SIOB 296 Introduction to Programming with R

Eric Archer (eric.archer@noaa.gov)

Week 5: May 2, 2017

Reading

The Book of R:
Chapters 9 - 11

The Art of R:
Chapter 7

Function Definition

The basic format for declaring a function actually uses a function called (wait for it…) `function`. I like to think of every function as having four components:

- **Name** - The name a user will use to call the function.
- **Arguments** - The input values a function needs to operate on.
- **Body** - The code that processes the arguments.
- **Return Value** - The output from the result of the processing in **Body**.

However, in any given function, depending on its purpose, one or more of the above items may be missing. Here is a simple function that has all four components designed to determine if `x` is between `a` and `b`:

```
isBetween <- function(x, a, b) {
  gt.a <- x > a
  lt.b <- x < b
  btwn <- gt.a & lt.b
  return(btwn)
}
```

In this function, the **name** is `isBetween`, and if we call it with the **arguments** `x = 6`, `a = 2`, and `b = 10`, it will **return** the value `TRUE`:

```
isBetween(x = 6, a = 2, b = 10)
```

```
[1] TRUE
```

Name

A function's name should be short(ish), but also meaningful and easy to understand. This is an art and you should take the time to play with names until they fit. Pretend like you are a naive user who has no idea what the function does. You should be able to get most of that information from the name. Also pay attention to how other functions that your function will be working with are named. For example, if you have a function that reads a particular data file and formats it, but is only one of several data files that will be read, it would be bad form to call that function `readData`. It might be better to call it something like, `readSalinityData`. Even better might be, `readAndFormatSalinityData`. However, it would be unnecessarily long and mean to users to call it something like, `readSalinityDataFromCSVFileAndRemoveMissingDataPoints`.

Arguments

These are the input values that the function needs to operate on. It is good programming practice to make them both as short and as long as necessary to be descriptive. In general, names for arguments should also be short and descriptive. However, some argument names are frequently used, such as `x` for the first argument, and `y` for the second argument, especially in mathematically-based functions or for data for axes in plotting functions. **It is good practice to not refer to anything in the function body that is either not in the arguments, or is not created in the function body.**

Body

This is the code that is the heart of the function. It operates on the arguments to convert them to a value to be returned or perform an action. Curly braces (`{` and `}`) are used to denote the code that composes the body and belongs to the function. If the function only has one line for a body then the curly braces can be omitted. For example:

```
isBetween.2 <- function(x, a, b) return(x > a & x < b)
isBetween.2(x = 6, a = 2, b = 10)
```

```
[1] TRUE
```

Return Value

A function can only return one object. The function called `return` is often used to denote what this object is as in the above examples. However, if there is no call to `return`, then **the result of the last line in a function is its return value**. For example, our `addAndRaise` function could also be written as:

```
isBetween.3 <- function(x, a, b) {
  gt.a <- x > a
  lt.b <- x < b
  gt.a & lt.b
}
isBetween.3(x = 6, a = 2, b = 10)
```

```
[1] TRUE
```

Sometimes you want a function to do an action, but only return a value if it is assigned to something. In this case, use the `invisible` function. In this example, our `isBetween` function will not print the result when called by itself:

```
isBetween <- function(x, a, b) {
  gt.a <- x > a
  lt.b <- x < b
  invisible(gt.a & lt.b)
}
# nothing printed
isBetween(x = 6, a = 2, b = 10)

# assign to object
result <- isBetween(x = 6, a = 2, b = 10)
result
```

```
[1] TRUE
```

Arguments

To better understand how arguments are handled, let's first create a function that abbreviates vectors of scientific names to shorter versions. For instance, we want a function that takes "Homo" and "sapiens" and creates "H sap":

```
abbrev <- function(genus, species) {
  # get the first character from genus names
  g <- substr(genus, 1, 1)
  # get the first three characters from the species names
  spp <- substr(species, 1, 3)
  # paste the two together and return the result
  paste(g, spp)
}
```

Let's also load some data to use with it:

```
spp.codes <- read.csv("tblCodeSpecies.csv", stringsAsFactors = FALSE)
head(spp.codes)
```

```
  SPCODE   ORDER   SUBORDER      FAMILY  FAMILY.NAMES      GENUS
1    001 CETACEA ODONTOCETI   ZIPHIIDAE BEAKED WHALES Mesoplodon
2    002 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS   Stenella
3    003 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS   Stenella
4    004 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS   Stenella
5    005 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS  Delphinus
6    006 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS   Stenella
             SPECIES                  COMMON.NAME
1          peruvianus           Pygmy beaked whale
2           attenuata  Pantropical spotted dolphin
3 longirostris subsp. unidentified spinner dolphin
4             clymene              Clymene dolphin
5                 sp.  Unidentified common dolphin
6 attenuata graffmani      Coastal spotted dolphin
```

```
gns <- spp.codes$GENUS
spp <- spp.codes$SPECIES
```

...and test it out:

```
gspp <- abbrev(genus = gns, species = spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

Arguments - Matching

Argument are matched according to two rules. First, arguments that are specifically named, like `genus = gns` are matched. Then any remaining unnamed arguments are matched based on the order in which they are found. This is simple to understand in our two argument function, which we can call as we have before, or like this:

```
# 'species' is not named
gspp <- abbrev(genus = gns, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

```r
# 'genus' is not named
gspp <- abbrev(species = spp, gns)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

If you know the name and order of the arguments, none of them have to be named as long as they are always supplied in the correct order. In many commonly used functions, this is normal for the first few arguments. So we would normally call this function like this:

```r
gspp <- abbrev(gns, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

Let's add a third argument to specify the number of characters in the species name to demonstrate the name and order matching further:

```r
abbrev <- function(genus, species, num.spp) {
  # get the first character from genus names
  g <- substr(genus, 1, 1)
  # get the 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp)
}
gspp <- abbrev(gns, spp, 3)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

```r
# we can also call it like this:
gspp <- abbrev(num.spp = 3, gns, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

```r
# this is fine too:
gspp <- abbrev(gns, num.spp = 3, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

```r
# but this will not produce the desired output:
gspp <- abbrev(num.spp = 3, spp, gns)
head(gspp)
```

```
[1] "p Mes" "a Ste" "l Ste" "c Ste" "s Del" "a Ste"
```

Argument names can also be abbreviated as long as the abbreviations are unique. Let's add a fourth argument specifying the number of characters in the genus name to return:

```r
abbrev <- function(genus, species, num.g, num.spp) {
  # get the first 'num.g' characters from genus names
  g <- substr(genus, 1, num.g)
  # get the first 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp)
}
```

```
gspp <- abbrev(gns, spp, num.g = 1, num.spp = 3)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

We can abbreviate `gns` as just `g` because no other arguments start with "g":

```
gspp <- abbrev(s = spp, g = gns, num.g = 1, num.spp = 3)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

However if we want to abbreviate `num.spp`, the shortest we can make it is `num.s` because any shorter than that and you couldn't differentiate it from `num.g`:

```
gspp <- abbrev(s = spp, g = gns, num.g = 1, num.s = 3)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

This will produce an error:

```
gspp <- abbrev(s = spp, g = gns, num = 1, 3)
```

```
Error in abbrev(s = spp, g = gns, num = 1, 3): argument 3 matches multiple formal arguments
```

Arguments - Defaults

Sometimes it is useful to specify default values for arguments. This means that users do not have to enter the default values every time an argument is called, but they can be modified if need be. Default values are specified by setting them directly in the argument list:

```
abbrev <- function(genus, species, num.g = 1, num.spp = 3) {
  # get the first 'num.g' characters from genus names
  g <- substr(genus, 1, num.g)
  # get the first 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp)
}
gspp <- abbrev(gns, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

But they can be changed by specifying them by name or position in the function call:

```
# set num.g to 3
gspp <- abbrev(gns, spp, 3)
head(gspp)
```

```
[1] "Mes per" "Ste att" "Ste lon" "Ste cly" "Del sp." "Ste att"
```

```
# set num.spp to 1
gspp <- abbrev(gns, spp, num.spp = 1)
head(gspp)
```

```
[1] "M p" "S a" "S l" "S c" "D s" "S a"
```

Arguments - Ellipses

There are times when you want to be able to pass arguments on to functions within your function, but you don't want to have to specify all possible arguments for that function in your argument list. To solve this, you can use the ellipses or dot-dot-dot notation, `...`. Here we use them to pass on formatting arguments like `sep` and `collapse` to the `paste` function:

```
abbrev <- function(genus, species, num.g = 1, num.spp = 3, ...) {
  # get the first 'num.g' characters from genus names
  g <- substr(genus, 1, num.g)
  # get the first 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp, ...)
}
gspp <- abbrev(gns, spp, sep = ".")
head(gspp)
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

Disposable Functions

There are times when an argument to a function is another function. In these cases, the function being passed can be predefined or, if it is being defined only for this purpose, it can just be defined directly in the argument list. This latter method creates what is called an "anonymous" or "disposable" function. One place this is frequently done is with the 'apply family of functions where one function is applied to every element in an object and the results are collected and returned in a convenient form. For example, let's say that species code data.frame is actually a list of data.frames with one element per Family:

```
families <- split(spp.codes, spp.codes$FAMILY)
names(families)
```

```
 [1] ""                 "BALAENIDAE"       "BALAENOPTERIDAE"
 [4] "CHELONIDAE"       "DELPHINIDAE"      "DERMOCHELYIDAE"
 [7] "ESCHRICTIDAE"     "INIIDAE"          "KOGIIDAE"
[10] "MONODONTIDAE"     "MUSTELIDAE"       "NEOBALAENIDAE"
[13] "ODOBENIDAE"       "OTARIIDAE"        "PHOCIDAE"
[16] "PHOCOENIDAE"      "PHYSETERIDAE"     "PLATANISTIDAE"
[19] "PONTOPORIIDAE"    "RHINCODONTIDAE"   "SIRENIA"
[22] "UNIDENTIFIED"     "URSIDAE"          "ZIPHIIDAE"
```

Now we want to know how many genera are in each Family. Let's first make a function that takes a data.frame and returns the number of unique genera in that data.frame:

```
num.genera <- function(df) length(unique(df$GENUS))
# Number of genera in entire data.frame
num.genera(spp.codes)
```

```
[1] 82
```

```
# Number of genera in Ziphiidae
num.genera(families$ZIPHIIDAE)
```

```
[1] 6
```

To get the number of genera in each family, we want to use `sapply` on the `families` list, where the second argument is the function we want to apply to every element in `families`:

```r
sapply(families, num.genera)
```

```
                      BALAENIDAE BALAENOPTERIDAE      CHELONIDAE 
              2               2               2               6 
    DELPHINIDAE  DERMOCHELYIDAE    ESCHRICTIDAE         INIIDAE 
             20               2               1               1 
       KOGIIDAE    MONODONTIDAE      MUSTELIDAE   NEOBALAENIDAE 
              1               2               1               1 
     ODOBENIDAE       OTARIIDAE        PHOCIDAE     PHOCOENIDAE 
              1               8               8               5 
   PHYSETERIDAE   PLATANISTIDAE  PONTOPORIIDAE RHINCODONTIDAE 
              1               1               2               1 
        SIRENIA    UNIDENTIFIED         URSIDAE       ZIPHIIDAE 
              2               7               1               6 
```

However, if we only need the `num.genera` function for the `sapply` then we don't have to predefine it. We can create the disposable function directly in the sapply argument list for the `FUN` argument, which is second (see `?sapply`):

```r
sapply(families, function(df) length(unique(df$GENUS)))
```

```
                      BALAENIDAE BALAENOPTERIDAE      CHELONIDAE 
              2               2               2               6 
    DELPHINIDAE  DERMOCHELYIDAE    ESCHRICTIDAE         INIIDAE 
             20               2               1               1 
       KOGIIDAE    MONODONTIDAE      MUSTELIDAE   NEOBALAENIDAE 
              1               2               1               1 
     ODOBENIDAE       OTARIIDAE        PHOCIDAE     PHOCOENIDAE 
              1               8               8               5 
   PHYSETERIDAE   PLATANISTIDAE  PONTOPORIIDAE RHINCODONTIDAE 
              1               1               2               1 
        SIRENIA    UNIDENTIFIED         URSIDAE       ZIPHIIDAE 
              2               7               1               6 
```

The disposable function may be more than one line in which case we just need to use curly braces to collect the statements. In this example the function returns the number of genera and the number of species. Make sure to include the closing curly brace for the function and the closing parentheses for the `sapply`:

```r
sapply(families, function(df) {
  num.gen <- length(unique(df$GENUS))
  num.spp <- length(unique(df$SPECIES))
  c(num.gen = num.gen, num.spp = num.spp)
})
```

```
          BALAENIDAE BALAENOPTERIDAE CHELONIDAE DELPHINIDAE DERMOCHELYIDAE
num.gen 2          2               2          6          20              2
num.spp 1          3               9          7          45              2
        ESCHRICTIDAE INIIDAE KOGIIDAE MONODONTIDAE MUSTELIDAE
num.gen            1       1        1            2          1
num.spp            1       1        3            2          1
        NEOBALAENIDAE ODOBENIDAE OTARIIDAE PHOCIDAE PHOCOENIDAE
num.gen             1          1         8        8           5
num.spp             1          1        10       14           7
        PHYSETERIDAE PLATANISTIDAE PONTOPORIIDAE RHINCODONTIDAE SIRENIA
num.gen            1             1             2              1       2
num.spp            1             2             2              1       3
```

```
         UNIDENTIFIED URSIDAE ZIPHIIDAE
num.gen             7       1         6
num.spp             1       1        24
```

Flow Control

Normally when code is executed, the "flow" proceeds in a linear fashion. The first line is executed, then the second, and so forth until the last line of code is reached. There can be situations where you want to direct this flow either in branching form: some piece of code is executed based on one condition, while another piece is executed based on another condition, or in a looping manner: the same code is executed repeatedly until some stopping criterion is reached (either number of iterations, or a condition is met). There are several functions that allow you to manage this flow control, the help for which can be found with `?Control`.

Branching

There are three standard branching functions:

- `if(cond) cons.expr else alt.expr` : executes a set of code (`cons.expr`) if `cond` evaluates to `TRUE` or (optionally) alternative code (`alt.expr`) if it is `FALSE`.
- `ifelse(test, yes, no)` : returns elements from `yes` matching to elements in `test` that are `TRUE` and elements in `no` for elements in `test` that are `FALSE`.
- `switch(EXPR, ...)` : executes individual code for named or numbered values in `EXPR`.

The thing to remember is that `if` is used for a single branching event (when `else` is not used) or a bifurcating branch (when `else` is used) that is based on a single condition (one `T` or `F`). `ifelse` is used to return a vector that is the same length as the logical vector with one set of values of for `TRUE` elements and another set for `FALSE` elements. `switch` is used in places where you want different pieces of code run for different discrete values. This is usually preferred if there are more than two possible conditions.

if

As an example of `if`, lets construct some checks of argument ranges in our species abbreviation code:

```r
abbrev <- function(genus, species, num.g = 1, num.spp = 3, ...) {
  # 'num.g' must be 1 or greater
  if(num.g < 1) num.g <- 1
  # 'num.g' shouldn't be too big
  if(num.g > 3) num.g <- 3
  # get the first 'num.g' characters from genus names
  g <- substr(genus, 1, num.g)
  # get the first 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp, ...)
}
gspp <- abbrev(gns, spp, num.g = 0, sep = ".")
head(gspp)
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```r
gspp <- abbrev(gns, spp, num.g = 10, sep = ".")
head(gspp)
```

```
[1] "Mes.per" "Ste.att" "Ste.lon" "Ste.cly" "Del.sp." "Ste.att"
```

Let's say that there are just two abbreviation formats, a short one like: "H.sap" and just the abbreviated genus: "H. sapiens". We control this with a simple argument, called `short`:

```
abbrev <- function(genus, species, short = T) {
  if(short) {
    g <- substr(genus, 1, 1)
    spp <- substr(species, 1, 3)
    g.spp <- paste(g, spp, sep = ".")
    return(g.spp)
  } else {
    g <- substr(genus, 1, 1)
    g.spp <- paste(g, species, sep = ". ")
    return(g.spp)
  }
}
# The short form
head(abbrev(gns, spp))
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```
# The longer form
head(abbrev(gns, spp, F))
```

```
[1] "M. peruvianus"           "S. attenuata"
[3] "S. longirostris subsp." "S. clymene"
[5] "D. sp."                  "S. attenuata graffmani"
```

We used `return(g.spp)` in order to make sure the function returns the result from the execution branch for each condition. We can simplify this code in several convenient ways. The first is based on the fact that the last line in the expression for each condition is the return value of the `if` function. So we can assign `g.spp` based on each branch then return it once at the end:

```
abbrev <- function(genus, species, short = T) {
  g.spp <- if(short) {
    g <- substr(genus, 1, 1)
    spp <- substr(species, 1, 3)
    paste(g, spp, sep = ".")
  } else {
    g <- substr(genus, 1, 1)
    paste(g, species, sep = ". ")
  }
  return(g.spp)
}
# The short form
head(abbrev(gns, spp))
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```
# The longer form
head(abbrev(gns, spp, F))
```

```
[1] "M. peruvianus"           "S. attenuata"
[3] "S. longirostris subsp." "S. clymene"
[5] "D. sp."                  "S. attenuata graffmani"
```

Also notice that we create `g <- substr(genus, 1, 1)` in each expression, so we can move that to the outside. Also, because **the result of the last line in a function is its return value** we can remove the `return(g.spp)` line:

```r
abbrev <- function(genus, species, short = T) {
  g <- substr(genus, 1, 1)
  if(short) {
    spp <- substr(species, 1, 3)
    paste(g, spp, sep = ".")
  } else {
    paste(g, species, sep = ". ")
  }
}
# The short form
head(abbrev(gns, spp))
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```r
# The longer form
head(abbrev(gns, spp, F))
```

```
[1] "M. peruvianus"           "S. attenuata"
[3] "S. longirostris subsp." "S. clymene"
[5] "D. sp."                  "S. attenuata graffmani"
```

ifelse

The `ifelse` function returns a vector that is as long as its first argument and chooses from the corresponding `yes` and `no` vectors to fill the elements. As an example, here's a function that will create the abbreviation "H. sapiens", but if the species name is longer than 8 characters, it will abbreviate that too:

```r
abbrev <- function(genus, species) {
  g <- substr(genus, 1, 1)
  spp <- ifelse(nchar(species) > 8, substr(species, 1, 8), species)
  paste(g, spp, sep = ". ")
}
head(abbrev(gns, spp))
```

```
[1] "M. peruvian" "S. attenuat" "S. longiros" "S. clymene"  "D. sp."
[6] "S. attenuat"
```

The expressions in `ifelse` can be multiple lines too, but must be wrapped by curly braces. This modification adds a "." to the end of the abbreviated species name:

```r
abbrev <- function(genus, species) {
  g <- substr(genus, 1, 1)
  spp <- ifelse(
    nchar(species) > 8,
    {
      spp.sub <- substr(species, 1, 8)
      paste0(spp.sub, ".")
    },
    species
  )
  paste(g, spp, sep = ". ")
}
head(abbrev(gns, spp))
```

```
[1] "M. peruvian." "S. attenuat." "S. longiros." "S. clymene"
[5] "D. sp."       "S. attenuat."
```

switch

The final branching function is `switch` which allows us to choose one of a series of expressions to execute based on a numeric or character value. For example, our abbreviation code will have an argument, `type` that will allow for three formats: `short` = "H.sap", `medium` = "H. sapiens", and `long` = "Homo sapiens":

```r
abbrev <- function(genus, species, type) {
  g <- substr(genus, 1, 1)
  # we only need an `if` statement to format the species
  spp <- if(type == "short") substr(species, 1, 3) else species
  # choose the pasting format based on `type`
  switch(type,
    short = paste0(g, ".", spp),
    medium = paste0(g, ". ", spp),
    long = paste(genus, species)
  )
}
# The short form
head(abbrev(gns, spp, "short"))
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```r
# The medium form
head(abbrev(gns, spp, "medium"))
```

```
[1] "M. peruvianus"          "S. attenuata"
[3] "S. longirostris subsp." "S. clymene"
[5] "D. sp."                 "S. attenuata graffmani"
```

```r
# The long form
head(abbrev(gns, spp, "long"))
```

```
[1] "Mesoplodon peruvianus"        "Stenella attenuata"
[3] "Stenella longirostris subsp." "Stenella clymene"
[5] "Delphinus sp."                "Stenella attenuata graffmani"
```

Looping

There are three functions to control looping:

- `for(var in seq)` : Executes a set of code for a number of iterations equal to the length of `seq`. In each iteration `var` gets sequential values of `seq`.
- `while(cond) expr` : Executes a set of code as long as `cond` is `TRUE`.
- `repeat expr` : Repeats code. To stop looping, execute `break`.

for

With `for` we execute a set of code for each element in a vector and in each execution, a variable takes sequential values of that vector. In the below example, we calculate the first `n` values of the fibonacci series.

```r
fib <- function(n) {
  x <- 0
  for(i in 2:n) {
    if(i == 2) {
      x[i] <- 1
    } else {
      x[i] <- x[i - 1] + x[i - 2]
    }
  }
  x
}
fib(10)
```

```
 [1]  0  1  1  2  3  5  8 13 21 34
```

A `for` loop can be stopped with the `break` function. For example, we will put in a function that stops the loop the first time a number greater than 50 is reached:

```
fib <- function(n) {
  x <- 0
  for(i in 2:n) {
    if(i == 2) {
      x[i] <- 1
    } else {
      x[i] <- x[i - 1] + x[i - 2]
    }
    if(x[i] > 50) break
  }
  x
}
fib(13)
```

```
 [1]  0  1  1  2  3  5  8 13 21 34 55
```

We can also force the `for` loop to iterate again before reaching the natural end of code in an iteration with the `next` command. In this example, we only print values greater than the number specified in `print.num`:

```
fib <- function(n, print.num = 10) {
  x <- 0
  for(i in 2:n) {
    if(i == 2) {
      x[i] <- 1
    } else {
      x[i] <- x[i - 1] + x[i - 2]
    }
    if(x[i] < print.num) next
    cat(i, " : ", x[i], "\n")
  }
  x
}
fib(15)
```

```
8  :  13
9  :  21
10  :  34
11  :  55
12  :  89
13  :  144
14  :  233
15  :  377
```

```
 [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

while

The `while` function is designed to repeat some code until a condition is met. If the condition is never met, the loop will continue indefinitely. The `break` and `next` commands will perform the same with function and `repeat`. In this example, we print the fibonacci series until the specified number is exceeded:

```
fib <- function(n) {
  first <- 0
  second <- 1
  cat(first, " ")
  while(second <= n) {
    cat(second, " ")
    new.val <- first + second
    first <- second
    second <- new.val
  }
}
fib(20)
```

```
0  1  1  2  3  5  8  13
```

repeat

The `repeat` function will continuously execute code until it is stopped by a `break`. Here, we do the same loop as above, but replace `while` with a `repeat` and `break`:

```
fib <- function(n) {
  first <- 0
  second <- 1
  cat(first, " ")
  repeat {
    cat(second, " ")
    new.val <- first + second
    if(new.val > n) break
    first <- second
    second <- new.val
  }
}
fib(20)
```

```
0  1  1  2  3  5  8  13
```

Argument error checking

Despite our best efforts, functions are susceptible to users entering improper arguments or errors showing up that keeps a function from completing. We have some tools at our disposal to mitigate this. First, we can check that arguments what we expect and require using `if` statements. If they're aren't we can do something like return `NA` or `NULL`:

```
addTwo <- function(a, b) {
  # confirm that a and b are numbers
  if(!(is.numeric(a) | is.numeric(b))) return(NULL)
  a + b
}
addTwo(1, "x")
```

```
Error in a + b: non-numeric argument to binary operator
```

```
addTwo(1, 2)
```

```
[1] 3
```

We can also issue warnings when something unexpected happens:

```
addTwo <- function(a, b) {
  # confirm that a and b are numbers
  if(!(is.numeric(a) | is.numeric(b))) {
    warning("'a' and 'b' must be numbers. NULL returned.")
    return(NULL)
  }
  a + b
}
addTwo(1, "x")
```

```
Error in a + b: non-numeric argument to binary operator
```

If execution cannot or should not continue, then an error can be thrown with the `stop` function:

```
divideTwo <- function(a, b) {
  # confirm that a and b are numbers
  if(!(is.numeric(a) | is.numeric(b))) {
    stop("'a' and 'b' must be numbers. NULL returned.")
  }
  if(b == 0) {
    stop("'b' cannot be 0")
  }
  a / b
}
divideTwo(1, "x")
```

```
Error in a/b: non-numeric argument to binary operator
```

```
divideTwo(5, 0)
```

```
Error in divideTwo(5, 0): 'b' cannot be 0
```

Scope

It is important to note that objects declared in a function only exist within that function. On the flipside, objects declared outside of a function are accessible by that function. However, it is very bad form to refer to an object in a function that has not been passed as an argument or declared in the function itself. For example:

```
a <- 2

my.func <- function(x, y) (x + y) * a

# this works
my.func(2, 3)
```

```
[1] 10
```

```r
# remove a from the workspace
rm(a)

# a can't be found, so this produces an error
my.func(2, 3)
```

```
Error in my.func(2, 3): object 'a' not found
```