MTH 2520 R Notes 12

11 Object-Oriented Programming

11.1 Objects and Classes

- In R:
  - "Everything that *exists* is an object.
  - Everything that *happens* is a function call."

  (The quote is from John Chambers.)

- Each object belongs to a specific ***class*** of objects. The most important classes are:

`"numeric"` vectors	`"logical"` vectors
`"character"` vectors	`"matrix"`
`"array"`	`"list"`
`"data.frame"`	`"factor"`
`"function"`	

  (We'll use quotes, as above, when referring to a *class* of objects.)

- We can get (or set) the class of an object using:

```
class()         # Returns the class of an object.  Can also be used
                # to assign a class attribute to an object.
is.cname()      # Returns TRUE if an object belongs to the class "cname"
                # (e.g. is.numeric(), is.data.frame(), etc.) and FALSE
                # otherwise.
```

- For example:

```
> x <- c(2, 4, 7, 9)
> class(x)

[1] "numeric"

> is.numeric(x)

[1] TRUE
```

- Another example:

```
> a <- matrix(1:6, nrow = 2, ncol = 3)
> class(a)

[1] "matrix"

> is.matrix(a)

[1] TRUE
```

- One more example:

```
> y <- data.frame(y1 <- c(1, 4), y2 <- c("u", "v"))
> class(y)

[1] "data.frame"

> is.data.frame(y)

[1] TRUE
```

- It's important to distinguish between the *class* of an object (returned by `class()`) and the *type* of the object (returned by `typeof()`).
  - The *type* refers to how R *stores* the object internally.
  - The *class* is a property (or *attribute*) of the object that affects what functions do with the object when it's passed to them as an argument.

  Sometimes the type and class are the same, but it's not always the case.

Section 11.1 Exercises

Exercise 1 In R, everything that "exists" is an *object*, and each object belongs to a *class* of objects. Guess the class of each of the objects below, then check your answers.

a)
```
> u <- c("a", "b", "c")
> class(u)
```

b)
```
> x <- c(3, 6, 1)
> class(x)
```

c)
```
> y <- list(3, 5, 2)
> class(y)
```

Exercise 2 I In R, everything that "exists" is an *object*, and each object belongs to a *class* of objects. Even *functions* are objects.

a) Guess the class of `sqrt()`, then check your answer.

```
> class(sqrt)
```

b) Guess what will be returned by the following command, then check your answer.

```
> is.function(sqrt)
```

c) Because functions are objects, they can be passed as arguments to other functions. The function `apply.fun()` below takes a function `FUN` and a value `x` as arguments and applies `FUN` to `x`:

```
> apply.fun <- function(x, FUN) {
    do.call(FUN, args = list(x))
  }
```

(Above, `do.call()` executes a function call by passing the value `x` to the function `FUN`. It wouldn't work to just type `FUN(x)`.)

Guess what the result of each of the following commands will be, then check your answers:

```
> apply.fun(x = 4, FUN = sqrt)
```

```
> apply.fun(x = -3, FUN = abs)
```

Exercise 3 The *type* of an object refers to how R *stores* the object internally. The *class* is a property (or *attribute*) of the object that affects what functions do with the object when it's passed to them as an argument.

Sometimes the type and class of an object are the same, but not always. Here are two objects:

```
> x <- c("a", "b", "c")
> y <- factor(c("a", "b", "c"))
```

a) Guess what the result of each of the following commands will be, then check your answers:

```
> class(x)
```

```
> typeof(x)
```

b) Guess what the result of each of the following commands will be, then check your answers:

```
> class(y)
```

```
> typeof(y)
```

11.2 Generic Functions and Methods

11.2.1 Generic Functions

- For some R functions, the object passed to it as an argument has to belong to a *single*, *specific* class.

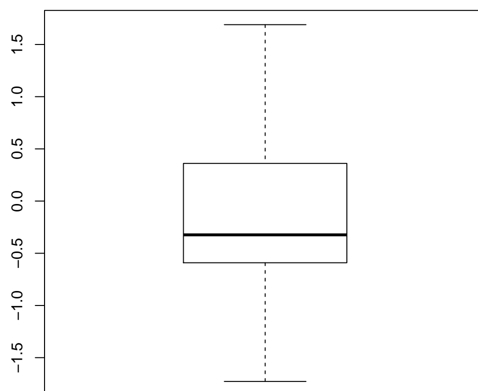
  For other functions, called ***generic*** functions, the object passed to it can belong to any of *several* classes, and the function does a different thing depending on the class of that object.

- For example, `boxplot()` is *generic* because we can pass it a `"numeric"` vector *or* a `"matrix"` (among other classes of objects), and it does something different depending on which of these we pass to it.

  Below, `boxplot()` is first passed a `"numeric"` vector and then a `"matrix"`:

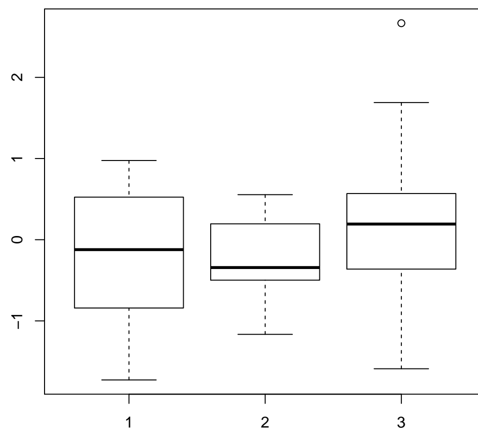
```
> x <- rnorm(n = 25)

> boxplot(x)                    # x is a numeric vector.
```




```
> a <- matrix(rnorm(n = 30), nrow = 10, ncol = 3)

> boxplot(a)                    # a is a matrix with 3 columns.
```



When passed a `"numeric"` vector, `boxplot()` produces a single boxplot, but when passed a `"matrix"`, it produces side-by-side boxplots of the matrix's columns.

11.2.2 Methods

- Actually, a generic function such as `boxplot()` is a whole *family* of functions, each of which is referred to as a ***method*** and is designed to handle a *specific* class of objects.
- When we pass an object to a generic function like `boxplot()`, the function determines the class of the object and then ***dispatches*** it to the appropriate *method*.
- If we look at a generic function's definition, we see that it calls another function, `UseMethod()`:

```
> boxplot

function (x, ...)
UseMethod("boxplot")
<bytecode: 0x0ba1eeac>
<environment: namespace:graphics>
```

  `UseMethod()` determines the class of the object that was passed to the generic function (`boxplot()` above), and then *dispatches* that object to the appropriate method.

- We can view the methods associated with a generic function using:

```
methods()       # Determine the S3 methods that are associated with a
                # given generic function
showMethods()   # Determine the S4 methods that are associated with a
                # given generic function
```

- For example, to see the methods for `boxplot()` type:

```
> methods(boxplot)

[1] boxplot.default  boxplot.formula* boxplot.matrix
see '?methods' for accessing help and source code
```

  (Asterisked methods are "Non-visible", meaning they're "not found" when you type their name on the command line, but R can find them when it needs them. If you need to view a "Non-visible" method's definition, use `getAnywhere()`, for example `getAnywhere(boxplot.formula)`.)

  If an object passed to `boxplot()` isn't a `"matrix"` or a `"formula"`, R dispatches the object to the `boxplot.default()` method. Thus `boxplot.default()` is the method that accepts `"numeric"` vectors. The `boxplot.matrix()` method is the one that accepts `"matrix"`s. So typing:

```
> boxplot(x)                    # x is a numeric vector.
```

  is equivalent to typing:

```
> boxplot.default(x)
```

  and typing:

```
> boxplot(a)                    # a is a matrix with 3 columns.
```

  is equivalent to typing:

```
> boxplot.matrix(a)
```

- *Methods must have names of the form* `fname.cname()`, where `fname()` is the name of the generic function and `cname` is the class of objects that the method accepts (or `default`).

Section 11.2 Exercises

Exercise 4 This problem concerns the function `summary()`.

a) Look at the definition of `summary()` by typing:

```
> summary
```

   How can you tell from the function definition that `summary()` is *generic*?

b) Use `methods()` to look at `summary()`'s *methods*. Is there a method for `"data.frames"`? How about for `"factors"`?

c) Try the following commands, and describe the results:

```
> x <- data.frame(x1 = c(4, 3, 6), x2 = c(4, 7, 9), x3 = c(1, 1, 3))
> summary(x)
```

```
> y <- factor(c("a", "a", "b", "c", "c", "c", "c"))
> summary(y)
```

d) In part *c*, the *generic* function `summary()` passed the object (`x` in the first case and `y` in the second) to the appropriate `summary()` *method*. Now try the following commands, which pass the objects (`x` and `y`) to the appropriate `summary()` *method* directly:

```
> summary.data.frame(x)
```

```
> summary.factor(y)
```

Compare the results with those of part *c*.

Exercise 5 In part *b* of Exercise 4, you used `methods()` to find out which *methods* are available for the *generic* function `summary()`.

We can also use `methods()` to find out which *generic* functions have *methods* for a specified *class* of objects.

a) What happens when you type the following?

```
> methods(class = "data.frame")
```

b) Based on the output in part *a*, does the *generic* function `summary()` have a *method* for `"data.frame"`s?

Exercise 6 This problem concerns `plot()` and its help file.

a) Look at the help file for `plot()`:

```
> ? plot
```

and read the 'Description'. Is `plot()` generic? Is there a `plot()` *method* for `"data frame"`s?

b) Now look for `x` under 'Arguments'. Do you think `x` could be a `"data frame"`?

c) One issue with *generic* functions is that each *method* might have arguments that would need to be passed to it through the call to the *generic* function.

Look for '`...`' under 'Arguments' in `plot()`'s help file. What does '`...`' stand for?

d) A *generic* function is actually a whole *family* of functions, each of which is called a *method*. Each method is a unique function.

We can find out what arguments a specific *method* takes by looking at it's help file. One of `plot()`'s methods is `plot.default()`. Type:

```
> ? plot.default
```

What happens?

Exercise 7 When an object `x` is passed to a *generic* function, the *generic* function calls another function, `UseMethod()`, which checks the class of `x`, and then ***dispatches*** `x` to the appropriate *method*.

Consider the following *generic* function:

```
> fun <- function(x) {
+    UseMethod("fun")
+ }
```

Suppose there are two *methods* for `fun()`:

```
> methods(fun)

[1] fun.aaa     fun.classB  fun.default
see '?methods' for accessing help and source code
```

whose definitions are:

```
> fun.aaa <- function(x) {
+   return("hello")
+ }

> fun.default <- function(x) {
+   return("goodbye")
+ }
```

a) Here's an object `my.aaa` that belongs to the class `"aaa"`:

   ```
   > class(my.aaa)

   [1] "aaa"
   ```

   Guess what the result of the following command would be (`"hello"` or `"goodbye"`).

   ```
   > fun(my.aaa)
   ```

b) Many *generic* functions have a `default` *method* to which objects are dispatched if a specific *method* doesn't exist for that class of object.

   Here's an object `my.mat` that belongs to the class `"matrix"`:

   ```
   > class(my.mat)

   [1] "matrix"
   ```

   Guess what the result of the following command would be (`"hello"` or `"goodbye"`).

   ```
   > fun(my.mat)
   ```

11.3 Writing Your Own Classes and Methods

11.3.1 S3 Classes

- There are two (main) class systems in R, the **S3 class system** and the **S4 class system**. S3 is the dominant class, and most of R's built-in classes are of the S3 type. So we'll focus on S3 classes.
- Recall that an ***attribute*** is a bit of extra information (or *metadata*) about an R object. We can check the attributes of an object using:

```
attributes()        # Determines the attributes of an object.
```

- Many R objects have a *class* attribute that indicates the class that the object belongs to.

  (If the object doesn't have a class attribute, it belongs to one of the so-called *implicit classes*, which include `"matrix"`, `"array"`, and the internal storage modes `"numeric"`, `"character"`, `"logical"`, `"list"`, `"function"`, etc., as returned by `typeof()` or `class()`).

  For example, here's a data frame:

```
> my.df

  w  x   y
1 c 23 101
2 b 22 119
3 a 19 110
```

  It has three attributes, `names`, `row.names`, and `class`:

```
> attributes(my.df)

$names
[1] "w" "x" "y"

$row.names
[1] 1 2 3

$class
[1] "data.frame"
```

  A data frame is really a list (whose elements are its columns), but with a `"data.frame"` class attribute:

```
> is.list(my.df)

[1] TRUE
```

- **An S3 class consists of a list with a class attribute.**

11.3.2 Writing Your Own S3 Classes

- Because **an S3 class consists of a list with a class attribute**, to invent a new class:
  1. Create a list object.
  2. Assign a class attribute to it (using `class()`).
- For example, suppose we want to invent a new class called `"couples"`, whose objects contain the ages of husbands and wives.

  To create a `"couples"` object, we first create a list:

```
> y <- list(
+   husband = c(31, 45, 27, 66, 51, 35, 40),
+   wife = c(29, 44, 27, 62, 48, 34, 37))
```

  and then give it the `"couples"` class attribute:

```
> class(y) <- "couples"
```

  We can check that it belongs to the newly invented class:

```
> class(y)

[1] "couples"
```

  and we can check that it has a class attribute:

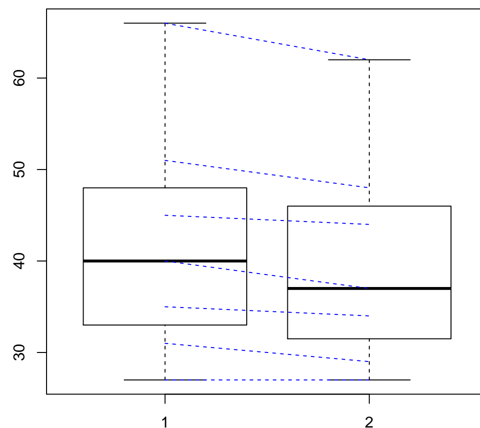
```
> attributes(y)

$names
[1] "husband" "wife"

$class
[1] "couples"
```

11.3.3 Writing Your Own S3 Methods

- To write a method that will be invoked when an an object of class `"cname"` is passed to an existing generic function `fname()`, just name the method function `fname.cname()`.
- For example, suppose we want to write a `boxplot()` method that accepts `"couples"` objects and produces a plot like the one below, with blue lines connecting spouse's ages:



To write the method, we simply create a function named `boxplot.couples()` that does what we want when passed a `"couples"` object:

```
> boxplot.couples <- function(x) {
+   h <- x[["husband"]]
+   w <- x[["wife"]]
+   n <- length(h)
+   boxplot.default(h, w, names = c("Husband", "Wife"), ylab = "Age")
+   segments(rep(1, n), h, rep(2, n), w, col = "blue", lty = "dashed")
+ }
```

(The `segments()` function is used to add the dashed lines connecting spouse's ages.)

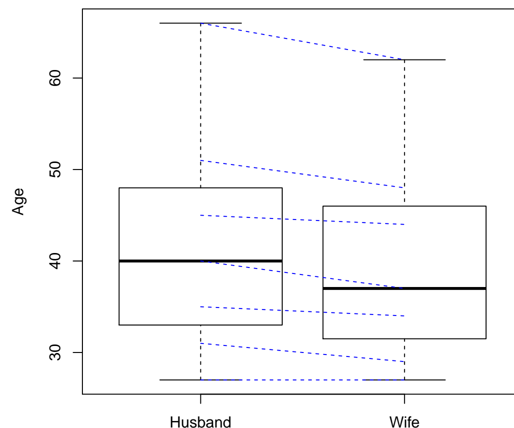
Now there's a `boxplot()` method for `"couples"` objects:

```
> methods(boxplot)

[1] boxplot.couples  boxplot.default  boxplot.formula* boxplot.matrix
see '?methods' for accessing help and source code
```

and when we pass a `"couples"` object to the generic `boxplot()` function, `boxplot()` dispatches it to the `boxplot.couples()` method automatically:

```
> boxplot(y)
```



Section 11.3 Exercises

Exercise 8 We want to define a new (S3) *class* named `"diet"`. An object belonging to this class should be a list with two elements, the person's weight before a dietary weight loss program, and `WtAfter`, the person's weight after.

Write commands that create a `"diet"` object. **Hint**: First use `list()` to create a list containing the two elements `WtBefore`, and `WtAfter` (both `"numerical"` vectors). Then use `class()` to assign `"diet"` as the class attribute of the list.

Exercise 9 We want to define a new (S3) *method* for the *generic* function `mean()` that operates on `"diet"` objects (from Exercise 8). When passed a `"diet"` object, `mean()` should dispatch the object to the `"diet"` method, which should return a vector with two elements, the first the average of `WtBefore` and the second the average of `WtAfter`.

Write commands that create the `mean()` *method* for `"diet"` objects. **Hint**: All you need to do in order for your function to be a *method* for `mean()` is to give it the name `mean.diet()`.

Make sure to test your `mean()` *method* by passing it a `"diet"` object (such as the one you created in part *a*).

11.4 Using Inheritance

- We can use an *existing* class to create a more specialized class whose objects contain extra list elements.
- For example, suppose we have in mind a class called `"families"` whose objects contain the ages of husbands and their wives, and also the number of children they have.

Below, we create a "families" object by creating a list having the desired elements and then assigning it the "families" class attribute.

```
> z <- list(
+   husband = c(31, 45, 27, 66, 51, 35, 40),
+   wife = c(29, 44, 27, 62, 48, 34, 37),
+   children = c(1, 2, 0, 3, 2, 0, 0))
```

Note though that "families" objects are a *subclass* of "couples" objects because they do contain the husbands' and wives' ages (*and* the number of children). Thus we assign them *both* class attributes:

```
> class(z) <- c("families", "couples")
```

We can check that the newly created object belongs to both classes:

```
> class(z)

[1] "families" "couples"
```

and we can look at its attributes:

```
> attributes(z)

$names
[1] "husband"  "wife"     "children"

$class
[1] "families" "couples"
```

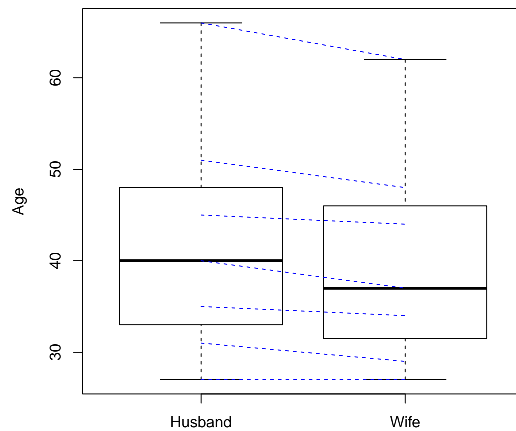
- New classes created from existing ones by adding new list elements, and then assigning *both* class names as the class attribute, as above when "families" was created from "couples", ***inherit*** not only the existing class's *list elements* but also the *methods* associated with that class.

  What this means is that when an object belonging to the *new* class is passed to a generic function, UseMethod() first looks for a method for that *new* class. If it doesn't find one, it dispatches the object to the method for the *existing* class.

- For example, any method written for "couples" objects will also accept "families" objects.

  When we pass z (which is a "families" object) to boxplot() (which doesn't have a "families" method but *does* have a "couples" method), it gets dispatched to the "couples" boxplot() method:

```
> boxplot(z)
```



Above, `boxplot()` calls `useMethod()`, which first searches for a `"families"` method for `boxplot()` (because `"families"` is first in the list of classes that `y` belongs to), and then, upon finding none, searches for a `"couples"` method, which it finds and passes it `z`.

Section 11.4 Exercises

Exercise 10 Suppose an object `y` belongs to *two* classes of objects, `"classA"` and `"classB"`, and `class()` returns:

```
> class(y)

[1] "classA" "classB"
```

When the object is passed to a *generic* function, the *generic* function calls `UseMethod()`, which looks first for a `"classA"` method to which it can dispatch the object. If no `"classA"` method exists, it looks for a `"classB"` method. If no `"classB"` method exists, it looks for a `default` method.

Suppose also that a *generic* function `fun()` has two methods:

```
> methods(fun)

[1] fun.classB  fun.default
see '?methods' for accessing help and source code
```

In the function call:

```
> fun(y)
```

which *method*, `fun.classB` or `fun.default`, will be used?

11.5 S4 Classes

- S4 classes incorporate some ”safety” measures that prevent accidents such as the following from happening:
  - When creating a `"couples"` object, we forget to enter the wives ages.
  - We misspell *husband* as *busband*.
  - We create an object of some class other than `"couples"` but accidentally set its class attribute to `"couples"`.
- The table below shows an overview of the differences between S3 and S4 classes:

Operation	S3	S4
Define class	Implicit on constructor code	`setClass()`
Create object	Build list using `list()`, set class using `class()`	`new()`
Reference member element	`$`	`@`
Define a method	Define `fname.cname()`	`setMethod()`
Define a generic	`UseMethod()`	`setGeneric()`

12 Packages and the Search Path

12.1 Packages

- A ***package*** is a collection of specialized functions (or other R objects) that are stored together in a bundle.

  Some packages are built into R. Others need to be downloaded and installed from the Internet.
- To see which packages are built-in, type:

```
> search()

[1] ".GlobalEnv"        "package:stats"     "package:graphics"  "package:grDevices"
[5] "package:utils"     "package:datasets"  "package:methods"   "Autoloads"
[9] "package:base"
```

  As can be seen, the packages `stats`, `graphics`, `grDevices`, etc. are built-in.
- To download and install a non-built-in package, use:

```
install.packages()     # Install a package from the web.  You can also
                       # use the pull-down menu in the R console.
```

  (You can also install a package from R’s pull down menu.)
- For example, to install the `"leaps"` package, type:

```
> install.packages("leaps")
```

(You'll be prompted to choose a CRAN mirror from which to download the package.)

- After installing a package, you need to load it into the current R session using either of the functions:

```
library()       # Loads a package into the current R session.  Must be
                # called from the command line.
require()       # Loads a package into the current R session.  Can be
                # called from within a function.
```

- Both `library()` and `require()` load a previously installed package into the current R session. `require()` is more versatile because it can be called from within a function, whereas `library()` can only be called from the R command line.

  As an example, to load the `"leaps"` package, type:

```
> library(leaps)
```

- Once a package has been loaded into the current R session, all of its functions and other objects will be available for use.
- You can see what objects are contained in a package by looking at it's help file, for example by typing

```
> help(package = leaps)
```

12.2 The Search Path

- When a package is loaded into the current R session, the objects it contains are placed into an *environment*. Recall that an ***environment*** is a "container" that holds R objects. Two different objects can share the same name (e.g. `x`) as long as they're stored in different *environments*. So placing a package's objects into their own *environment* avoids the possibility of existing objects being over-written when the package is loaded.
- To understand how R looks for objects when it needs them, it helps to understand the *search path*. The ***search path*** is a sequence of environments through which R searches when it encounters the name of an object while executing a command.
- When R encounters the name of an object in a command, it searches one environment after another in the order specified by the *search path* until it either finds the object or reaches the end of the search path, in which case it prints an error message:

```
Error: object 'ffrg' not found
```

- Typing `search()` shows the current search path:

```
search()              # Shows the current search path.
```

- For example:

```
> search()

[1] ".GlobalEnv"        "package:stats"     "package:graphics"  "package:grDevices"
[5] "package:utils"     "package:datasets"  "package:methods"   "Autoloads"
[9] "package:base"
```

This indicates that when R encounters the name of an object, it first searches is the *Global Environment* (aka the Workspace). If it doesn't find the object there, it searches the environment associated with the `"stats"` package, then the one associated with the `"graphics"` package, and so on.

- When a package is downloaded from the Internet and then loaded into the current R session, R places the package in the 2nd position in the search path, right after the Global Environment (Workspace). For example, after installing the `"leaps"` package, we get:

```
> search()

> search()
 [1] ".GlobalEnv"        "package:leaps"     "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"     "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

Notice that `"leaps"` is in the 2nd position.

- If a user-defined object (which will be stored in the Workspace) shares the same name as one in a package, when the package is loaded R prints a message indicating that the object in the package is ***masked*** by the one in the Workspace.

For example, the `"leaps"` package has a function called `regsubsets()`. If we create a function of the same name in the Workspace, and then load `"leaps"`, we get:

```
> regsubsets <- function(x) return(x^2)

> library(leaps)

Attaching package: 'leaps'

The following object is masked _by_ '.GlobalEnv':

    regsubsets
```

This means that when R encounters the name `regsubsets` while executing a command, it will use the user-defined version of `regsubsets()`, not the one that's in the `"leaps"` package.

13 Input/Output

13.1 Accessing the Keyboard and Monitor

- R has several built-in functions for accessing the keyboard and monitor, among them:

```
scan()      # Read data from the keyboard or a file to a vector or list
readline()  # Read a line of input from the keyboard
print()     # Print an object to the R console
cat()       # Prints objects to the R console or to a file, "concat-
            # enating" (combining) them if there are more than one
```

13.1.1 Using `scan()`

- `scan()` will read data from a file or the keyboard into a *vector*
- To input data from the keyboard, just type `scan()`. You'll be prompted to input values one at a time, and you should either hit 'Enter' or insert a space after each one:

```
> scan()
1: 35
2: 33
3: 21
4: 44
5: 19
6:

Read 5 items
[1] 35 33 21 44 19
```

  Hitting 'Enter' without inputting a value terminates the process.

- Of course in practice you'll want to save the values to a *vector*, for example:

```
> x <- scan()
```

13.1.2 Using `readline()`

- If you want to read a single line from the keyboard, we type `readline()` followed by the line we want to input:

```
> x <- readline()
Here's a line of user input

> x

[1] "Here's a line of user input"
```

- An optional argument `prompt` allows us to prompt the user for input:

```
> init <- readline(prompt = "Enter your initials: ")
Enter your initials: NG

> init

[1] "NG"
```

13.1.3 Using `print()`

- The `print()` function prints its argument to the R console:

```
> x <- 1:3
> print(x)

[1] 1 2 3
```

(Actually, if you just type the name of an object on the command line and hit 'Enter', R calls the function `print()`, passing it the object whose name you typed.)

- `print()` is a *generic* function, so its output to the console depends on what class of argument is passed to it.

13.1.4 Using `cat()`

- Unlike `print()`, `cat()` can print more than one expression at a time, and it combines (or "concatenates") them when there are more than one:

```
> cat("The vector x contains the values", x, "\nTheir squares are", x^2)

The vector x contains the values 1 2 3
Their squares are 1 4 9
```

(Recall that `\n` is the newline character.)

- In addition, `cat()` has an optional argument `file` that allows us to print to a file (or more generally, to a *connection*) instead of to the R console.
- By default, `cat()` separates its printed output by spaces. The optional argument `sep` can be used to specify other separators, e.g. commas (`sep = ","`) or no separation at all (`sep = ""`).

Section 13.1 Exercises

Exercise 11 After typing:

```
> x <- scan()
```

type in a few values, hitting 'Enter' after each one. Then hit 'Enter' (without typing in

a value) to terminate the process. What does this do?

Exercise 12 After typing:

```
> x <- readline()
```

type in line of text, hitting 'Enter' when you're done. What does this do?

Exercise 13 `print()`, `scan()`, and `cat()` can be used inside functions for user interactions.

a) Guess what the call to `my.fun1()` will do, then check your answer.

```
> my.fun1 <- function() {
    print("Input three numbers")
    x <- scan(n = 3)
    cat("The mean of your numbers is", mean(x), "\n")
  }

> my.fun1()
```

(Recall that `\n` is the "new line" character.)

b) Guess what the call to `my.fun2()` will do, then check your answer.

```
> my.fun2 <- function() {
    x <- readline(prompt = "Enter your favorite color: ")
    cat("Your favorite color is", x, "\n")
  }

> my.fun2()
```

13.2 Getting Directory Information

- The ***working directory*** is the default folder to which R saves files and searches for them when they're referred to by their simple file name (e.g. `scan('myfile.txt')`).

  We can get or set the current working directory using:

```
getwd()    # Get the current working directory
setwd()    # Set the current working directory
```

- To determine the current working directory, just type `getwd()` with no arguments:

```
> getwd()
```

```
"C:/Users/Username/Documents"
```

- If we refer to a file, '**myfile.txt**', say, for example in:

  ```
  > my.data <- read.table('myfile.txt')
  ```

  R looks for it in the working directory, in this case '**C:/Users/Username/Documents**'. Likewise, if we write to '**myfile.txt**' using, say:

  ```
  > cat("x", "y", "z", file = 'myfile.txt')
  ```

  R creates (or overwrites) '**myfile.txt**' in the working directory, '**C:/Users/Username/Documents**'.

- To change the working directory to, say, '**C:/Documents and Settings/Data**', use `setwd()`:

  ```
  > setwd('C:/Documents and Settings/Data')
  ```

Section 13.2 Exercises

Exercise 14

a) Suppose `getwd()` reports the following:

```
> getwd()

[1] "C:/Users/"
```

If we try to read in data from a file named '**myfile.txt**' by typing:

```
> read.table('myfile.txt')
```

(without specifying a folder or directory), where (i.e. in what folder or directory) will R look for the file?

b) Suppose we use `setwd()` to change the working directory by typing:

```
> setwd('D:/MyFolder')
```

and type:

```
> read.table('myfile.txt')
```

Now where will R look for the file?

13.3 Reading and Writing From/To Files

- We sometimes need to:

  1. **Read data** from or write it to a file.

2. **Read R commands** from (or write them to) a file.
3. **Write R output** to a file.

R has several built-in functions for performing these tasks:

```
read.table(),   # Read data from a text (.txt) file into a data frame
write.table()   # Write data from a data frame (or matrix) to a text
                # (.txt) file
read.csv()      # Read data from a 'comma separated value' (.csv) file
                # into a data frame
write.csv()     # Write data from a data frame (or matrix) to a 'comma
                # separated value' (.csv) file
readLines()     # Read from a text file, one line at at time, into a
                # "character" vector
writeLines()    # Write data from a "character" vector into lines of a
                # file
scan()          # Read data from the keyboard or a file to a vector or
                # list
cat()           # Prints objects to the R console or to a file, "con-
                # catenating" (combining) them if there are more than
                # one
```

13.3.1 Reading and Writing Data From/To Files

Using `read.table()`, `write.table()`, `read.csv()`, and `write.csv()`

- `read.table()` is used to read data from a text (.txt) file into a data frame. `read.csv()` is similar, but reads from a 'comma separated value' (.csv) file.
- `write.table()` and `write.csv()` do the opposite, they write data from a data frame (or matrix) to a text (or .csv) file. For example, to write the following `mice` data frame:

```
> mice

  Color Weight Length
1 white     23    3.8
2  grey     21    3.7
3 black     18    3.0
4 brown     26    3.4
5 black     25    3.4
6 white     22    3.1
7 black     26    3.5
8 white     19    3.2
```

to the text file '**C:\myfiles\micedatafile.txt**', type:

```
> write.table(mice, file = 'C:/myfiles/micedatafile.txt')
```

The file '**C:\myfiles\micedatafile.txt**' now contains:

	"color"	"weight"	"length"
"1"	"purple"	23	3.8
"2"	"yellow"	21	3.7
"3"	"red"	18	3
"4"	"brown"	26	3.4
"5"	"green"	25	3.4
"6"	"purple"	22	3.1
"7"	"red"	26	3.5
"8"	"purple"	19	3.2

- If we don't want the quotes, we specify `quote=FALSE`, and if we don't want the row numbers, we specify `row.names=FALSE`. Thus typing:

```
> write.table(mice, file = 'C:/myfiles/micedatafile.txt',
              quote = FALSE, row.names = FALSE)
```

produces the file '**C:\myfiles\micedatafile.txt**' containing:

color	weight	length
purple	23	3.8
yellow	21	3.7
red	18	3
brown	26	3.4
green	25	3.4
purple	22	3.1
red	26	3.5
purple	19	3.2

Using `readLines()` and `writeLines()`

- The `readLines()` function will read one line at a time from a text file, placing each line into a separate element of a `"character"` vector.
- For example, if the file '**C:\myfiles\textlines.txt**' contains the poem:

```
'Hope' is the thing with feathers
That perches in the soul
And sings the tune without the words
And never stops at all
```

then typing:

```
> poem <- readLines('C:/myfiles/textlines.txt')
```

results in the `"character"` vector with four elements:

```
> poem

[1] "'Hope' is the thing with feathers"    "That perches in the soul"
[3] "And sings the tune without the words" "And never stops at all"

> is.vector(poem)

[1] TRUE

> length(poem)

[1] 4
```

- `writeLines()` does the opposite – it writes elements of a `"character"` vector to a file, each element of the `"character"` vector going onto its own line of the file.

Using `scan()`

- When `scan()` is used to read data from a file to a vector, it assumes vector elements are separated in the file by "white-space", which includes
  - Blank spaces
  - Tabs
  - Carriage return/newline indicators
- For example, suppose a file '**C:\myfiles\mydata.txt**' contains the following three lines:

```
24 17 32
44   19      20
45
```

  Then typing:

```
> x <- scan('C:/myfiles/mydata.txt')

Read 7 items
```

  would result in the vector:

```
> x

[1] 24 17 32 44 19 20 45
```

- By default, `scan()` assumes the data to be read are numeric. If they're text (characters), we need to specify `what=""`, which tells R that the values to be read are characters.

- For example, suppose we have a text file '**C:\myfiles\mytext.txt**' containing the following three lines:

```
dog cat gerbil hamster
parakeet goldfish
iguana
```

We read the data into a vector `pets` by typing:

```
> pets <- scan('C:/myfiles/mytext.txt', what = "")

Read 7 items

> pets

[1] "dog"      "cat"      "gerbil"   "hamster"  "parakeet" "goldfish" "iguana"
```

- An optional argument `sep` can be used to specify other delimiters besides "white-space". For example, if the data values are separated by commas in the input file, we use `sep=","`. If they're separated by carriage return/newline indicators, we use `sep="\n"`.

Using `cat()`

- To use `cat()` to write to a file, specify the filename via the argument `file` and separate the items to be written to the file by commas. For example typing

```
> cat(file = 'C:/myfiles/mynewdata.txt', "a", 2, 14, "b")
```

would produce a file '**C:\myfiles\mynewdata.txt**' containing the line:

```
"a"  2  14  "b"
```

- An optional argument `append` can be used in `scan()` to append a line to an existing file.

13.3.2 Writing R Output to a File

- To write R output to a file (or more generally, to a *connection*), we use:

```
sink()          # Write R output to a file
```

- `sink()` writes all subsequent R output to a specified text (.txt) file, until we tell R to back revert to printing it to the console (i.e. until we close the *connection* to the file). For example, typing:

```
> squeaky <- c("The squeaky wheel", "gets the oil")

> sink('C:/Routput.txt')

> squeaky
> matrix(1:6, nrow = 2, ncol = 2)
> 2 + 2
```

produces a file, '**C:\Routput.txt**', containing:

```
[1] "The squeaky wheel" "gets the oil"
      [,1]  [,2]
[1, ]    1     3
[2, ]    2     4
[1] 4
```

To close the *connection* to the file and return to printing to the R console, type:

```
> sink()
```

with no arguments.

13.3.3 Reading and Writing R Commands From/To a File

- The functions below are useful for reading R commands from a (text) file, writing commands to a file, and writing R output to a file:

```
source()      # Read and execute R commands from a file
dump()        # Write an R-code representation of an object to a file
```

- `dump()` will write an R-code representation of an object to a file. For example, consider the matrix `x` below:

```
> x

     [,1] [,2] [,3]
[1,]    4    5    2
[2,]    3    8    0
[3,]    7    8    9
```

Typing:

```
> dump("x", 'C:/xcode.txt')
```

creates the text file '**C:\xcode.txt**' containing the following:

x <- structure(c(4, 3, 7, 5, 8, 8, 2, 0, 9), .Dim = c(3L, 3L))

Although it may not look like it, executing the command above will recreate the matrix `x`.

- `source()` will read R commands from a file and execute them in the current R session. For example, after removing the matrix `x` created above:

```
> rm(x)
```

watch what happens after we type:

```
> source('C:/xcode.txt')          # Executes the commands in 'C:/xcode.txt'
```

We get:

```
> x

     [,1] [,2] [,3]
[1,]    4    5    2
[2,]    3    8    0
[3,]    7    8    9
```

Section 13.3 Exercises

Exercise 15 Create a text (.txt) file containing the following three lines:

3 2 6
4 9 8
1 7 5

a) Now read the data into R using `scan()`. **Hint**: You can obtain the pathname (folder) for the file by typing:

```
> file.choose()
```

then selecting the file in the dialog box. This pathname can then be specified in `scan()`.

b) If we had read the data in using `read.table()`, the resulting object would have been a *data frame*. What type of object does `scan()` produce?

Exercise 16 Create a text (.txt) file containing the following three lines:

3 7 2
4 9 0
1 5 8

Save the file as, say, **myfile.txt** in the *working directory* (which can be determined by typing `getwd()`).

Now read the contents of the file into R using the following three methods, and report the differences in the results:

- Using `read.table()`, i.e.

  ```
  > x <- read.table('myfile.txt')
  > x
  ```

- Using `readLines()`, i.e.

  ```
  > y <- readLines('myfile.txt')
  > y
  ```

  (You may get a warning message if you didn't hit 'Enter' at the end of the last line in your text file. You can ignore the warning or fix the text file.)

- Using `scan()`, i.e.

  ```
  > z <- scan('myfile.txt')
  > z
  ```

Exercise 17 Create a text (.txt) file containing the following lines of R code:

```
my.x <- numeric(0)
for (i in 1:5) {
  my.x[i] <- 5 * i
}
```

Save the file as, say, **myRcode.txt** in the *working directory* (which can be determined by typing `getwd()`).

Now execute the lines of code by typing:

```
> source('myRcode.txt')
```

and confirm that they were executed by typing:

```
> my.x
```

13.4 Accessing Files on Remote Machines via URLs

- Several of R's functions for reading data from files (`read.table()`, `read.csv()`, `scan()`, etc.) can read from files stored on remote computers that are connected to the Internet.

- To read from files on remote computers, we specify the URL for the file.

  For example, the file '**micedata.txt**' that's posted on the course website is stored on a remote computer, and can be accessed by typing:

  ```
  > x <- read.table('http://mcs.msudenver.edu/__metadot__/attachment/download/8347',
                    header = TRUE)
  ```

  The URL address was obtained by right clicking the link to the file on the course website, and then selecting 'Properties'.

13.5 Introduction to Connections

- Reading from and writing to files is just one of the ways data can be inputted and outputted in R.
- The term ***connection*** generalizes the notion of a file, and can refer to any of several ways data can be inputted and outputted, including not only files but also URLs, compressed files, and some others.
- *Connections* can be ***opened*** and ***closed***. When they're opened, R can pass data to and from them. When they're closed, R can't read from or write to them without opening them first.

  Built-in functions like `read.table()` and `scan()` open a connection to a file, then close the connection when they're done using the file.

- We'll look at two types of connections, ***files*** and ***URLs***:

  ```
  file()        # Create (but don't open) a connection to a file
  url()         # Create (but don't open) connection to a URL
  open()        # Open a connection for reading, writing, etc.
  close()       # Close a connection
  ```

  You can see the complete list of the different types of connections by typing:

  ```
  > ? connection
  ```

- One use of a connection is to read "chunks" of data from a file, one "chunk" at a time. For example, consider a text file '**C:\people.txt**' containing:

john	25	7
kim	22	9
karen	38	8
mark	26	5
bill	45	8
anne	27	5
jennifer	46	9
otis	19	9

To read the data into R, **four lines at a time**, type:

```
> con <- file('C:/people.txt')                   # Create a connection
> open(con, "r")                                 # Open the connection for "reading"
> first.four.lines <- readLines(con, n = 4)
> next.four.lines <- readLines(con, n = 4)
> close(con)                                     # Close the connection
```

The result is:

```
> first.four.lines

[1] "john 25 7"  "kim 22 9"   "karen 38 8" "mark 26 5"

> next.four.lines

[1] "bill 45 8"     "anne 27 5"     "jennifer 46 9" "otis 19 9"
```

The important point is that **R was able to remember which line to start at** when it encountered the second call to `readLines()`.

If we hadn't opened the connection, and instead just typed:

```
> first.four.lines <- readLines('C:/people.txt', n = 4)
> next.four.lines <- readLines('C:/people.txt', n = 4)
```

R would've read the *same four lines* in both calls to `readLines()`.

- We use `url()` to open a connection to a file on a remote computer via its URL.

  For example, consider the file '**micedata.txt**' that's located on a remote computer as described in Subsection 13.4.

  The following command opens a connection to the file, reads the first four lines, and then closes the connection:

```
> con <- url('http://mcs.msudenver.edu/__metadot__/attachment/download/8347')
> open(con, "r")
> readLines(con, n = 5)
> close(con)
```