Vectors can be explicitly declared using the \texttt{c()} (concatenate) function.

\begin{verbatim}
> c(1, 2, 3)
[1] 1 2 3

> c("A", "B", "C")
[1] "A" "B" "C"

> x = c(1, 3)
> y = c(2, 4)
> c(x, y)
[1] 1 3 2 4
\end{verbatim}
Numeric vectors over a range can be created using the `:` operator

```r
> 1:10
[1]  1  2  3  4  5  6  7  8  9 10

> 12:5
[1] 12 11 10  9  8  7  6  5

> -3.2:5
[1] -3.2 -2.2 -1.2 -0.2  0.8  1.8  2.8  3.8  4.8
```
Be aware of order of operations (: takes precedence over -) or you can get unexpected results:

```r
> 1:length(x) - 1
[1] 0 1 2 3 4
```

```r
> 1:(length(x) - 1)
[1] 1 2 3 4
```
A more general way to create numerical sequences is the `seq` function which takes `from`, `to`, `by` as arguments.

```r
> seq(from = 1, to = 2, by = 0.2)
[1] 1.0 1.2 1.4 1.6 1.8 2.0

> seq(1, 4, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0

> seq(-3, 2)
[1] -3 -2 -1 0 1 2
```
seq function can be called with the length.out argument instead of by

> seq(from = 0, to = 5, length.out = 11)
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

> seq(0, 10, length.out = 5)
[1] 0.0 2.5 5.0 7.5 10.0

> seq(0, 10, len = 5)
[1] 0.0 2.5 5.0 7.5 10.0

> seq(0, 10, 5)
[1] 0 5 10
In some cases it is easier to create the desired sequence by repeating shorter sequences. This can be done with the `rep` function.

```r
> rep(TRUE, 3)
[1] TRUE TRUE TRUE

> rep(c(1, 2, 3), 2)
[1] 1 2 3 1 2 3

> rep(c(4, 5), c(3, 3))
[1] 4 4 4 5 5 5
```
Part II

Subsetting
In R (like many other languages) we use brackets [] to subset.

R has 5 ways to select specific elements from (most) objects.

1. Indexing by position
2. Indexing by exclusion
3. Indexing by name
4. Indexing by logical mask
5. Empty subsetting
Subsetting by position works by selecting elements by their numerical index.

Note that R starts its indexes from 1 and not 0 like some other languages.

```r
> x = c(1, 1, 2, 3, 5, 8)
> x[1]
[1] 1

> x[6]
[1] 8

> x[c(2, 3)]
[1] 1 2

> x[1:6]
[1] 1 1 2 3 5 8
```
Subsetting by exclusion works by excluding elements by their numerical index. Indexes to exclude are indicated by a negative value.

```r
> x = c(1, 1, 2, 3, 5, 8)
> x[-1]
[1] 1 2 3 5 8

> x[-6]
[1] 1 1 2 3 5

> x[-c(2, 3)]
[1] 1 3 5 8

> x[-(1:6)]
numeric(0)
```
Subsetting by name selects elements by matching their names (if they exist, more on this later).

```r
> x = c(a = 1, b = 2, c = 3)
> x["a"]
a
1

> x["b"]
b
2

> x[c("b", "c")]
b c
2 3
```
Subsetting by logical mask works by selecting elements using a logical vector of the same length where TRUE indexes are included and FALSE indexes excluded.

```r
> x = c(1, 1, 2, 3, 5, 8)
> x[c(TRUE, TRUE, FALSE, FALSE, TRUE, TRUE)]
[1] 1 1 5 8

> x[c(TRUE, FALSE)]
[1] 1 2 5

> x == 1
[1] TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> x[x == 1]
[1] 1 1

> x[x%%2 == 0]
[1] 2 8
```
Empty subsetting selects all elements. This is useful when using subsetting for assignment.

> x = c(1, 2, 3)
> y = c(1, 2, 3)
> x[]
[1] 1 2 3

> x = 3
> y[] = 3
> x
[1] 3

> y
[1] 3 3 3
It is possible to use subsetting with assignment to change values of only specific elements.

```r
> x = c(1, 1, 2, 3, 5, 8)
> x[1] = 8
> x
[1] 8 1 2 3 5 8

> x[1:4] = c(-1, -2)
> x
[1] -1 -2 -1 -2 5 8

> x[x%%2 != 0] = 0
> x
[1] 0 -2 0 -2 0 8
```
Part III

Data Classes
Every R object has a number of attributes. Some of the most important are:

- **mode**: Mutually exclusive classification of objects according to their basic structure.
  - logical, integer, double, complex, raw, character, list, expression, function, NULL, ...

- **class**: Property assigned to an object that determines how generic functions operate with it. If no specific class is assigned to object, by default it is the same as the mode.
  - vector, matrix, array, data.frame, list, factor, ...
<table>
<thead>
<tr>
<th>Mode / Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>boolean value that is either TRUE or FALSE</td>
</tr>
<tr>
<td>numeric</td>
<td>numerical value can either be an integer or floating point (double)</td>
</tr>
<tr>
<td>complex</td>
<td>complex numerical value</td>
</tr>
<tr>
<td>character</td>
<td>character string</td>
</tr>
<tr>
<td>function</td>
<td>collection of arbitrary R expressions</td>
</tr>
<tr>
<td>Class</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>vector</td>
<td>1-d array of elements of the same class</td>
</tr>
<tr>
<td>matrix</td>
<td>2-d array of elements of the same class</td>
</tr>
<tr>
<td>array</td>
<td>n-d array of elements of the same class</td>
</tr>
<tr>
<td>data.frame</td>
<td>2-d array of elements where elements in the same column have the same class</td>
</tr>
<tr>
<td>list</td>
<td>Generic vector where elements can be of any class</td>
</tr>
<tr>
<td>factor</td>
<td>Categorical variable with defined levels</td>
</tr>
</tbody>
</table>
R allows for automatic type coercion. This is usually helpful but can also cause problems.

```r
> x = c(1, 2, 3, 8)
> y = rep(TRUE, 3)
> c(x, y)
[1] 1 2 3 8 1 1 1

> c(x, "a")
[1] "1" "2" "3" "8" "a"

> c(y, "a")
[1] "TRUE" "TRUE" "TRUE" "a"
```
Problematic coercion:

```r
> z = factor(c("A", "A", "B", "A"))
> z
[1] A A B A
Levels: A B

> c(z, 1)
[1] 1 1 2 1 1

> c(z, FALSE)
[1] 1 1 2 1 0

> c(z, "A")
[1] "1" "1" "2" "1" "A"
```
Explicit coercion is also possible within R usually using "as" functions:

```r
> as.numeric("151")
[1] 151

> as.complex("1")
[1] 1+0i

> as.factor(c("h", 1, "3"))
[1] h 1 3
Levels: 1 3 h

> as.character(c(1, 2, 3))
[1] "1" "2" "3"
```
Lengths of objects are also subject to implicit coercion (this is some times known as recycling).

For many operations on two objects if one object is shorter than the other, then the elements of the shorter object are repeated to produce an object of the same length as the longer one.

```r
> 1:4 + 1
[1] 2 3 4 5

> 1:4 + c(2, 3)
[1] 3 5 5 7

> 1:4 + 1:3
[1] 2 4 6 5

> 1:4 + c(1, 1, 1, 1)
[1] 2 3 4 5

> 1:4 + c(2, 3, 2, 3)
[1] 3 5 5 7

> 1:4 + c(1, 2, 3, 1)
[1] 2 4 6 5
```
### Special Objects / Classes:

<table>
<thead>
<tr>
<th>Function</th>
<th>Test Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>is.na</td>
<td>Represents missing data</td>
</tr>
<tr>
<td>NULL</td>
<td>is.null</td>
<td>Represents the null / empty object</td>
</tr>
<tr>
<td>NaN</td>
<td>is.nan</td>
<td>Represents a numerical value that is not a number</td>
</tr>
<tr>
<td>Inf</td>
<td>is.infinite</td>
<td>Represents an infinite value</td>
</tr>
</tbody>
</table>

```r
> x = NA
> x == 1
[1] NA

> x = c()
> x == 1
logical(0)

> x == NA
[1] NA

> x == NULL
logical(0)

> is.na(x)
[1] TRUE

> is.null(x)
[1] TRUE
```
Character objects are immutable strings.

Unlike other languages there are no direct ways of accessing the characters that make up the string.

```r
> length("text")
[1] 1

> nchar("text")
[1] 4
```
To create or modify character vectors the most useful function is `paste` which concatenates string arguments

```r
> paste("X", "Y")
[1] "X Y"

> paste("X", "Y", sep = " + ")
[1] "X + Y"

> paste("Fig", 1:4)
[1] "Fig 1" "Fig 2" "Fig 3" "Fig 4"

> paste(c("X", "Y"), 1:4, sep = "", collapse = " + ")
[1] "X1 + Y2 + X3 + Y4"
```
### Other Common Character Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>substr</td>
<td>Extracts or replaces a substring</td>
</tr>
<tr>
<td>strsplit</td>
<td>Splits string at specific patterns</td>
</tr>
<tr>
<td>toupper</td>
<td>All characters in string to upper case</td>
</tr>
<tr>
<td>tolower</td>
<td>All characters in string to lower case</td>
</tr>
<tr>
<td>grep</td>
<td>Regex string matching</td>
</tr>
<tr>
<td>gsub</td>
<td>Regex string replacement</td>
</tr>
</tbody>
</table>
Matrices are an extension of a vector into 2 dimensions. There are several approaches available to construct a matrix.

```r
> matrix(1:9, ncol = 3, nrow = 3)

[,1] [,2] [,3]
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9

> matrix(1:9, ncol = 3, nrow = 3, byrow = TRUE)

[,1] [,2] [,3]
[1,]  1  2  3
[2,]  4  5  6
[3,]  7  8  9
```
Matrices may be of any mode, but all elements must have the same mode.

```r
> matrix(c(TRUE, FALSE, TRUE), ncol = 3, nrow = 3)
  [,1] [,2] [,3]
[1,]  TRUE TRUE TRUE
[2,] FALSE FALSE FALSE
[3,]  TRUE TRUE TRUE

> matrix(rep(c("AB", "MN", "YZ"), 3), ncol = 3, nrow = 3)
  [,1] [,2] [,3]
[1,] "AB" "AB" "AB"
[2,] "MN" "MN" "MN"
[3,] "YZ" "YZ" "YZ"
```
Matrices can also be built by adding on a vector/matrix on to an existing vector/matrix by row or column.

```r
> rbind(1:3, 4:6, 7:9)
    [,1] [,2] [,3]
[1,]   1   2   3
[2,]   4   5   6
[3,]   7   8   9

> cbind(1:3, 4:6, 7:9)
    [,1] [,2] [,3]
[1,]   1   4   7
[2,]   2   5   8
[3,]   3   6   9
```
What would $m$ look like if the following code was run?

```r
> m = matrix(c(1, 2), 3, 2, byrow = TRUE)
```

```
[,1] [,2]
[1,] 1 2
[2,] 1 2
[3,] 1 2
```
What would \( m \) look like if the following code was run?

```r
> m = matrix(c(1, 2), 3, 2, byrow = TRUE)
> m

[,1] [,2]
[1,] 1  2
[2,] 1  2
[3,] 1  2
```
What about \( m2 \) if the following code was run?

\[
> m2 = cbind(rbind(m, 1:2), 3)
\]
What about \( m2 \) if the following code was run?

\[
> m2 = cbind(rbind(m, 1:2), 3)
\]

\[
> m2
\]

\[
[,1]  [,2]  [,3]  
[1,]  1  2  3  
[2,]  1  2  3  
[3,]  1  2  3  
[4,]  1  2  3  
\]
Subsetting of matrices works the same way as vectors, except there are now 2 dimensions to subset on. Note that if the subsetting result has only 1 dimension then a vector is returned.

```r
> x = matrix(1:9, 3, 3)
> x[1, ]
[1] 1 4 7
> x[, 1]
[1] 1 2 3
> x[1, 1]
[1] 1
> x[c(1, 2), ]
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
> x[1:2, 2:3]
[,1] [,2]
[1,] 4 7
[2,] 5 8
```
### Common Matrix Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>dimensions of the matrix</td>
</tr>
<tr>
<td>ncol</td>
<td>number of columns</td>
</tr>
<tr>
<td>nrow</td>
<td>number of rows</td>
</tr>
<tr>
<td>colSums</td>
<td>calculates the sum of columns</td>
</tr>
<tr>
<td>rowSums</td>
<td>calculates the sum of rows</td>
</tr>
<tr>
<td>t</td>
<td>transpose of the matrix</td>
</tr>
<tr>
<td>%*%</td>
<td>matrix multiplication operator</td>
</tr>
<tr>
<td>solve</td>
<td>calculates inverse of the matrix</td>
</tr>
</tbody>
</table>
Be careful when using a function that does not explicitly take a matrix as an argument as the results can be unpredictable.

```
> m2
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 1 2 3
[3,] 1 2 3
[4,] 1 2 3

> sum(m2)
[1] 24

> mean(m2)
[1] 2

> sd(m2)
[1] 0 0 0
```
Some functions have assignment methods which can be used to manipulate the passed arguments. For example the `dim` function can be used to alter the dimensions of a matrix.

```r
> m2
 [,1] [,2] [,3]
[1,] 1 2 3
[2,] 1 2 3
[3,] 1 2 3
[4,] 1 2 3

> dim(m2) = c(2, 6)
> m2
[1,] 1 1 2 2 3 3
[2,] 1 1 2 2 3 3
```
Arrays are the same as matrices except that they can have an arbitrary number of dimensions.

```r
> (a = array(1:4, c(1, 4, 3)))
[1,]  1  2  3  4

> a[, , 3]
[1] 1 2 3 4

> a[, 1:2, ]
[1,]  1  2  3  4

> a[1, , ]
[1,]  1  2  3  4
[2,]  2  2  2  2
[3,]  3  3  3  3
[4,]  4  4  4  4
```
In the second and third examples from the last slide, R is dropping the third dimension from the array this behavior can be suppressed by the `drop` argument.

```
> a[, 1:2, ]
 [,1] [,2] [,3]
[1,]  1  1  1
[2,]  2  2  2

> a[, 1:2, , drop = FALSE]
 , , 1
 [,1] [,2]
[1,]  1  2
 , , 2
 [,1] [,2]
[1,]  1  2
 , , 3
 [,1] [,2]
[1,]  1  2
```

Colin Rundel  crundel@stat.ucla.edu
Intermediate R  UCLA SCC
A data frame is a 2-d array of data where only the columns are constrained to be of the same type. This is the default data class used when reading in data from a file.

```r
> scores = read.csv("scores.csv")
> scores
    Name id Quiz1 Exam1 Exam2 Quiz2 Exam3
 1  Susan 123412  50   47   33   67   79
 2    John 548963  38   61   75   59   65
 3    Bob 234563  89   97   85   88   92
 4    Bill 429591  72   73   74   75   76
 5    Mary 245887  92   95   79   89   90
 6    Paul  97522  99    3   55   60   72
```
With this data set there was a header row, R uses the elements in this row to name the columns in the data frame. We can use these names to access the columns using $ or $[[]].

```r
> scores$Name  
[1] Susan John Bob Bill Mary Paul  
Levels: Bill Bob John Mary Paul Susan

> scores$Quiz2  
[1] 67 59 88 75 89 60

> scores[["Exam1"]]
[1] 47 61 97 73 95 3

> x = "id"
> scores[[x]]
[1] 123412 548963 234563 429591 245887 97522
```
We can also use traditional indexing to access elements, rows, and columns in a data frame.

```r
> scores[, 1]
[1] Susan John Bob Bill Mary Paul
Levels: Bill Bob John Mary Paul Susan

> scores[2, ]
 Name   id Quiz1 Exam1 Exam2 Quiz2 Exam3
2 John 548963  38   61   75   59   65

> scores[3, 3:7]
 Quiz1 Exam1 Exam2 Quiz2 Exam3
3   89   97   85   88   92
```
If you wanted a list of names and ids of all students who scored under 50 on one of the three exams?

```r
> scores[, c(4, 5, 7)] < 50
Exam1 Exam2 Exam3
[1,] TRUE TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,] TRUE FALSE FALSE
> as.logical(rowSums(scores[, c(4, 5, 7)] < 50))
[1] TRUE FALSE FALSE FALSE FALSE TRUE
> scores[as.logical(rowSums(scores[, c(4, 5, 7)] < 50)), c(1, 2)]
Name id
1 Susan 123412
6 Paul 97522
```
If you wanted a list of names and ids of all students who scored under 50 on one of the three exams?

```r
> scores[, c(4, 5, 7)] < 50
  Exam1 Exam2 Exam3
[1,] TRUE TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,] TRUE FALSE FALSE
```

> as.logical(rowSums(scores[, c(4, 5, 7)] < 50))

```r
[1] TRUE FALSE FALSE FALSE FALSE TRUE
```

> scores[as.logical(rowSums(scores[, c(4, 5, 7)] < 50)), c(1, 2)]

<table>
<thead>
<tr>
<th>Name</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susan</td>
<td>123412</td>
</tr>
<tr>
<td>Paul</td>
<td>97522</td>
</tr>
</tbody>
</table>
If you wanted a list of names and ids of all students who scored under 50 on one of the three exams?

```r
> scores[, c(4, 5, 7)] < 50
     Exam1 Exam2 Exam3
[1,]  TRUE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,]  TRUE FALSE FALSE

> as.logical(rowSums(scores[, c(4, 5, 7)] < 50))
[1]  TRUE FALSE FALSE FALSE FALSE  TRUE
```
If you wanted a list of names and ids of all students who scored under 50 on one of the three exams?

```r
> scores[, c(4, 5, 7)] < 50
   Exam1 Exam2 Exam3
[1,]  TRUE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,]  TRUE FALSE FALSE

> as.logical(rowSums(scores[, c(4, 5, 7)] < 50))
[1]  TRUE FALSE FALSE FALSE FALSE  TRUE

> scores[as.logical(rowSums(scores[, c(4, 5, 7)] < 50)), c(1, 2)]
  Name  id
1 Susan 123412
6  Paul  97522
```
Data frames can be created/modified in much the same way as matrices.

```r
(d = data.frame(a = c(1, 2, 3), b = c("m", "n", "o"), c = TRUE))
a b c
1 1 m TRUE
2 2 n TRUE
3 3 o TRUE

d$d = factor(c("a", "b", "c"))

d[, 5] = as.complex(1)

cbind(d, f = 1)
a b c d V5 f
1 1 m TRUE a 1+0i 1
2 2 n TRUE b 1+0i 1
3 3 o TRUE c 1+0i 1
```
Lists are a generic vector that allows for the collection of arbitrary objects / classes.

```r
> (l = list(a = c(TRUE, FALSE), b = matrix(1:4, 2, 2), "hello"))
$a
[1] TRUE FALSE

$b
[,1] [,2]
[1,] 1 3
[2,] 2 4

[[3]]
[1] "hello"
```
Elements of lists can be accessed using $ and or $[[ ]]$.

```r
> l$a
[1] TRUE FALSE

> l["b"]
[,1] [,2]
[1,] 1  3
[2,] 2  4

> l[[3]]
[1] "hello"
```
[] can also be used to select one or more elements, but the returned object will be a list.

```r
> l[3]
[[1]]
[1] "hello"

> class(l[3])
[1] "list"

> l[c(1, 2)]
$a
[1] TRUE FALSE

> class(l[c(1, 2)])
[1] "list"

> l["a"]
$a
[1] TRUE FALSE

> class(l["a"])
[1] "list"
```
Names of elements can also be altered after the fact using the `names` replacement function.

```r
> names(l)  
[1] "a" "b" ""  

> names(l)[3] = "c"  
> l["c"]  
[1] "hello"  

> names(l) = c("x", "y", "z")  
> names(l)  
[1] "x" "y" "z"  

> l["x"]  
[1] TRUE FALSE
```
The *names* function can also be used with any of the other data classes we have discussed so far.

```r
> a = c(1, 2, 3, 4)
> names(a) = c("w", "x", "y", "z")
> a
w x y z
1 2 3 4

> a["x"]

x
2

> a[c("w", "y")]
w y
1 3
```
In the case of matrices you can label columns and rows via the `colnames` and `rownames` functions.

```r
> b = matrix(1:4, 2, 2)
> colnames(b) = c("x", "y")
> rownames(b) = c("m", "n")
> b
     x  y
   m 1 3
   n 2 4

> b[, "x"]
   m n
   1 2

> b["n", ]
   x y
   2 4
```
Additional uses for names

Remember the data frame we used earlier?

```r
> scores
   Name id Quiz1 Exam1 Exam2 Quiz2 Exam3
1  Susan 123412   50    47    33    67    79
2   John  548963   38    61    75    59    65
3   Bob  234563   89    97    85    88    92
4   Bill  429591   72    73    74    75    76
5   Mary  245887   92    95    79    89    90
6  Paul  97522     99     3    55    60    72
```
With objects that have a name attribute you can use the *attach* or the *with* command.

```r
> attach(scores)
> mean(Exam1 + Exam2 + Exam3)
[1] 208.5

> Name
[1] Susan John Bob Bill Mary Paul
Levels: Bill Bob John Mary Paul Susan

> detach(scores)
> with(scores, mean(Exam1 + Exam2 + Exam3))
[1] 208.5
```

Note - *attach* should never be used as it can result namespace collisions.
Part IV

Defining Functions
Functions in R are defined using the *function* keyword. Curly braces are used to define the body of the function.

The value / object returned by the function is indicated by the *return* keyword.

```r
> square = function(x) {
+    return(x^2)
+ }
> square(5)
[1] 25
> square(1:3)
[1] 1 4 9
```
If the function’s code is only one line then the braces are not needed (this is true for anywhere curly braces are used)

`return` calls are also implicit if not present, the output of the last expression in the function is returned.

```r
> cube = function(x) x^3
> cube(4)
[1] 64

> cube(1:3)
[1] 1 8 27
```
Arguments for functions can be given default values using =

When calling a function arguments can be explicitly referenced by name otherwise ordering is used.

```r
> pow = function(x, y = 2) x^y
> pow(2)
[1] 4

> pow(2, 4)
[1] 16

> pow(y = 4, 2)
[1] 16

> pow(y = 3, x = 3)
[1] 27
```
In some cases it is desirable to not explicitly define the arguments a function takes, for example the `sum` function. This can be accomplished by using `...` in the function definition.

It is also possible to return multiple values by combining them in a list object.

```r
> mini.summary = function(...) {
+   n = c(...)
+   return(list(mean = mean(n), median = median(n)))
+ }
> mini.summary(1, 2, 2, 3)

$mean
[1] 2

$median
[1] 2
```
... can also be used with explicitly named arguments.

```r
> test = function(x, y, ...) {
+   dots = paste(c(...), collapse = " ")
+   print(paste("x="", x, " y="", y, " ...="", dots, sep = " "))
+ }
> test(1, 2, 3, 4, 5, 6, 7)
[1] "x=1 y=2 ...=3 4 5 6 7"
> test(1, 2, 3, 4)
[1] "x=1 y=2 ...=3 4"
> test(1, 2, 3)
[1] "x=1 y=2 ...=3"
```
Arguments contained in ... can also be named.

```r
> test2 = function(x, ...) {
+   l = list(...)
+   n = names(l)
+   for (i in n[n != ""]){
+     cat(paste(i, "=" , l[i]), "\n")
+   }
+ }

> test2(1, 2, z = 3, y = 4)
z = 3
y = 4

> test2(1, 2, mean = 3)
mean = 3
```
Changes made to variables within a given scope (function) are lost when the scope finishes.

It is possible to modify variables outside of the local scope by using the global assignment operator `<<-`.

```r
> x = 3
> (function() x = 2)()
> x
[1] 3

> (function() x <<- 2)()
> x
[1] 2

> (function() x)()
[1] 2
```
General Advice:

- Write your own functions, (re)use them everywhere
- Writing functions will make you a better programmer
- If you find yourself copy and pasting blocks of code, write a function instead
- Read other people’s functions/code (R makes this tremendously easy)
- Do not reinvent the wheel, particularly if that wheel is in base
Part V

Flow Control and Loops
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td>&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>logical equals</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>xor(x,y)</td>
<td>exclusive OR</td>
</tr>
<tr>
<td>isTRUE(x)</td>
<td>equivalent to x==TRUE</td>
</tr>
</tbody>
</table>
Basic flow control in R is accomplished with *if* *else* commands.

```r
> even.odd = function(x) {
+   if (!is.numeric(x)) {
+     print("neither")
+   }
+   else if (x%%2 == 0) {
+     print("even")
+   }
+   else {
+     print("odd")
+   }
+ }

> even.odd(3) [1] "odd"
> even.odd(4) [1] "even"
> even.odd("A") [1] "neither"
> even.odd(NA) [1] "neither"
> even.odd(c(3, 4)) [1] "odd"
```
*if* does not work with vectors, only the first element is used. To test logical vectors the *any* and *all* functions can be used.

```r
> any(3 == 1:5)
[1] TRUE

> any(0 == 1:5)
[1] FALSE

> all(1:5 == 1:5)
[1] TRUE

> all(1 == 1:2)
[1] FALSE
```
ifelse is a function with similar use, it returns a specified value if the test is TRUE or a second specified value if it is FALSE. (This function handles vector arguments properly)

```r
> ifelse(3%%2 == 0, "even", "odd")
[1] "odd"

> ifelse(4%%2 == 0, "even", "odd")
[1] "even"

> ifelse(c(3, 4)%%2 == 0, "even", "odd")
[1] "odd"    "even"
```
There are three main types of loops in R, *for*, *while* and *repeat*. *for* loops are defined by a variable that iterates over a set of elements.

```r
> for (x in 1:3) {
+   print(x)
+ }
[1] 1
[1] 2
[1] 3

> for (x in c("hello", "goodbye")) {
+   print(x)
+ }
[1] "hello"
[1] "goodbye"
```
for loops will iterate over just about any basic object. (Not that I suggest you do this)

```r
> m = matrix(1:4, nrow = 2, ncol = 2)
> for (x in m) print(x)
[1] 1
[1] 2
[1] 3
[1] 4

> d = data.frame(a = c(1, 2), b = "A")
> for (x in d) print(x)
[1] 1 2
[1] A A
Levels: A

> l = list(a = c(1, 2), b = c("A"))
> for (x in d) print(x)
[1] 1 2
[1] A A
Levels: A
```
Behavior within loops can be modified using the *next* and *break* commands.

Within a *for* loop:

- *next* moves the iterator to the next element and continues at the start of the loop
- *break* immediately exits the loop

```r
> for (x in 1:9) {
+   if (x%%2 == 0)
+     next
+   if (x == 7)
+     break
+   print(x)
+ }
[1] 1
[1] 3
[1] 5
```
A `while` loop repeats until the given condition becomes false.

- `next` forces the loop back to the start
- `break` immediately exits the loop

```r
> x = 1
> while (x < 3) {
+     print(x)
+     x = x + 1
+ }
[1] 1
[1] 2
```
A `repeat` loop is equivalent to a `while` loop where the condition is always true, the only way to exit is by using break.

- `next` forces the loop back to the start.

```r
> x = 1
> repeat {
+   print(x)
+   x = x + 1
+   if (x > 3)
+     break
+ }
[1] 1
[1] 2
[1] 3
```

Why is this output different from the `while` loop on the last slide?
In R there is a family of *apply* functions that applies a given function to each element of a vector, matrix, list, etc. These functions are usually much faster than equivalent implementations using loops.

- **Usage:** `apply(X, MARGIN, FUN, ...)`

```r
> (x = matrix(1:9, 3, byrow = T))
   [,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
[3,] 7 8 9

> apply(x, 1, mean)
[1] 2 5 8

> apply(x, 2, function(x) sum(x)/length(x))
[1] 4 5 6
```
*lapply* and *sapply* are similar functions that work with vectors, lists, and data frames.

Both are functions are nearly identical but *sapply* simplifies the results when possible.

```r
> sapply(scores[, 3:7], mean)
  Quiz1  Exam1  Exam2  Quiz2  Exam3
    73.3333 62.6667 66.8333  73.0000  79.0000
> sapply(scores[, 3:7], sd)
  Quiz1  Exam1  Exam2  Quiz2  Exam3
```
Conventional wisdom and Advice

- Looping in R is slow
- Speed mostly depends on what is occurring inside the loop(s) and the size of your data
- R has power functional programming features, use them when you can
- Most important factor - working/running code
- Premature optimization is the root of all evil
- In general: vectorization is faster than apply is faster than a loop
Part VI

Packages
Packages consist in a set of pre-programmed functions, sometimes developed for performing specific tasks.

There are two types of packages:

- Those that come with the base installation of R.
- Those that are available for download and need to be installed manually.
To check which packages are installed and to install new ones use the Package Installer
<table>
<thead>
<tr>
<th>Package</th>
<th>Installed Version</th>
<th>Repository Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td></td>
<td>1.1</td>
</tr>
<tr>
<td>abind</td>
<td>1.1-0</td>
<td>1.1-0</td>
</tr>
<tr>
<td>AcceptanceSampling</td>
<td></td>
<td>1.0-1</td>
</tr>
<tr>
<td>ACCLMA</td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>accuracy</td>
<td>1.35</td>
<td></td>
</tr>
<tr>
<td>acepack</td>
<td>1.3-3.0</td>
<td></td>
</tr>
<tr>
<td>aCGH.Spline</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>actuar</td>
<td>1.1-1</td>
<td></td>
</tr>
<tr>
<td>ada</td>
<td>2.0-2</td>
<td></td>
</tr>
<tr>
<td>adabag</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>AdaptFit</td>
<td>0.2-2</td>
<td></td>
</tr>
<tr>
<td>adaptTest</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>ade4</td>
<td>1.4-16</td>
<td>1.4-16</td>
</tr>
<tr>
<td>ade4TkGUI</td>
<td>0.2-5</td>
<td></td>
</tr>
<tr>
<td>adegenet</td>
<td>1.2-5</td>
<td>1.2-6</td>
</tr>
</tbody>
</table>
Once a package is installed it can be loaded in the R environment using the `library` or `require` functions.

To see the names of functions available in a given package use `library` with the `help` argument.

```
> library(ggplot2)
> library(rgdal)
> library(help = splines)
```
Information on package 'splines'

Description:

Package: splines
Version: 2.11.1
Priority: base
Imports: graphics, stats
Title: Regression Spline Functions and Classes
Author: Douglas M. Bates <bates@stat.wisc.edu> and William N. Venables <Bill.Venables@csiro.au>
Maintainer: R Core Team <R-core@r-project.org>
Description: Regression spline functions and classes
License: Part of R 2.11.1
Built: R 2.11.1; universal-apple-darwin9.8.0; 2010-05-31 14:43:06 UTC; unix

Index:

asVector Coerce an Object to a Vector
backSpline Monotone Inverse Spline
bs B-Spline Basis for Polynomial Splines
interpSpline Create an Interpolation Spline
ns Generate a Basis Matrix for Natural Cubic Splines
periodicSpline Create a Periodic Interpolation Spline
polySpline Piecewise Polynomial Spline Representation
predict.bspline Evaluate a Spline at New Values of x
predict.bs Evaluate a Spline Basis
sp: classes and methods for spatial data

A package that provides classes and methods for spatial data. The classes document where the spatial location information resides, for 2D or 3D data. Utility functions are provided, e.g. for plotting data as maps, spatial selection, as well as methods for retrieving coordinates, for subsetting, print, summary, etc.

Version: 0.9-71
Depends: R (≥ 2.10.0), methods
Imports: utils, lattice, grid
Suggests: lattice, RColorBrewer, rgdal
Published: 2010-10-11
Author: Edzer Pebesma, Roger Bivand and others
Maintainer: Edzer Pebesma <edzer.pebesma at uni-muenster.de>
License: GPL (≥ 2)
URL: http://r-spatial.sourceforge.net/
Citation: sp citation info
In views: Spatial
CRAN checks: sp results

Downloads:

Package source: sp_0.9-71.tar.gz
MacOS X binary: sp_0.9-70.tgz
Windows binary: sp_0.9-71.zip
Reference manual: sp.pdf
Vignettes: sp: classes and methods for spatial data
CRAN Task Views

- Bayesian
  Bayesian Inference

- ChemPhys
  Chemometrics and Computational Physics

- ClinicalTrials
  Clinical Trial Design, Monitoring, and Analysis

- Cluster
  Cluster Analysis & Finite Mixture Models

- Distributions
  Probability Distributions

- Econometrics
  Computational Econometrics

- Environmetrics
  Analysis of Ecological and Environmental Data

- ExperimentalDesign
  Design of Experiments (DoE) & Analysis of Experimental Data

- Finance
  Empirical Finance

- Genetics
  Statistical Genetics

- Graphics
  Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization

- gR
  gRaphical Models in R

- HighPerformanceComputing
  High-Performance and Parallel Computing with R

- MachineLearning
  Machine Learning & Statistical Learning

- MedicalImaging
  Medical Image Analysis

- Multivariate
  Multivariate Statistics

- NaturalLanguageProcessing
  Natural Language Processing

- OfficialStatistics
  Official Statistics & Survey Methodology

- Optimization
  Optimization and Mathematical Programming

- Pharmacokinetics
  Analysis of Pharmacokinetic Data

- Phylogenetics
  Phylogenetics, Especially Comparative Methods

- Psychometrics
  Psychometric Models and Methods
Part VII

Additional Resources
R-Seek Search Engine

http://www.rseek.org
Stackoverflow

http://stackoverflow.com/questions/tagged/r

Extract Links from Webpage using R

The two posts below are great examples of different approaches of extracting data from websites and parsing it into R. Scraping html tables into R data frames using the XML package. How can I use R ...

>Linux, delivered_
Twitter

http://search.twitter.com/search?q=#rstats
R-bloggers

http://www.r-bloggers.com/

R Tutorial Series: Labeling Data Points on a Plot

POSTED ON R TUTORIAL SERIES
RETRIEVED ON SEPTEMBER 19, 2010
Inside R

http://www.inside-r.org/
UCLA Statistics Information Portal

http://info.stat.ucla.edu/grad/

Colin Rundel  crundel@stat.ucla.edu

Intermediate R
E-consulting and Walk-in Consulting

http://scc.stat.ucla.edu

<table>
<thead>
<tr>
<th>Subject</th>
<th>Views</th>
<th>Written By</th>
<th>Posted</th>
</tr>
</thead>
<tbody>
<tr>
<td>weights case</td>
<td>161</td>
<td>joscani</td>
<td>03/09/2009 09:01AM</td>
</tr>
<tr>
<td>Re: weights case</td>
<td>75</td>
<td>David Diaz</td>
<td>03/10/2009 08:38AM</td>
</tr>
<tr>
<td>Re: weights case</td>
<td>96</td>
<td>joscani</td>
<td>03/10/2009 11:41AM</td>
</tr>
<tr>
<td>Re: weights case</td>
<td>66</td>
<td>Jan de Leeuw</td>
<td>03/11/2009 06:07PM</td>
</tr>
<tr>
<td>Adding matrices in an array</td>
<td>190</td>
<td>Mine Cetinkaya</td>
<td>03/02/2009 03:06PM</td>
</tr>
<tr>
<td>Re: Adding matrices in an array</td>
<td>85</td>
<td>David Diaz</td>
<td>03/02/2009 06:32PM</td>
</tr>
<tr>
<td>Re: Adding matrices in an array</td>
<td>54</td>
<td>Mine Cetinkaya</td>
<td>03/03/2009 07:49AM</td>
</tr>
<tr>
<td>Re: Adding matrices in an array</td>
<td>53</td>
<td>David Diaz</td>
<td>03/03/2009 07:58AM</td>
</tr>
<tr>
<td>Re: Adding matrices in an array</td>
<td>55</td>
<td>Mine Cetinkaya</td>
<td>03/03/2009 09:11AM</td>
</tr>
</tbody>
</table>
# SCC Mini-Courses

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Presenter</th>
<th>Materials</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Week 2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>October 5</td>
<td>R Programming &amp; Graphics I: Introduction to Programming and Graphics</td>
<td>Mark Nakamura</td>
<td></td>
</tr>
<tr>
<td><strong>Week 3</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>October 12</td>
<td>R Programming II: Data Manipulation and Functions</td>
<td>Colin Rundel</td>
<td></td>
</tr>
<tr>
<td><strong>Week 4</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>October 19</td>
<td>LaTeX I: Writing a Document, Paper or Thesis, Bibliographies, Style and Math</td>
<td>Denise Tsai</td>
<td></td>
</tr>
<tr>
<td><strong>Week 5</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>October 26</td>
<td>LaTeX II: Academic Talks and Presentations</td>
<td>Mine Cetinkaya</td>
<td></td>
</tr>
<tr>
<td><strong>Week 6</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>November 2</td>
<td>R Graphics II: Graphics for Exploratory Data Analysis</td>
<td>Irina Kukuyeva</td>
<td></td>
</tr>
<tr>
<td><strong>Week 7</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>November 9</td>
<td>R Stats I: Introductory Statistics</td>
<td>Kekona Kai Sorenson</td>
<td></td>
</tr>
<tr>
<td><strong>Week 8</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>November 16</td>
<td>R Stats II: Linear Regression</td>
<td>Masanao Yajima</td>
<td></td>
</tr>
<tr>
<td><strong>Week 9</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>November 23</td>
<td>R Stats III: Generalized Linear Models</td>
<td>Tiffany Himmel</td>
<td></td>
</tr>
</tbody>
</table>