Contents

1 The `summary()` command

`summary()` is a generic but very useful, function to summarize many types of R objects, including datasets.

When used on a dataset, summary returns distributional summaries of variables in the dataset.

2 Creating Data with R

2.1 Data Import

It is necessary to import outside data into R before you start analysing it. Here we will look at some relevant issues.

2.1.1 Microsoft XLS File

Very often, the sample data is in MS Excel™ format, and needs to be imported into R prior to use. For this, we could use the `read.xls()` function from the ***gdata*** package. The command reads from an Excel spreadsheet and returns a data frame. The following shows how to load an Excel spreadsheet named "mydata.xls". As the package is not in the core R library, it has to be installed and loaded into the R workspace. (This spreadsheet is saved in the working directory).

```
> library(gdata)                  # load the gdata package
> help(read.xls)                  # documentation
> mydata = read.xls("mydata.xls")  # read from first sheet
```

2.1.2 CSV File

The sample data can be in ***comma separated values*** (CSV) format. Each cell inside such data file is separated by a special character, which usually is a comma, although other characters can be used as well. (As it is vendor neutral, CSV is more versatile). The first row of the data file should contain the column names instead of the actual data. Here is a sample of the expected format.

```
Col1,Col2,Col3
100,a1,b1
200,a2,b2
300,a3,b3
```

After we copy and paste the data above in a file named "mydata.csv" with a text editor, we can read the data with the `read.csv()` function. (The file should be in the working directory).

```
> mydata = read.csv("mydata.csv")  # read csv file
> mydata                          # print data frame
  Col1 Col2 Col3
1  100   a1   b1
2  200   a2   b2
3  300   a3   b3
```

(Remark: In various European locales, as the comma character serves as decimal point, the `read.csv2()` function should be used instead.)

Exercise: Load the GNW dataset into the R Environment. (We will do more work with it shortly).

2.1.3 Data export - `write.csv()`

The basic tool to produce output files is `write.csv()`.

The only two required arguments to `write.csv()` are ,firstly, the name of a dataset or matrix (with just a single argument) then the name (in quotations) of the file to be created.

```
> getwd()
[1] "C:/Users/Kevin/Documents"
> write.csv(iris,"iris2")
>
```

The `sep=` argument can be used to specify a separator other than a blank space. Using sep=',' (comma separated) or sep='\t' (tab-separated) are two common choices.

2.1.4 Data export - `sink()`

The code first instructs R to re-direct output away from the R terminal to the file "output.txt" and then the relevant output (below the summary of the GNW data frame) which goes to the sink. To close the sink, use `sink()` with no arguments.

```
 sink("GNW.txt")
 summary(GNW)
sink()
```

While a sink is open all output will go to it, replacing what is already in the file. To append output to a file, use the `append=TRUE` option with `sink()`.

(Here we will use the `describe()` command from the ***psych*** packages)

```
install.packages("psych")
library(psych)

sink("GNW.txt",append=TRUE)
describe(GNW)
sink()
```

2.2 Inputting an R script - the `source()` command

The `source()`function runs a previously written R script in the current session. If the filename does not include a path (such as C:/WorkArea) the file is taken from the current working directory. This is a particularly useful for loading pre-written data or functions (from later on), and other items of code into the current session.

```
# input a few scripts
source("mypackages.R")
source("myfunctions.R")
source("mydata.R")
```

2.3 Using the `scan()` command

The`scan()` function is a useful method of inputting data quickly. You can use to quickly copy and paste values into the R environment. It is best used in the manner as described in the following example. Create a variable X and use the `scan()` function to populate it with values. Type in a value, and then press return. Once you have entered all the values, press return again to return to normal operation.

```
> X=scan()
1: 4
2: 5
3: 5
4: 6
5:
Read 4 items
```

For the data that has been scanned, equivalent code that would be used to define it can be retrieved and recorded using the `edit()` command.

2.4 Using the `scan()` command to input character data

Previously we have seen the `scan()` command used to quickly input numeric data. The command can also be used to input character data. The addition argument , `what=" "`, must be used. Create a variable grouping that comprises, in order, five As and then six Bs.

```
# inputting character data

grouping = scan(what=" ")

# Enter values
# Hit return again when you have finished.
```

2.5 Spreadsheet Interface

R provides a spreadsheet interface for editing the values of existing data sets. We use the command `data.entry()` , and name of the data object as the argument.

```
> data.entry(X) # Edit the data set and exit interface
> X
```

3 Vectors and Sequences

3.1 Vectors

A vector in R is a container vector, a statistician's collection of data, not a mathematical vector. The R language is designed around the assumption that a vector is an ordered set of measurements rather than a geometrical position or a physical state. (R supports mathematical vector operations, but they are secondary in the design of the language.) This helps explain, for example, R's otherwise inexplicable vector recycling feature.

Adding a vector of length 22 and a vector of length 45 in most languages would raise an exception; the language designers would assume the programmer has made an error and the program is now in an undefined state. However, R allows adding two vectors regardless of their relative lengths.

3.1.1 Recycling

The elements of the shorter summand are recycled as often as necessary to create a vector the length of the longer summand. This is not attempting to add physical vectors that are incompatible for addition, but rather a syntactic convenience for manipulating sets of data. (R does issue a warning when adding vectors of different lengths and the length of the longer vector is not an integer multiple of the length of the shorter vector. So, for example, adding vectors of lengths 3 and 7 would cause a warning, but adding vectors of length 3 and 6 would not.)

The R language has no provision for scalars, nothing like a double in C-family languages. The only way to represent a single number in a variable is to use a vector of length one. And while it is possible to iterate through vectors as one might do in a for loop in C, it is usually clearer and more efficient in R to operate on vectors as a whole.

3.2 Creating Vectors

Vectors are created using the c function. For example, `p <- c(2,3,5,7)` sets p to the vector containing the first four prime numbers.

```
p <- c(2,3,5,7)
```

- Elements of a vector can be accessed using the square bracket operators []. So in the above example, `p[3]` is 5.
- Vectors automatically expand when assigning to an index past the end of the vector.
- Negative indices are legal, but they have a very different meaning than in some other languages. If x is an array in Python or Perl, x[-n] returns the nth element from the end of the vector. In R, x[-n] returns a copy of x with the nth element removed.

```
p <- c(2,3,5,7)
p[2]
p[-2]
```

```
> p <- c(2,3,5,7)
> p
[1] 2 3 5 7
> p[2]
[1] 3
> p[-2]
[1] 2 5 7
```

4 Vectors

Vectors are the simplest type of object in R. There are 3 main types of vectors:

- Numeric vectors
- Character vectors
- Logical vectors

To set up a numeric vector x consisting of 7 numbers; 10, 5, 3, 6, 21,11,41, we use the `c()` command to concatenate them i.e create a vector of individual values. To print the contents of x, simply type x

```
x <- c(10, 5, 3, 6, 21,11,41)
```

```
> x <- c(10, 5, 3, 6, 21,11,41)
> x
[1] 10 5 3 6 21 11 41
```

The [1] in front of the result is the index of the first element in the vector x. (The single value variables from earlier on are simply vectors containing one element). To access a particular element of a vector, and the position of the element enclosed in square brackets.

(End of Edit - Move to Reserve - Vectors)

4.1 Useful Commands For Vectors

`Newvec = c(13,16,36,55,23,11)`

- `sort(Newvec)` - sort data set in ascending order
- `rev(Newvec)` - reverse the data set order
- `rep(Newvec,n)` - replicate the data set n times
- `rep(Newvec,each=n)` - replicate each element of the data set $n-$times
- `diff(Newvec)` - sequential difference of each element
- `order(Newvec)`
- `rank(Newvec)`

```
> Newvec = c(13,16,36,55,23,11)
>
> sort(Newvec)
[1] 11 13 16 23 36 55
> rev(Newvec)
```

```
[1] 11 23 55 36 16 13
>
> rep(Newvec,2)
 [1] 13 16 36 55 23 11 13 16 36 55 23 11
> rep(Newvec,3)
 [1] 13 16 36 55 23 11 13 16 36 55 23 11 13 16 36 55 23 11
>
> rep(Newvec,each=3)
 [1] 13 13 13 16 16 16 36 36 36 55 55 55 23 23 23 11 11 11
> diff(Newvec)
[1]   3  20  19 -32 -12
> order(Newvec)
[1] 6 1 2 5 3 4
>
> rank(Newvec)
[1] 2 3 5 6 4 1
```

4.2 Sequences

4.2.1 Using the colon operator

A ‘count-up’ or a ‘count-down’ sequence of integers will be determined automatically. This operator is very useful and we will make use of it frequently.

```
1:20
20:1
10:20
```

4.2.2 Using the `seq()` operator

Firstly we will mimic the sequences that we have created using the colon operator.

```
seq(1,20)
seq(20,1)
```

5 Indexing and Subsetting

5.1 Relational and Logical Operators

Relational operators allow for the comparison of values in vectors.

greater than	>
less than	<
equal to	==
less than or equal to	<=
greater than or equal to	>=
not equal to	! =

Note the difference of the equality operator "==" with assignment operator "=".

& and && indicate logical AND The shorter form performs element-wise comparisons in much the same way as arithmetic operators. The longer form is appropriate for programming control-flow and typically preferred in "if" clauses.

- We can use relational operators to subset vectors (as well as more complex data objects such as data frames, which we will meet later).
- We specify the relational condition in square brackets.
- We can construct compound relational conditions too, using logical operators

```
> vec=1:19
> vec[vec<5]
[1] 1 2 3 4
> vec[(vec<6)|(vec>16)]
[1]  1  2  3  4  5 17 18 19
```

5.2 Conditional Subsetting

`The Subset command`

5.3 Selection using the Subset Function

The subset( ) function is the easiest way to select variables and observeration. In the following example, we select all rows that have a value of age greater than or equal to 20 or age less then 10. We keep the ID and Weight columns.

6 Data Frames

Another way that information is stored is in data frames. This is a way to take many vectors of different types and store them in the same variable. The vectors can be of all different types. For example, a data frame may contain many lists, and each list might be a list of factors, strings, or numbers.

There are different ways to create and manipulate data frames. Most are beyond the scope of this introduction. They are only mentioned here to offer a more complete description.

6.1 Data Frames

Technically, a data frame in R is a very important type of data object; a type of table where the typical use employs the rows as observations (or cases) and the columns as variables. Inter alia, a data frame differs from a matrix in that it can contain character values. Many data sets are stored as data frames. Let us consider the following two variables; age and height.

```
> age=18:29
> age
[1] 18 19 20 21 22 23 24 25 26 27 28 29
```

In similar fashion, we entered the average heights in a vector called height.

```
>height=c(76.1,77,78.1,78.2,78.8,79.7,79.9,81.1,81.2,81.8,82.8,83.5)
> height
[1] 76.1 77.0 78.1 78.2 78.8 79.7 79.9 81.1 81.2 81.8 82.8 83.5
```

We will now use R's `data.frame()` command to create our first data frame and store the results in the data frame "village".

```
> village=data.frame(age=age,height=height)
```

How do we access the data in each column? One way is to state the variable containing the data frame, followed by a dollar sign, then the name of the column we wish to access (as with Lists earlier). For example, if we wanted to access the data in the "age" column, we would do the following:

```
 > village$age
 [1] 18 19 20 21 22 23 24 25 26 27 28 29
```

The additional typing required by the "dollar sign" notation can quickly become tiresome, so R provides the ability to "attach" the variables in the dataframe to our workspace.

```
> attach(village)
```

Let's re-examine our workspace. (The `ls()` command lists all data objects in the workspace)

```
> ls()
[1] "village"
```

No evidence of the variables in the workspace. However, R has made copies of the variables in the columns of the data frame, and most importantly, we can access them without the "dollar notation." (later)

```
> age
 [1] 18 19 20 21 22 23 24 25 26 27 28 29
> height
 [1] 76.1 77.0 78.1 78.2 78.8 79.7 79.9 81.1 81.2 81.8 82.8
[12] 83.5
```

Previously we have seen `rownames()` and `colnames()` to determine the names from an existing data frame. We can use these commands to create names for a new data frame also.

7 Matrices

7.1 Matrices

7.1.1 Creating Matrices

```
A=matrix(c(1,-2,0,3,0,-1),nrow=2,byrow=TRUE)
B=matrix(c(4,1,0,2,-1,3),nrow=3,byrow=TRUE)
C=matrix(c(2,1,0,-3),nrow=2,byrow=TRUE)
```

7.2 Creating a matrix

Matrices can be created using the matrix() command.

The arguments to be supplied are 1. vector of values to be entered. 2. dimensions of the matrix, specifying either the numbers of rows or columns. Additionally you can specify if the values are to be allocated by row or column. By default they are allocated by column.

```
Vec1 = c(1,4,5,6,4,5,5,7,9)  # 9 elements

A = matrix(Vec1,nrow=3)      #3 by 3 matrix, assigned by column.

A

#      [,1] [,2] [,3]
# [1,]    1    6    5
# [2,]    4    4    7
# [3,]    5    5    9
```

Notice how the rows and column are preceded with row and column indexes. To assign by row, we must specify it by setting the appropriate argument accordingly.

```
#3 by 3 matrix. Values assigned by row.
C= matrix(  c(1,6,7,0.6,0.5,0.3,1,2,1), ncol=3 , byrow =TRUE)

C
     [,1] [,2] [,3]
[1,]  1.0  6.0  7.0
[2,]  0.6  0.5  0.3
[3,]  1.0  2.0  1.0
```

7.3 Accessing Rows and Columns

Particular rows and columns of a data object (matrix as well as other objects such as data frames) can be accessed by specifying the row number or column number, leaving the other value blank.

```
A[1,]   # access first row of A
#[1] 1 6 5

C[,2]   # access second column of C
#[1] 6.0 0.5 2.0
```

Naturally - particular elements may be accessed by specifying the row number and column number

```
A[1,1]
# [1] 1

# This is not just for matrices.
# It is for all suitable data objects.

iris[,1]
iris[,1:3]
iris[2,]
iris[2,3]

mtcars[2:5,4:6]
```

7.3.1 Addition and subtractions

For matrices, addition and subtraction works on an element-wise basis. The first elements of the respective matrices are added, and so on.

```
A+C

#     [,1] [,2] [,3]
#[1,]  2.0 12.0 12.0
#[2,]  4.6  4.5  7.3
#[3,]  6.0  7.0 10.0

A-C

#     [,1] [,2] [,3]
#[1,]  0.0  0.0 -2.0
#[2,]  3.4  3.5  6.7
#[3,]  4.0  3.0  8.0
```

7.4 Matrix Multiplication

To multiply matrices, we require a special operator for matrices; see examples If we just used the normal multiplication, we would get an element-wise multiplication. This type of operation is very useful as a substitute for FOR loops on many occasions.

```
 A %*% C
#     [,1] [,2] [,3]
#[1,]  9.6 19.0 13.8
#[2,] 13.4 40.0 36.2
#[3,] 17.0 50.5 45.5

 A*C
#     [,1] [,2] [,3]
#[1,]  1.0   36 35.0
#[2,]  2.4    2  2.1
#[3,]  5.0   10  9.0
```

7.4.1 Basic Matrix Calculations

1) Inverting a matrix

   To invert a matrix we use the command solve() with no additional argument.

   Remember - Not all matrices are invertible. It the determinant of a matrix is zero, then no inverse exists.

```
    > solve(C)
                [,1]      [,2]       [,3]
    [1,] -0.03333333  2.666667 -0.5666667
    [2,] -0.10000000 -2.000000  1.3000000
    [3,]  0.23333333  1.333333 -1.0333333
```

   We can use this same command to solve a system of linear equations Ax=b. We would specify the vector b as the additional argument. (We will look at this matter more in the MATLAB component of the course).

2) Computing the determinant

   To compute the determinant, the command is simply det()

3) Determining the dimensions

   To find the dimensions of matrix A, we use the dim() command

4) Compute the transpose

```
det(C)
#[1] 3

dim(C) # number of rows and columns.

t(C)
     [,1] [,2] [,3]
[1,]    1  0.6    1
[2,]    6  0.5    2
[3,]    7  0.3    1
```

To compute the transpose of matrix A, we use the command t().

5) Cross Products and Kronecker Product

We can compute cross products using the `crossprod()` command. The Kronecker product (a very useful command in numerical computation) is also easily implementable using the `kronecker()` command.

```
crossprod(A,C)
kronecker(A,C)
kronecker(C,A)
```

7.4.2 Diagonals and the Identity Matrix

The `diag()` command is a very versatile function for using matrices.

It can be used to create a diagonal matrix with elements of a vector in the principal diagonal. For an existing matrix, it can be used to return a vector containing the elements of the principal diagonal.

Most importantly, if k is a scalar (i.e. single number such as 3) , `diag()` will create a $k \times k$ identity matrix.

```
Vec2=c(1,2,3)

diag(Vec2)     #      Constructs a diag. matrix based on Vec2

diag(A)        #     Returns diagonal elements of A as a vector

diag(3)        #     Creates a 3 x 3 identity matrix

diag(diag(A))  #     Creates the diagonal matrix D of matrix A
```

7.4.3 Linear Algebra Functions

R supports many import linear algebra functions such as cholesky decomposition, trace, rank, eigenvalues etc.

The required results may be determinable from the output of a command that pertains to an overall approach.

The eigenvalues and eigenvectors can be computed using the eigen() function. A data object known as a list is then created.

```
eigen(A)       #eigenvalues and eigenvectors

qr(A)          #returns Rank of a matrix

svd(A)
```

This is a very important type of matrix analysis, and many will encounter it again in future modules.

```
Y = eigen(A)
names(Y)

#   y$val are the eigenvalues of A
#   y$vec are the eigenvectors of A
```

7.4.4 More on Matrices

Note that the following commands are often useful.

- `rowMeans()`
- `rowSums()`
- `colMeans()`
- `colSums()`

7.5 Using rbind() and cbind()

Another methods of creating a matrix is to “bind a number of vectors together, either by row or by column. The commands are rbind() and cbind() respectively.

```
> x1 =c(1,2) ; x2 = c(3,8)

> D= rbind(x1,x2)

> E = cbind(x1,x2)

> det(D)

[1] 2

> det(E)

[1] 2
```

7.5.1 Solving a System of Linear Equations

To solve a system of linear equations in the form Ax=b , where A is a square matrix, and b is a column vector of known values, we use the solve() command to determine the values of the unknown vector x.

```
b=vec2  # from before

solve(A, b)
```

8 Lists

Many data objects returned as output are structured as lists, and some knowledge about them is quite important.

An R list is an object consisting of an ordered collection of objects known as its components. There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on.

Here is a simple example of how to make a list:

```
Lst <- list(name="Fred", wife="Mary", no.children=3,
                child.ages=c(4,7,9))
```

Components are always numbered and may always be referred to as such.

- Thus if Lst is the name of a list with 4 components, these may be individually referred to as Lst[[1]], Lst[[2]],etc.
- If Lst[[4]] is a vector, then Lst[[4]][1] is its first entry.

```
> Lst
$name
[1] "Fred"

$wife
[1] "Mary"

$no.children
[1] 3

$child.ages
[1] 4 7 9

> Lst[[1]]
[1] "Fred"
> Lst[[4]][1]
[1] 4
```

The function `length(Lst)` gives the number of (top level) components that the list has.

Components of lists may also be named, and in this case the component may be referred to either by giving the component name as a character string in place of the number in double square brackets, or, more conveniently, by giving an expression of the form

```
> name$component_name
```

for the same thing.

This is a very useful convention as it makes it easier to get the right component if you forget the number. This dollar sign operator is very useful , particularly when looking at the output of a complex statistical function. To find out the names assigned to a list use the command `names()`.

```
> names(Lst)
[1] "name"        "wife"        "no.children" "child.ages"
> Lst$name
[1] "Fred"
```

9 A Brief Introduction to fitting Linear Models (Lists)

A very commonly used statistical procedure is **simple linear regression**

- `lm()`
- `summary()`

```
Y <- c( )
X <- c( )

plot(X,Y)
cor(X,Y)
lm(Y~X)
```

```
FitA =lm(Y~X)
summary(FitA)
```

Let's look at this summary output in more detail, to see how it is structured. Importantly this object is structured as a list of named components.

```
names(summary(FitA))
class(summary(FitA))
mode(summary(FitA))
str(summary(FitA))
```

The summary of `FitA` is a data object in it's own right. We will save it under the name `Sum.FitA` (N.B. The dot in the name has no particular meaning).

```
Sum.FitA=summary(FitA)
Sum.FitA[1]
Sum.FitA$pvalue
```

Suppose we wish require the $p-$value for the slope estimate only.

```
class(Sum.FitA$pvalue)
mode(Sum.FitA$pvalue)
dim(Sum.FitA$pvalue)
```

10 The apply() family of functions

The "apply" family of functions keep you from having to write loops to perform some operation on every row or every column of a matrix or data frame, or on every element in a list.

10.1 The apply() function

The `apply()` function is a powerful device that operates on arrays and, in particular, matrices. The `apply()` function returns a vector (or array or list of values) obtained by applying a specified function to either the row or columns of an array or matrix. To specify use for rows or columns, use the additional argument of 1 for rows, and 2 for columns.

```
# create a matrix of 10 rows x 2 columns
m <- matrix(c(1:10, 11:20), nrow = 10, ncol = 2)

# mean of the rows

apply(m, 1, mean)
# [1]  6  7  8  9 10 11 12 13 14 15

# mean of the columns
apply(m, 2, mean)
#[1]  5.5 15.5
```

The local version of `apply()` is `lapply()`, which computes a function for each argument of a list, provided each argument is compatible with the function argument (e.g. that is numeric).

The `lapply()` command returns a list of the same length as a list `X`, each element of which is the result of applying a specified function to the corresponding element of X.

10.1.1 The sapply() command

A user friendly version of `lapply()` is `sapply()` .The `sapply()` command is a variant of `lapply()` , returning a matrix instead of a list - again of the same length as a list X, each element of which is the result of applying a specified function to the corresponding element of X.

```
> x <- list(a=1:10, b=exp(-3:3), logic=c(T,F,F,T))
>
> # compute the list mean for each list element
>
> lapply(x,mean)
$a
[1] 5.5

$b
[1] 4.535125

$logic
[1] 0.5
>
> sapply(x,mean)
       a        b    logic
5.500000 4.535125 0.500000
>
```

11 Functions

The function definition syntax of R is similar to that of other languages. For example:

```
func <- function(a, b) {
  return (a+b)
  }
```

The function `function()` returns a function, which is usually assigned to a variable, `func` in this case, but need not be. You may use the function statement to create an anonymous function (lambda expression).

Note that return is a function; its argument must be contained in parentheses (unlike C where parentheses are optional). The use of return is optional; otherwise the value of the last line executed in a function is its return value.

Default values can be defined . In the following example,a is set to 3 ad b is set to 10 by default.

```
f <- function(a=3, b=10)
   {
    return (a+b)
}
```

So `f(5, 1)` would return 6, and f(5) would return 15. `R` allows more sophisticated default values than does C++. C++ requires that if an argument has a default value then so do all values to the right. This is not the case in R, though it is still a good idea. The function definition

```
f <- function(a=10, b) { return (a+b)}
```

is legal, but calling f(5) would cause an error. The argument a would be assigned 5, but no value would be assigned to b. The reason such a function definition is not illegal is that one could still call the function with one named argument. For example, f(b=2) would return 12.

`sapply()`

```
> sapply(2:5,log)
[1] 0.6931472 1.0986123 1.3862944 1.6094379
>
> sapply(2:5,log,2)
[1] 1.000000 1.584963 2.000000 2.321928
```

12 Mathematical and Statistical Commands

12.1 Useful Statistical Commands

- `mean()` mean of a data set
- `median()` median of a data set
- `length()` Sample Size
- `IQR()` Inter-Quartile Range of a sample
- `var()` Variance of a sample
- `sd()` Standard Deviation of a sample
- `range()` Range of a data set
- `fivenum()` Tukey's five number summary

12.2 useful operators

- Factorials $n! = n \times n-1 \times \ldots \times 2 \times 1$
- Binomial Coefficients
$$\binom{n}{k} = \frac{n!}{(n-k)! \times k!}$$

The `R` commands are `factorial()` and `choose()` respectively.

12.3 Managing Precision

- `floor()` Floor function of x, $\lfloor x \rfloor$.
- `ceiling()` Ceiling function of x, $\lceil x \rceil$.
- `round()` Rounding a number to a specified number of decimal places.

12.4 The Birthday function

The R command pbirthday() computes the probability of a coincidence of a number of randomly chosen people sharing a birthday, given that there are n people to choose from. Suppose there are four people in a room. The probability of two of them sharing a birthday is computed as about 1.6 %

```
> pbirthday(4)
[1] 0.01635591
```

How many people do you need for a greater than 50% chance of a shared birthday? (choose from 23,43,63,83)?

12.5 Set Theory Operations

- `union()` union of sets A and B
- `intersect()` intersection of sets A and B
- `setdiff()` set difference A-B (order is important)

```
x = 5:10
y = 8:12
union(x,y)
intersect(x,y)
setdiff(x,y)
setdiff(y,x)
```