the return value of function
is a function object
Anonymous functions don’t clutter your workspace, but they don’t
stick around for you to examine later
> mse.for.optimization <- function(param,...) {
+   mse(y0=param[1],a=param[2],...)
+ }
> gradient.descent(f=mse.for.optimization,initial.x=c(6611,0.15),
+   max.iterations=1e5,step.scale=c(1e-3,1e-12),stopping.deriv=1e-2,
+   deriv.step=c(1,1e-6))
$argmin
[1] 6441.0585327 0.1283084
$final.gradient
[1] -1.773238e-06 7.450581e-03
$iterations
[1] 41156
> gradient.descent(function(param,...) {mse(y0=param[1],a=param[2],...)}
+   initial.x=c(6611,0.15),max.iterations=1e5,step.scale=c(1e-3,1e-12),
+   stopping.deriv=1e-2,deriv.step=c(1,1e-6))
$argmin
[1] 6441.0585327 0.1283084
$final.gradient
[1] -1.773238e-06 7.450581e-03
$iterations
[1] 41156
Cautions

**Scoping**  \( f \) takes values for all names which aren’t its arguments from the environment where it was defined, not the one where it is called (e.g., not from inside `gradient` or `gradient.descent`)

**Debugging**  If \( f \) and \( g \) are both complicated, avoid debugging \( g(f) \) as a block; divide the work by writing *very simple* \( f . 0 \) to debug/test \( g \), and debug/test the real \( f \) separately
In R, functions are objects, and can be arguments to other functions.
Use this to automate doing the same thing to many different functions.
Separate writing the high-level operation from writing the first-order functions.
Use `sapply` (etc.), wrappers, anonymous functions as adaptors.

Next time: functions as outputs.
**EXERCISE:** Using the built-in function `contour`, write a function `surface`, which works like `curve` but for two variables, `x` and `y`.