Circle 5

Not Writing Functions

We came upon the River Styx, more a swamp really. It took some convincing, but Phlegyas eventually rowed us across in his boat. Here we found the treasoners.

5.1 Abstraction

A key reason that R is a good thing is because it is a language. The power of language is abstraction. The way to make abstractions in R is to write functions.

Suppose we want to repeat the integers 1 through 3 twice. That's a simple command:

```
c(1:3, 1:3)
```

Now suppose we want these numbers repeated six times, or maybe sixty times. Writing a function that abstracts this operation begins to make sense. In fact, that abstraction has already been done for us:

```
rep(1:3, 6)
```

The `rep` function performs our desired task and a number of similar tasks.

Let's do a new task. We have two vectors; we want to produce a single vector consisting of the first vector repeated to the length of the second and then the second vector repeated to the length of the first. A vector being repeated to a shorter length means to just use the first part of the vector. This is quite easily abstracted into a function that uses `rep`:

```
repeat.xy <- function(x, y)
{
    c(rep(x, length=length(y)), rep(y, length=length(x)))
}
```

The `repeat.xy` function can now be used in the same way as if it came with R.

```
repeat.xy(1:4, 6:16)
```

The ease of writing a function like this means that it is quite natural to move gradually from just using R to programming in R.

In addition to abstraction, functions crystallize knowledge. That π is approximately 3.14159265358979323846264338327950288419716939937510582097494459230 78 is knowledge.

The function:

```
circle.area <- function(r) pi * r ^ 2
```

is both knowledge and abstraction—it gives you the (approximate) area for whatever circles you like.

This is not the place for a full discussion on the structure of the R language, but a comment on a detail of the two functions that we've just created is in order. The statement in the body of `repeat.xy` is surrounded by curly braces while the statement in the body of `circle.area` is not. The body of a function needs to be a single expression. Curly braces turn a number of expressions into a single (combined) expression. When there is only a single command in the body of a function, then the curly braces are optional. Curly braces are also useful with loops, `switch` and `if`.

Ideally each function performs a clearly specified task with easily understood inputs and return value. Very common novice behavior is to write one function that does everything. Almost always a better approach is to write a number of smaller functions, and then a function that does everything by using the smaller functions. Breaking the task into steps often has the benefit of making it more clear what really should be done. It is also much easier to debug when things go wrong.[1] The small functions are much more likely to be of general use.

A nice piece of abstraction in R functions is default values for arguments. For example, the `na.rm` argument to `sd` has a default value of `FALSE`. If that is okay in a particular instance, then you don't have to specify `na.rm` in your call. If you want to remove missing values, then you should include `na.rm=TRUE` as an argument in your call. If you create your own copy of a function just to change the default value of an argument, then you're probably not appreciating the abstraction that the function gives you.

Functions return a value. The return value of a function is almost always the reason for the function's existence. The last item in a function definition is returned. Most functions merely rely on this mechanism, but the `return` function forces what to return.

The other thing that a function can do is to have one or more side effects. A side effect is some change to the system other than returning a value. The philosophy of R is to concentrate side effects into a few functions (such as `print`, `plot` and `rm`) where it is clear that a side effect is to be expected.

[1] Notice "when" not "if".

Table 5.1: Simple objects.

object	type	examples
logical	atomic	`TRUE FALSE NA`
numeric	atomic	`0 2.2 pi NA Inf -Inf NaN`
complex	atomic	`3.2+4.5i NA Inf NaN`
character	atomic	`'hello world' '' NA`
list	recursive	`list(1:3, b='hello', C=list(3, c(TRUE, NA)))`
NULL		`NULL`
function		`function(x, y) x + 2 * y`
formula		`y ~ x`

Table 5.2: Some not so simple objects.

object	primary	attributes	comment
data frame	list	`class row.names`	a generalized matrix
matrix	vector	`dim dimnames`	special case of array
array	vector	`dim dimnames`	usually atomic, not always
factor	integer	`levels class`	tricky little devils

The things that R functions talk about are objects. R is rich in objects. Table 5.1 shows some important types of objects.

You'll notice that each of the atomic types have a possible value `NA`, as in "Not Available" and called "missing value". When some people first get to R, they spend a lot of time trying to get rid of `NA`s. People probably did the same sort of thing when zero was first invented. `NA` is a wonderful thing to have available to you. It is seldom pleasant when your data have missing values, but life is much better with `NA` than without.

R was designed with the idea that nothing is important. Let's try that again: "nothing" is important. Vectors can have length zero. This is another stupid thing that turns out to be incredibly useful—that is, not so stupid after all. We're not so used to dealing with things that aren't there, so sometimes there are problems—we'll see examples in Circle 8, Circle 8.1.15 for instance.

A lot of the wealth of objects has to do with attributes. Many attributes change how the object is thought about (both by R and by the user). An attribute that is common to most objects is `names`. The attribute that drives object orientation is `class`. Table 5.2 lists a few of the most important types of objects that depend on attributes. Formulas, that were listed in the simple table, have class `"formula"` and so might more properly be in the not-so-simple list.

A common novice problem is to think that a data frame is a matrix. They look the same. They are not that same. See, for instance, Circle 8.2.37.

The word "vector" has a number of meanings in R:

1. an atomic object (as opposed to a list). This is perhaps the most common

usage.

2. an object with no attributes (except possibly `names`). This is the definition implied by `is.vector` and `as.vector`.

3. an object that can have an arbitrary length (includes lists).

Clearly definitions 1 and 3 are contradictory, but which meaning is implied should be clear from the context. When the discussion is of vectors as opposed to matrices, it is definition 2 that is implied.

The word "list" has a technical meaning in R—this is an object of arbitrary length that can have components of different types, including lists. Sometimes the word is used in a non-technical sense, as in "search list" or "argument list".

Not all functions are created equal. They can be conveniently put into three types.

There are anonymous functions as in:

```
apply(x, 2, function(z) mean(z[z > 0]))
```

The function given as the third argument to `apply` is so transient that we don't even give it a name.

There are functions that are useful only for one particular project. These are your one-off functions.

Finally there are functions that are persistently valuable. Some of these could well be one-off functions that you have rewritten to be more abstract. You will most likely want a file or package containing your persistently useful functions.

In the example of an anonymous function we saw that a function can be an argument to another function. In R, functions are objects just as vectors or matrices are objects. You are allowed to think of functions as data.

A whole new level of abstraction is a function that returns a function. The empirical cumulative distribution function is an example:

```
> mycumfun <- ecdf(rnorm(10))
> mycumfun(0)
[1] 0.4
```

Once you write a function that returns a function, you will be forever immune to this Circle.

In Circle 2 (page 12) we briefly met `do.call`. Some people are quite confused by `do.call`. That is both unnecessary and unfortunate—it is actually quite simple and is very powerful. Normally a function is called by following the name of the function with an argument list:

```
sample(x=10, size=5)
```

The `do.call` function allows you to provide the arguments as an actual list:

```
do.call("sample", list(x=10, size=5))
```

Simple.

At times it is useful to have an image of what happens when you call a function. An environment is created by the function call, and an environment is created for each function that is called by that function. So there is a stack of environments that grows and shrinks as the computation proceeds.

Let's define some functions:

```
ftop <- function(x)
{
   # time 1
   x1 <- f1(x)
   # time 5
   ans.top <- f2(x1)
   # time 9
   ans.top
}
f1 <- function(x)
{
   # time 2
   ans1 <- f1.1(x)
   # time 4
   ans1
}
f2 <- function(x)
{
   # time 6
   ans2 <- f2.1(x)
   # time 8
   ans2
}
```

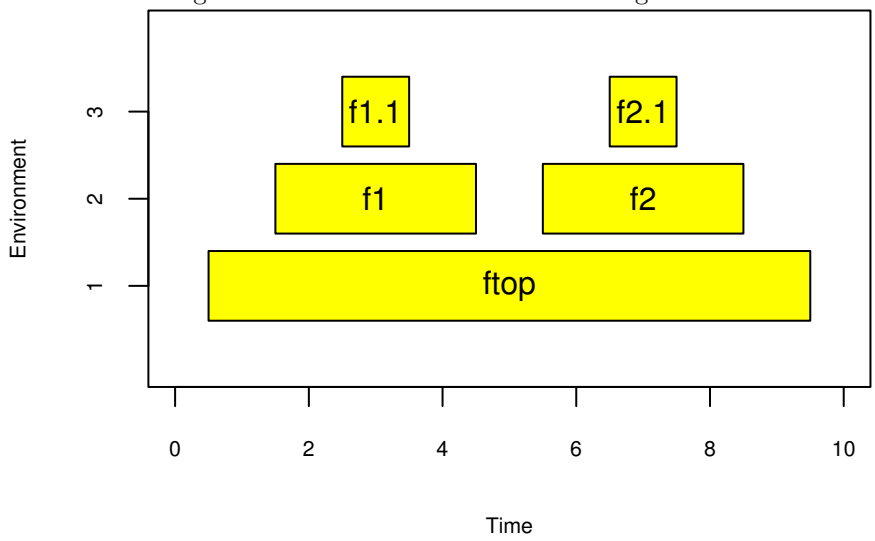
And now let's do a call:

```
# time 0
ftop(myx)
# time 10
```

Figure 5.1 shows how the stack of environments for this call changes through time. Note that there is an `x` in the environments for `ftop`, `f1` and `f2`. The `x` in `ftop` is what we call `myx` (or possibly a copy of it) as is the `x` in `f1`. But the `x` in `f2` is something different.

When we discuss debugging, we'll be looking at this stack at a specific point in time. For instance, if an error occurred in `f2.1`, then we would be looking at the state of the stack somewhere near time 7.

Figure 5.1: Stack of environments through time.



R is a language rich in objects. That is a part of its strength. Some of those objects are elements of the language itself—calls, expressions and so on. This allows a very powerful form of abstraction often called computing on the language. While messing with language elements seems extraordinarily esoteric to almost all new users, a lot of people moderate that view.

5.2 Simplicity

Make your functions as simple as possible. Simple has many advantages:

- Simple functions are likely to be human efficient: they will be easy to understand and to modify.
- Simple functions are likely to be computer efficient.
- Simple functions are less likely to be buggy, and bugs will be easier to fix.
- (Perhaps ironically) simple functions may be more general—thinking about the heart of the matter often broadens the application.

If your solution seems overly complex for the task, it probably is. There may be simple problems for which R does not have a simple solution, but they are rare.

Here are a few possibilities for simplifying:

- Don't use a list when an atomic vector will do.

- Don't use a data frame when a matrix will do.
- Don't try to use an atomic vector when a list is needed.
- Don't try to use a matrix when a data frame is needed.

Properly formatting your functions when you write them should be standard practice. Here "proper" includes indenting based on the logical structure, and putting spaces between operators. Circle 8.1.30 shows that there is a particularly good reason to put spaces around logical operators.

A semicolon can be used to mark the separation of two R commands that are placed on the same line. Some people like to put semicolons at the end of all lines. This highly annoys many seasoned R users. Such a reaction seems to be more visceral than logical, but there is some logic to it:

- The superfluous semicolons create some (imperceptible) inefficiency.
- The superfluous semicolons give the false impression that they are doing something.

One reason to seek simplicity is speed. The `Rprof` function is a very convenient means of exploring which functions are using the most time in your function calls. (The name `Rprof` refers to time profiling.)

5.3 Consistency

Consistency is good. Consistency reduces the work that your users need to expend. Consistency reduces bugs.

One form of consistency is the order and names of function arguments. Surprising your users is not a good idea—even if the universe of your users is of size 1.

A rather nice piece of consistency is always giving the correct answer. In order for that to happen the inputs need to be suitable. To insure that, the function needs to check inputs, and possibly intermediate results. The tools for this job include `if`, `stop` and `stopifnot`.

Sometimes an occurrence is suspicious but not necessarily wrong. In this case a warning is appropriate. A warning produces a message but does not interrupt the computation.

There is a problem with warnings. No one reads them. People have to read error messages because no food pellet falls into the tray after they push the button. With a warning the machine merely beeps at them but they still get their food pellet. Never mind that it might be poison.

The appropriate reaction to a warning message is:

1. Figure out what the warning is saying.

2. Figure out why the warning is triggered.
3. Figure out the effect on the results of the computation (via deduction or experimentation).
4. Given the result of step 3, decide whether or not the results will be erroneous.

You want there to be a minimal amount of warning messages in order to increase the probability that the messages that are there will be read. If you have a complex function where a large number of suspicious situations is possible, you might consider providing the ability to turn off some warning messages. Without such a system the user may be expecting a number of warning messages and hence miss messages that are unexpected and important.

The `suppressWarnings` function allows you to suppress warnings from specific commands:

```
> log(c(3, -1))
[1] 1.098612 NaN
Warning message:
In log(c(3, -1)) : NaNs produced
> suppressWarnings(log(c(3, -1)))
[1] 1.098612 NaN
```

We want our functions to be correct. Not all functions *are* correct. The results from specific calls can be put into 4 categories:

1. Correct.
2. An error occurs that is clearly identified.
3. An obscure error occurs.
4. An incorrect value is returned.

We like category 1. Category 2 is the right behavior if the inputs do not make sense, but not if the inputs are sensible. Category 3 is an unpleasant place for your users, and possibly for you if the users have access to you. Category 4 is by far the worst place to be—the user has no reason to believe that anything is wrong. Steer clear of category 4.

You should consistently write a help file for each of your persistent functions. If you have a hard time explaining the inputs and/or outputs of the function, then you should change the function. Writing a good help file is an excellent way of debugging the function. The `prompt` function will produce a template for your help file.

An example is worth a thousand words, so include examples in your help files. Good examples are gold, but any example is much better than none. Using data from the `datasets` package allows your users to run the examples easily.