Statistical Computing (36-350)

Lecture 10: Functions as Objects

Cosma Shalizi

1 October 2012

Agenda

- Functions are objects, and can be arguments to other functions
- Example: `curve`
- Example: `gradient` and `gradient.descent`
- Functions as return values
- Example: Linear predictor
- Example: the gradient operator
- Example: `surface`

READING: Sections 7.5, 7.11 and 7.13 of Matloff
OPTIONAL RECOMMENDED READING: Chapter 3 of Chambers
CODE FROM THIS LECTURE: At class website, with comments
NO LECTURE this Wednesday

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 outputs as well
Both ideas can be understood from your experience with calculus

Functions of Functions: Mathematically

You already know these very well!
Maximum, and location of the maximum: takes f, gives number

$$\max_x f(x)\,,\ \operatorname*{argmax}_x f(x)$$

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

$$\int_a^b f(x)dx \equiv \lim_{n\to\infty} \sum_{i=0}^{n-1} \left(\frac{b-a}{n}\right) f\left(a+i\frac{b-a}{n}\right)$$

Mathematical view cont'd.

Functions of functions which return numbers sometimes are sometimes called **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
∇f is another, vector-valued function
∇ takes a function and returns a function
∇ is an **operator**, not a functional

Mathematically

Something which takes a function in and gives a function back is an **operator**

Differentiation: the operator d/dx takes f and gives a new function

Gradient: the operator ∇ takes f and gives a new function

similarly $\nabla\cdot$, $\nabla\times$, ...

Indefinite integration: $\int_{-\infty}^{x} f(u)du$ takes f and gives a new function

Functions of Functions: Computationally

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 if its arguments

Some R Syntax Facts About Functions

A call to `function` returns a function object: body executed, arguments required, parent environment

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)

Example: curve

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

Using curve with your own functions

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

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

Try this! Also try

```
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

```
> 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?

sapply

`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

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

Now (try it!):

```
mse.plottable <- function(a,...){ return(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

Example: gradient

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!
$\therefore$ write code 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

```
gradient <- function(f,x,deriv.steps,...) {
  p <- length(x)
  stopifnot(length(deriv.steps)==p)
  f.old <- f(x,...)
  gradient <- vector(length=p)
  for (coordinate in 1:p) {
   x.new <- x
   x.new[coordinate] <- x.new[coordinate]+deriv.steps[coordinate]
   f.new <- f(x.new,...)
   gradient[coordinate] <- (f.new - f.old)/deriv.steps[coordinate]
  }
  return(gradient)
}
```

Works, but it's so repetitive!

Better: use matrix manipulation and `apply`

```
gradient <- function(f,x,deriv.steps,...) {
  p <- length(x)
  stopifnot(length(deriv.steps)==p)
  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(x,...))/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:

```
gradient.descent <- function(f,x,max.iterations,step.scale,
  stopping.deriv,...) {
  for (iteration in 1:max.iterations) {
    grad <- gradient(f,x,...)
    if(all(abs(grad) < stopping.deriv)) { break() }
    x <- x - step.scale*grad
  }
  fit <- list(argmin=x,final.gradient=grad,final.value=f(x,...),
    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, ψ error of a regression, (negative log) likelihood, cost of a production plan, ...

Wrappers and Anonymous Functions

`gradient.descent` presumes `f` takes a vector
`mse` takes two scalars
What to do?

1. Put a wrapper around `mse`:

```
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:

```
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

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

Returning Functions: A trivial example

Functions can be return values like anything else

```
make.noneuclidean <- function(ratio.to.diameter=pi) {
  circumference <- function(d) { return(ratio.to.diameter*d) }
  return(circumference)
}
```

Define `make.noneuclidean` but don't run it yet

```
> circumference(10)
Error: could not find function "circumference"
> kings.i <- make.noneuclidean(3)
> kings.i(10)
[1] 30
> formals(kings.i)
$d
> body(kings.i)
{
    return(ratio.to.diameter * d)
}
> environment(kings.i)
<environment: 0xe43d64>
> circumference(10)
Error: could not find function "circumference"
```

A Less Trivial Example

Create a linear predictor, based on sample values of two variables

```
make.linear.predictor <- function(x,y) {
  linear.fit <- lm(y~x)
  predictor <- function(x) {
   return(predict(object=linear.fit,newdata=data.frame(x=x)))
  }
  return(predictor)
}
```

The predictor function persists and works, even when the data we used to create it is gone

```
> library(MASS); data(cats)
> vet_predictor <- make.linear.predictor(x=cats$Bwt,y=cats$Hwt)
> rm(cats)              # Data set goes away
> vet_predictor(4.0)  # My cat's body mass in kilograms
       1
15.77959              # Predicted mass of my cat's heart in grams
```

A more mathematical example

Instead of finding $\nabla f(x)$, find the function ∇f:

```
nabla <- function(f,...) {
  g <- function(x,...) { gradient(f=f,x=x,...) }
  return(g)
}
```

```
> mse.gradient <- nabla(mse.for.optimization)
> mse.gradient(c(6611,0.15),deriv.steps=c(1,1e-6))
[1] 1.646082e+05 1.428795e+10
> gradient(mse.for.optimization,c(6611,0.15),c(1,1e-6))
[1] 1.646082e+05 1.428795e+10
> gradient(mse.for.optimization,c(6611,0.15),c(1,1e-6),Y=2*gmp$pcgmp)
[1] -2.908638e+05 -2.486987e+10
> mse.gradient(c(6611,0.15),deriv.steps=c(1,1e-6),Y=2*gmp$pcgmp)
[1] -2.908638e+05 -2.486987e+10
```

The simple first-differences method is not so hot, so use the `grad` function from `numDeriv`

```
del <- function(f,...) {
  require(numDeriv)
  g <- function(x,...) { grad(func=f,x=x, ...)}
  return(g)
}
```

How would you check this?

Example: surface

curve takes an expression and, as a side-effect, plots a 1-D curve by sweeping over x

Suppose we want something like that but sweeping over two variables

Built-in plotting function contour:

```
contour(x,y,z, [[other stuff]])
```

x and y are vectors of coordinates, z is a matrix of the corresponding shape

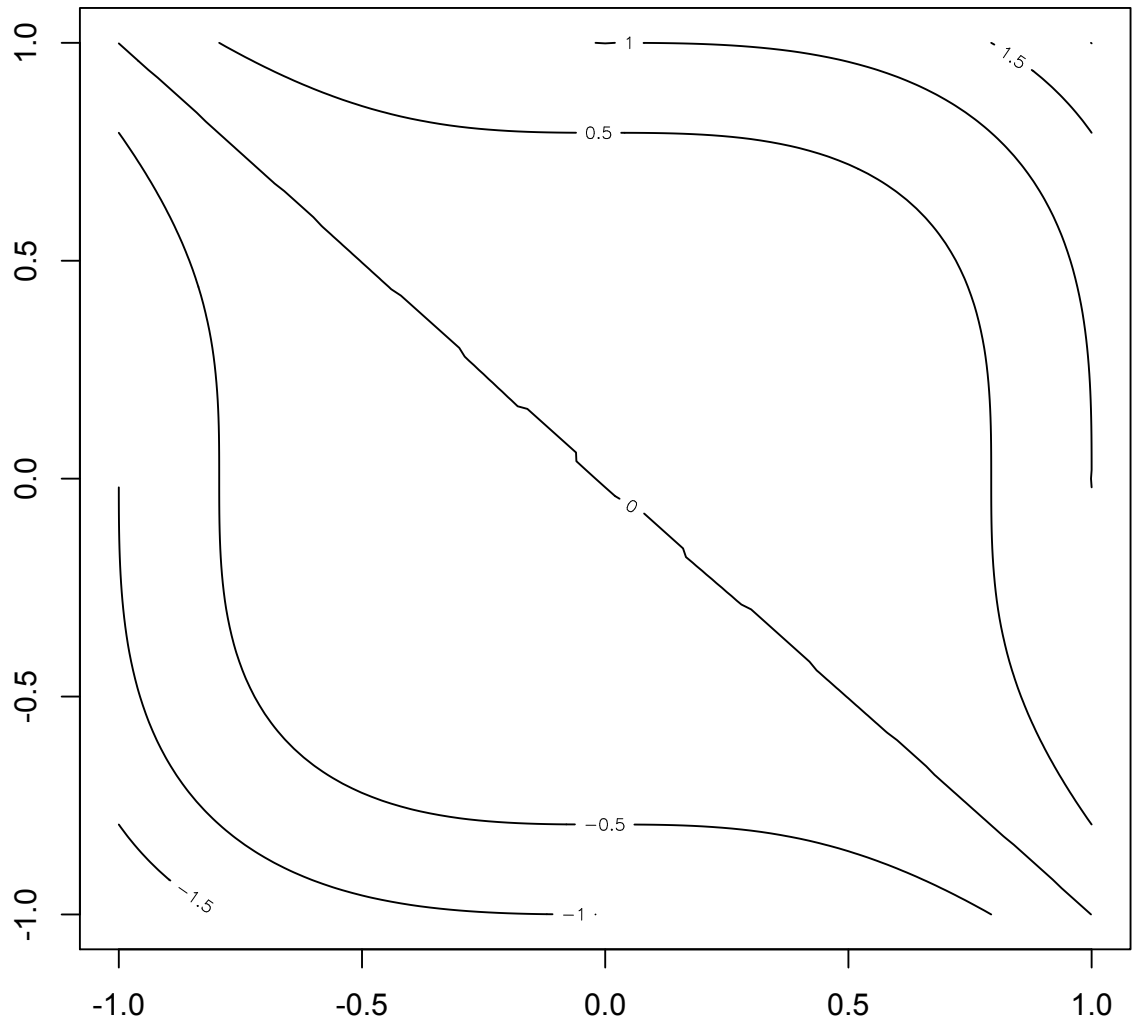
(see help(contour) for graphical options)

Strategy: surface should make x and y sequences, evaluate the expression at each combination to get z, and then call contour

First attempt at surface

Only works with vector-to-number functions:

```
surface.0 <- function(f,from.x=0,to.x=1,from.y=0,to.y=1,n.x=101,
  n.y=101,...) {
  x.seq <- seq(from=from.x,to=to.x,length.out=n.x)
  y.seq <- seq(from=from.y,to=to.y,length.out=n.y)
  plot.grid <- expand.grid(x=x.seq,y=y.seq)
  z.values <- apply(plot.grid,1,f)
  z.matrix <- matrix(z.values,nrow=n.x)
  contour(x=x.seq,y=y.seq,z=z.matrix,...)
  invisible(list(x=x.seq,y=y.seq,z=z.matrix))
}
```




```
surface.0(function(p){return(sum(p^3))},from.x=-1,from.y=-1)
```

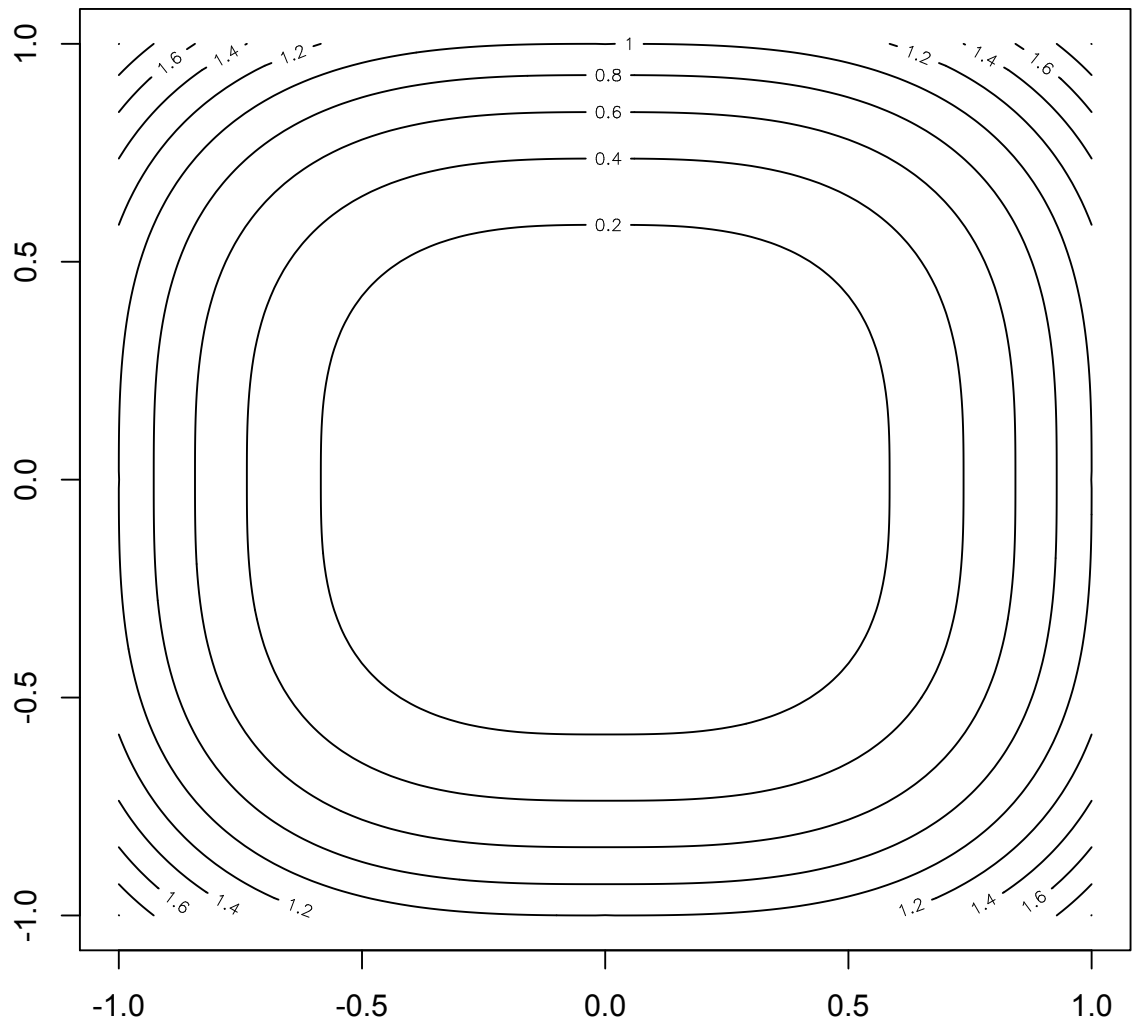
`curve` doesn't require us to write a function every time — what's it's trick?
Expressions are just another class of R object, so they can be created and manipulated
One manipulation is evaluation

```
eval(expr,envir)
```

evaluates the expression `expr` in the environment `envir`, which can be a data frame or even just a list
When we type something like `x^ 2+y^ 2` as an argument to `curve`, R tries to evaluate it prematurely
`substitute` returns the *unevaluted* expression
`curve` uses first `substitute(expr)` and then `eval(expr,envir)`, having made the right `envir`

Second attempt at surface

```
surface.1 <- function(expr,from.x=0,to.x=1,from.y=0,to.y=1,n.x=101,
  n.y=101,...) {
  x.seq <- seq(from=from.x,to=to.x,length.out=n.x)
  y.seq <- seq(from=from.y,to=to.y,length.out=n.y)
  plot.grid <- expand.grid(x=x.seq,y=y.seq)
  unevaluated.expression <- substitute(expr)
  z.values <- eval(unevaluated.expression,envir=plot.grid)
  z.matrix <- matrix(z.values,nrow=n.x)
  contour(x=x.seq,y=y.seq,z=z.matrix,...)
  invisible(list(x=x.seq,y=y.seq,z=z.matrix))
}
```



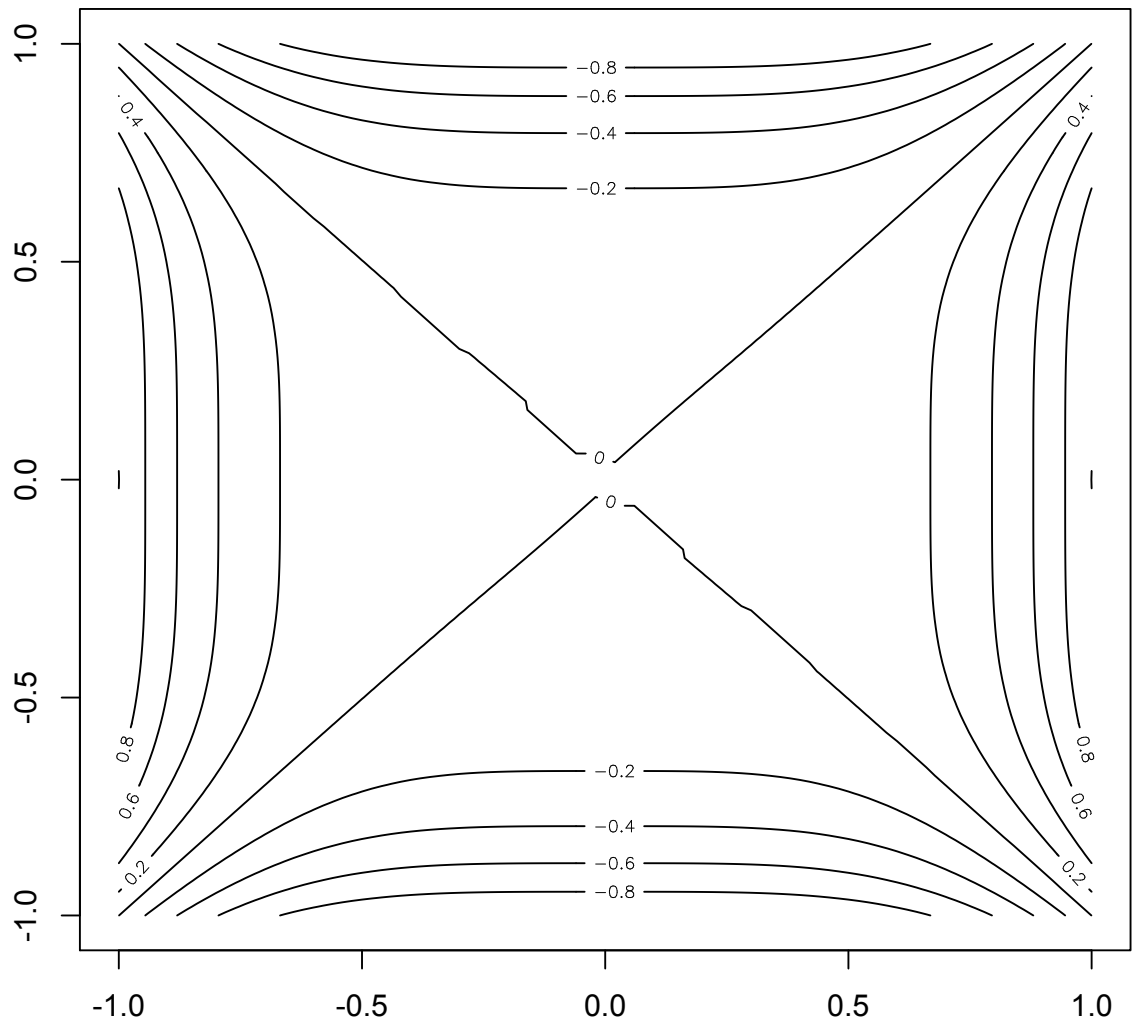

```
surface.1(abs(x^3)+abs(y^3),from.x=-1,from.y=-1)
```

Evaluating a function at every combination of two arguments is a really common task

There is a function to do it for us: `outer` (seen in lecture 3)

```
surface.2 <- function(expr,from.x=0,to.x=1,from.y=0,to.y=1,n.x=101,
  n.y=101,...) {
  x.seq <- seq(from=from.x,to=to.x,length.out=n.x)
  y.seq <- seq(from=from.y,to=to.y,length.out=n.y)
  unevaluated.expression <- substitute(expr)
  z <- function(x,y) {
    return(eval(unevaluated.expression,envir=list(x=x,y=y)))
  }
  z.values <- outer(X=x.seq,Y=y.seq,FUN=z)
  z.matrix <- matrix(z.values,nrow=n.x)
  contour(x=x.seq,y=y.seq,z=z.matrix,...)
  invisible(list(x=x.seq,y=y.seq,z=z.matrix))
}
```

could also include the function as part of the returned list




```
surface.2(x^4-y^4,from.x=-1,from.y=-1)
```

Summary

- In R, functions are objects, and can be arguments to other functions
  - Use this to do the same thing to many different functions
  - Separates writing the high-level operations and the first-order functions
  - Use `sapply` (etc.), wrappers, anonymous functions as adapters
- Functions can also be returned by other functions
  - Variables other than the arguments to the function are fixed by the environment of creation
  - Manipulating expressions lets us flexibly create functions

Next week: the split/apply/combine trick for doing big jobs in small pieces