(terse) Introduction to R

Starting an R session

Most of you use windoze, so you can start R by clicking on the desktop icon at Grice or by using the windows menu system to start R. On other systems you either click on an icon (macs) or type 'R' at the shell prompt (unix).

Elements of the R/S/S+ language

R is an object-oriented language. Everything (other than operators like +,-,*, etc) that you encounter will be some sort of object and these objects have *attributes* that confer certain properties.

Scalars

Scalar objects

Scalar objects hold one thing, a number, a word, a factor. These objects get their values through the use of the assignment operator. For example:

```
> height <- 10
```

assigns the value of 10 to the scalar object height. So the '<-' operator performs assignment. The "=" operator does the same thing. One occasionally useful aspect of the '<-' operator is that it can be used in two directions.

```
> width = 5
```

viewing contents of scalars

You can see the value of the scalar by just typing it at the command line:

```
> height

[1] 10

> width

[1] 5
```

Of course scalars can be constructed from other scalars too

```
> area <- height * width
> area

[1] 50
```

operators

Notice the use of another operator: '*' for multiplication. These operators work like you'd expect

```
> 10 + 1

[1] 11

> 10 * 2

[1] 20

> 10/3

[1] 3.333333

> 10 - 2

[1] 8
```

tests

There are also operators that help in testing of quantities:

```
> height == width

[1] FALSE

> height < width

[1] FALSE

> height > width

[1] TRUE

> height != width

[1] TRUE
```

Actions on objects

You can also apply functions to scalars (functions are special objects that perform an action). For example

```
> sqrt(height)

[1] 3.162278

> log(height)

[1] 2.302585
```

Finally, other objects such as character strings can be assigned to scalars

```
> nm <- "Suzy"
> nm

[1] "Suzy"
```

Vectors

Vectors

As I said in the introduction to this class, statistics is the study of variation. Scalars hold only a single value, so they aren't terribly useful objects for holding data to analyse. *Vectors* are a means to hold a number of values in a single object. To understand vectors you need to understand the 'c()' function.

c()

'c()' stands for concatenate and it takes all of its arguments and glues them together into a string of values, or a vector

```
> heights <- c(5, 10, 6, 15)
> heights

[1]  5 10  6 15

> widths <- c(7, 3, 4, 1)
> widths

[1] 7 3 4 1

> random.names <- c("red", "blue", "dog", "cat")
> random.names

[1] "red"  "blue" "dog"  "cat"
```

Digression on names of objects

Notice that the names of objects can contain a period. They can also contain, but not begin with numbers. They cannot contain operators because that confuses R.

functions on vectors

Functions can also operate on vectors

```
> heights

[1]  5 10  6 15

> sum(heights)

[1] 36

> widths

[1] 7 3 4 1

> sqrt(widths)

[1] 2.645751 1.732051 2.000000 1.000000
```

functions on vectors, cont.

You can also apply all kinds of operators to vectors as well, although the results might surprise you.

```
> heights + widths

[1] 12 13 10 16

> heights/widths

[1]  0.7142857  3.3333333  1.5000000 15.0000000

> areas <- heights * widths
> areas

[1] 35 30 24 15
```

Working with vectors

Once you are able to work with vectors statistics become possible. For example here are two ways to calculate the mean of a group of numbers (the function length() returns the length of the vector):

```
> length(heights)

[1] 4

> sum(heights)/length(heights)

[1] 9

> mean(heights)

[1] 9
```

Dataframes

Dataframes

	date	block	plant	diameter	damage
1	8/8/03	4	1	66	2
2	9/4/03	4	1	71	2
3	8/8/03	5	1	77	1
4	9/4/03	5	1	77	3
5	8/8/03	6	1	74	2
6	9/4/03	6	1	78	2
7	8/8/03	7	17	42	2
8	9/4/03	7	17	45	3
9	8/8/03	8	1	24	1
10	9/4/03	8	1	30	1
11	8/8/03	9	1	22	1
12	9/4/03	9	1	24	1

dataframes

A vector can hold the different observations or datapoints for a single variable. Most studies collect more than just one variable per individual. In R multiple variables can be stored in objects called dataframes.
A dataframe is essentially a matrix where columns are vectors that store the observations and rows represent a set of observations for a single individual. An example of a dataframe for data in the previous table looks like this in R (the dataset is called 'plantdata'):

```
> plantdata
```

	date	block	plant	diameter	damage
1	8/8/03	4	1	66.0	2
2	9/4/03	4	1	71.0	2
3	8/8/03	5	1	77.0	1
4	9/4/03	5	1	77.0	3
5	8/8/03	6	1	74.0	2

accessing individual variables in a

Each of the particular variables can be accessed by name (**you'll use this all the time**)

```
plantdata$diameter
```

```
 [1] 66.0 71.0 77.0 77.0 74.0 78.0 42.0 45.0 24.5 30.0 22.0 24.0 15.0 17.0 23.0
[16] 24.0 40.0 44.0 56.0 60.0 68.0 74.0 49.0 47.0
```

`plantdata$diameter` is the same as column '`diameter`' in the the dataset above

Row names

Dataframes have several additional characteristics. One is the rowname. Rownames are unique identifiers for each individual obervation in a matrix. Often (as in this case) rownames are just a series of integers

```
> rownames(plantdata)

 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15"
[16] "16" "17" "18" "19" "20" "21" "22" "23" "24"
```

Using indexes

The individual elements of a dataframe can be accessed by indexing. the syntax is `dataframe[1,2]` to retrieve the element at row 1, column 2.

```
> plantdata[1, 2]

[1] 4
```

indexing: assign individual observa

This construct can also be used in assignments

```
> tmp <- plantdata[1, 4]
> plantdata[1, 4] <- 100
> plantdata
```

	date	block	plant	diameter	damage
1	8/8/03	4	1	100.0	2
2	9/4/03	4	1	71.0	2
3	8/8/03	5	1	77.0	1
4	9/4/03	5	1	77.0	3
5	8/8/03	6	1	74.0	2
6	9/4/03	6	1	78.0	2
7	8/8/03	7	17	42.0	2
8	9/4/03	7	17	45.0	3
9	8/8/03	8	1	24.5	1
10	9/4/03	8	1	30.0	1
11	8/8/03	9	1	22.0	1
12	9/4/03	9	1	24.0	1
13	8/8/03	10	1	15.0	2
14	9/4/03	10	1	17.0	1

Slices of a dataframe

For all the variables for a single individual (an entire row):

```
> plantdata[2, ]

    date block plant diameter damage
2 9/4/03     4     1       71      2
```

For all the observations for a single variable:

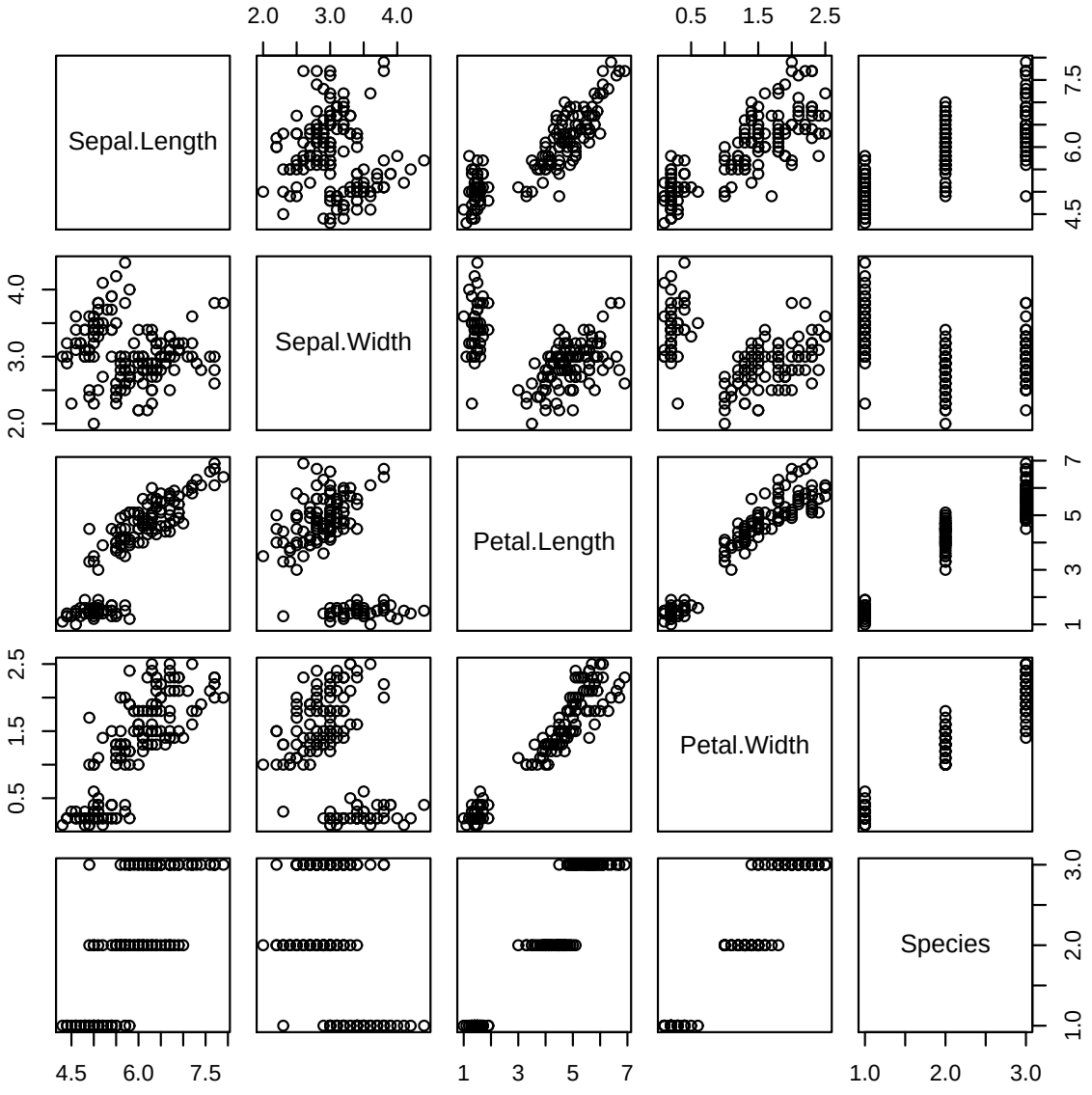
```
> plantdata[, 4]

 [1] 66.0 71.0 77.0 77.0 74.0 78.0 42.0 45.0 24.5 30.0 22.0 24.0 15.0 17.0 23.0
[16] 24.0 40.0 44.0 56.0 60.0 68.0 74.0 49.0 47.0
```

How are dataframes used?

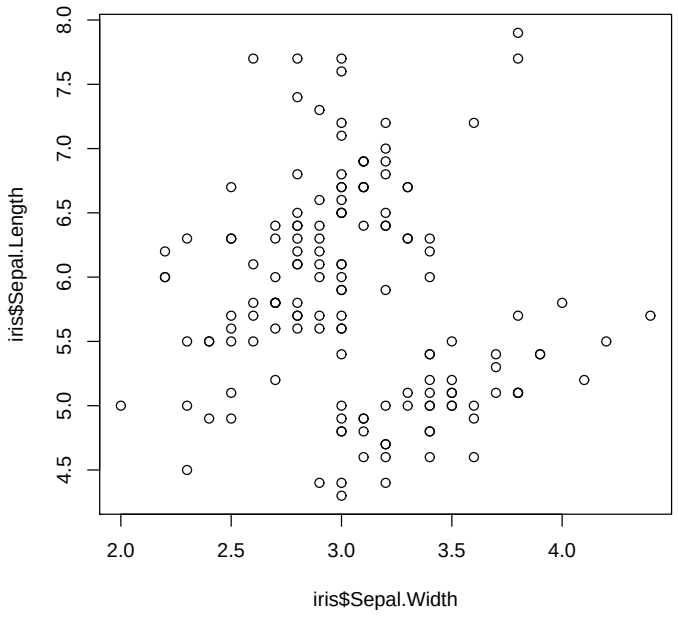
Dataframes will form the basis of most of what happens in R so it is worth checking out some of the things you can do. Without straying from plants, we are going to use a classical dataset that comes from Anderson (1935). He collected data on 50 individuals from each of three species of iris. The `data()` function loads a dataframe that is distributed with R or one of its packages.

```
> data(iris)
> names(iris)

[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

> unique(iris[, 5])

[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
```

plot(iris)



plot(iris$Sepal.Length iris$Sepal.W



functions and overloading

Several functions were just used on that dataframe. One interesting thing about the object oriented nature of R is *overloading* . This is the property that allows the same function to operate on different objects. Above, `plot` operated on both a dataframe and a *formula* object. The formula should be read `<dependent variables> ~<independent variables>`

One implication is that with well-behaved R packages, generic functions like `'summary()`, `print()`, and `plot()` produce meaningful results.

Conditioning

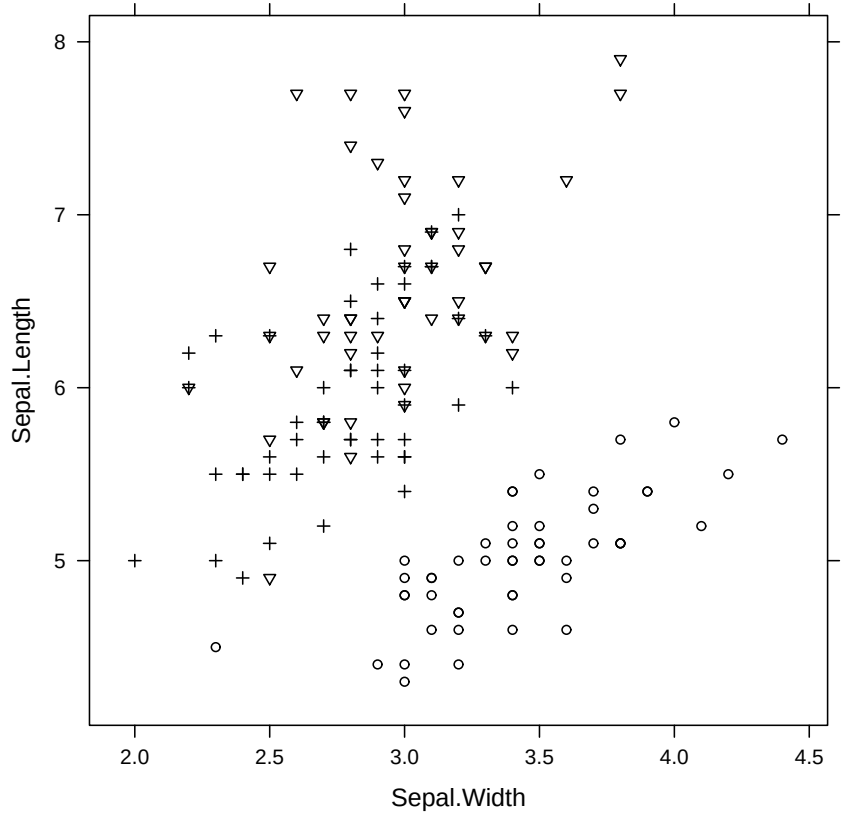
- It is pretty obvious from the previous two figures that there are some fairly complex things going on.
- One way to look at the data are to *condition* on some categorical variable.
- In R categorical variables are called *factors* `iris$Species` is a factor (see the `unique()` call)

```
> unique(iris$Species)

[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
```

- `xyplot` in the `lattice` library provides a means to do the conditioning.

xyplot()

```
> library(lattice)
> plot1 <- xyplot(Sepal.Length ~ Sepal.Width, group = Species,
+     data = iris)
> print(plot1)
```



Putting data into dataframes

Dataframes are the central type of object in R. In this section you can see how to populate dataframes.

1. construct from variables (vector columns)
2. construct from observations on individual (vector rows)
3. use datasets that are installed with R using the `data()` function (see the iris example in section ??)
4. enter data in a spreadsheet format by hand using the `edit()` command. This works really well and has a simple syntax: it's just `dataframename <- edit()`. To edit an existing dataframe: `dataframename <- edit(dataframename)`
5. Create data in another program like excel and import into R. This will be the most common approach

Data into dataframes II

1. Import datasets from another statistical program like SPSS.
2. Then for those with stronger stomachs:
   (a) text connections
   (b) direct connection with an SQL database
   (c) Use R as a DCOM server/client (direct connections with excel)
   (d) Inferface with GIS system (GRASS/Arcview)

Hand building dataframes from vec

This approach is ok for small amounts of data but it gets old quickly. We'll take advantage of vectors made earlier.

```
> shapes <- data.frame(widths, heights, areas)
> names(shapes)

[1] "widths"  "heights" "areas"

> shapes

  widths heights areas
1      7       5    35
2      3      10    30
3      4       6    24
4      1      15    15
```

Native R dataframes

Already seen this approach, but it is worth noting that there are lots of R datasets in R and its packages and it is possible to add more. `data()` with no arguments gives a list of the datasets available.

Using `edit()`

This approach is good for constructing a small dataframe or changing a few values in a large dataframe.
This approach will be demonstrated interactively.

Importing data from excel files

One of the easiest methods to get lots of data into R is to construct your dataframe in excel, export it, and then import it into R.
This approach will be demonstrated interactively, but it is worth noting that this will probably be the most frequent means of data processing.

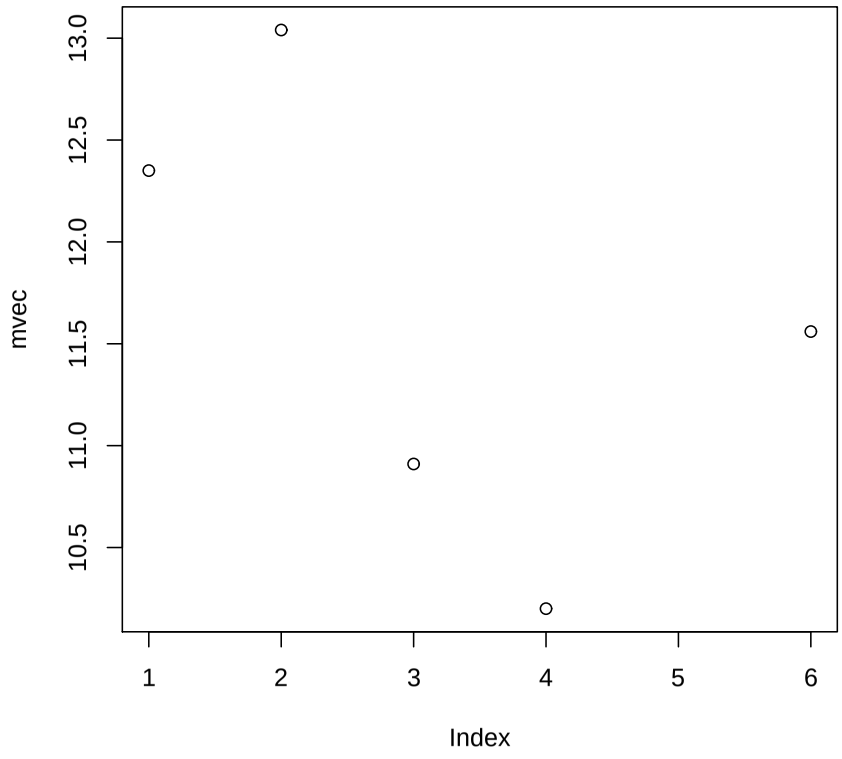
Missing values

Missing values

Missing data are a reality. It's going to happen to you. In R missing data are denoted as `NA`.
It is important to understand what functions do with missing values as arguments

```
> 1 + NA

[1] NA

> mvec <- c(12.35, 13.04, 10.91, 10.2, NA, 11.56)
> mvec

[1] 12.35 13.04 10.91 10.20    NA 11.56

> mean(mvec)

[1] NA

> mean(mvec, na.rm = TRUE)
```

plot(mvec)



Manipulating dataframes

More on indexing

Indexing in S/R is a very powerful (read sometimes tricky) way to manipulate data.

```
> iris[1, ]

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa

> iris[c(1, 2), ]

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa

> iris[, 2]

  [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 3.4 3.0 3.0 4.0 4.4 3.9 3.5
 [19] 3.8 3.8 3.4 3.7 3.6 3.3 3.4 3.0 3.4 3.5 3.4 3.2 3.1 3.4 4.1 4.2 3.1 3.2
 [37] 3.5 3.6 3.0 3.4 3.5 2.3 3.2 3.5 3.8 3.0 3.8 3.2 3.7 3.3 3.2 3.2 3.1 2.3
```

Boolean operators

There is a complete set of boolean operators in R

```
> a <- 1
> b <- 2
> a == b

[1] FALSE

> a > b

[1] FALSE

> a < b

[1] TRUE

> a >= b

[1] FALSE
```

if/then

S/R is a true programming language and implements if/then/else, so boolean operators can be used like this

```
> if (a > b) {
+     print("a > b")
+ } else {
+     print("b > a")
+ }

[1] "b > a"
```

Logical vectors

Boolean objects can be used in vector form to do lots of things in R. These logical vectors can be used as *selection vectors* .

```
> c <- NA
> is.na(c)

[1] TRUE

> is.na(a)

[1] FALSE

> is.na(mvec)

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

Selection vector

```
> avec <- c(1, 2, 3, 4)
> asel <- c(T, T, F, T)
> avec[asel]

[1] 1 2 4
```

Complex manipulations

```
> plantvec <- (plantdata[, 2] < 8) & (plantdata[, 3] == 17)
> plantdata[plantvec, ]

    date block plant diameter damage
7 8/8/03     7    17       42      2
8 9/4/03     7    17       45      3
```

Removing NA's

One typical application is in manipulating missing values

```
> mvec

[1] 12.35 13.04 10.91 10.20    NA 11.56

> mvec[is.na(mvec)]

[1] NA

> mvec[!is.na(mvec)]

[1] 12.35 13.04 10.91 10.20 11.56

> mean(mvec[!is.na(mvec)])

[1] 11.612

> mean(mvec, na.rm = T)
```

Duplicating rows/columns

Rows and columns can be duplicated in S/R. Make sure to update the corresponding row/column name.

```
> iris.dup <- iris[, c(1, 1, 2, 3, 4, 5)]
> names(iris.dup)

[1] "Sepal.Length"   "Sepal.Length.1" "Sepal.Width"    "Petal.Length"
[5] "Petal.Width"    "Species"

> names(iris.dup)[1] <- "dup1"
> names(iris.dup)

[1] "dup1"           "Sepal.Length.1" "Sepal.Width"    "Petal.Length"
[5] "Petal.Width"    "Species"
```

Removing rows/columns

The same type of mainupulation can be used to remove row/columns

```
> iris.remove <- iris[, c(-1, -2)]
> names(iris.remove)

[1] "Petal.Length" "Petal.Width"  "Species"
```

Visualizing data

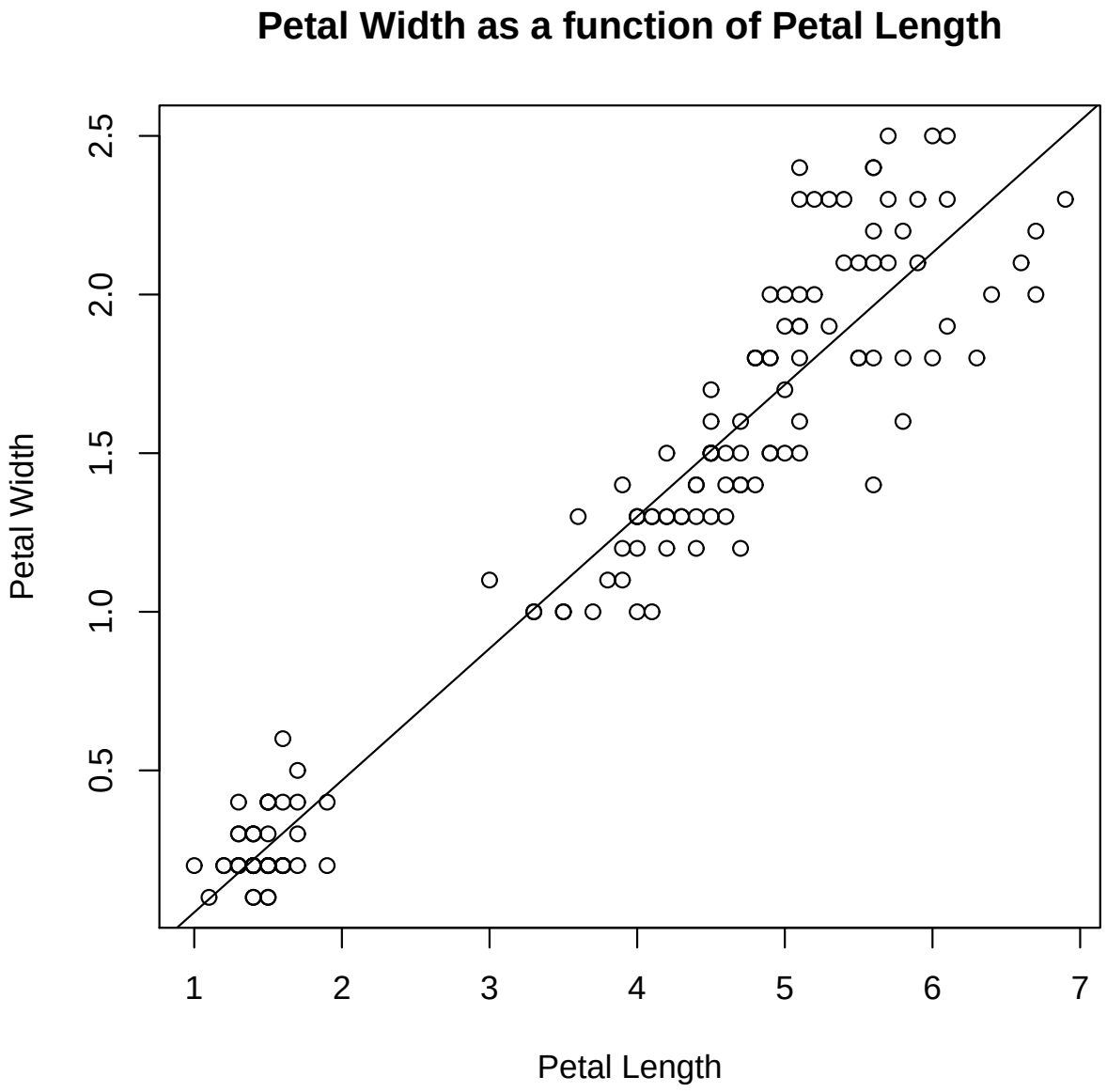
Presenting data graphically

- Your eye is remarkably perceptive in seeing patterns in data. Good visual data presentation can take advantage of that innate ability.
- This section is intended to demonstrate some of the uses of two types of R graphics, base graphics (revolves around `plot()`) and lattice graphics.
- Like many functions in S/R, `plot()` can operate with relatively few arguments, but it can be customized through additional arguments
  - `main` title of plot
  - `xlab, ylab` labels for x and y axes
  - `ylim, xlim` vectors of length 2 that specify the limits of the y and x axes, respectively
  - `help(plot)` for more

Plot example: code

```
> attach(iris)
> plot(Petal.Width ~ Petal.Length, main = "Petal Width as a function of Petal Leng
+     xlab = "Petal Length", ylab = "Petal Width")
> petal.lm <- lm(Petal.Width ~ Petal.Length)
> abline(petal.lm$coef)
> detach(iris)
```

Plot example: product

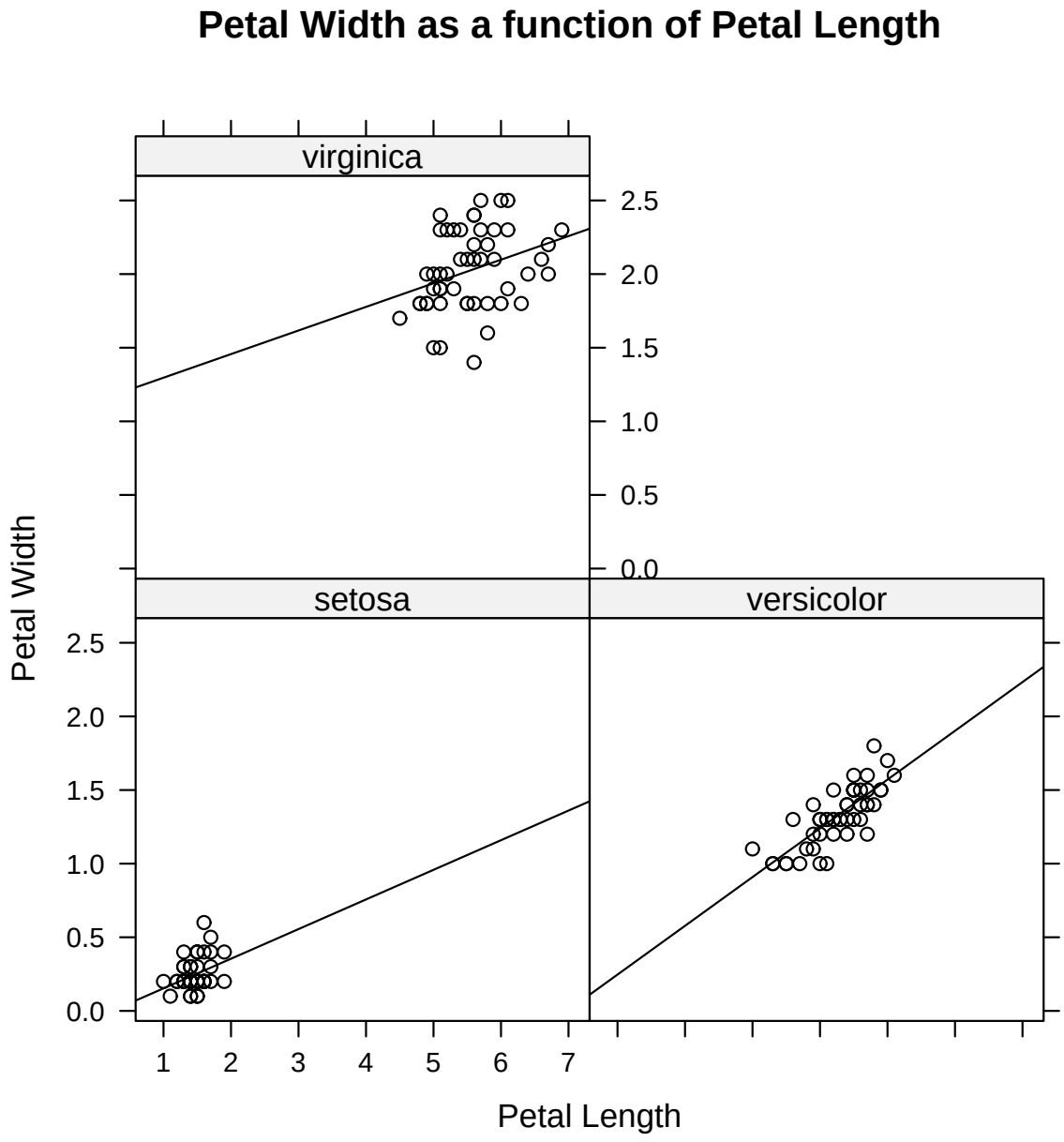


Glimpse at lattice

Lattice graphics used in the correct circumstances are much more powerful than base graphics, and of course it is more difficult to use them.

```
> library(lattice)
> xyobj <- xyplot(Petal.Width ~ Petal.Length | Species, main = "Petal Width as a
+     xlab = "Petal Length", ylab = "Petal Width", panel = function(...) {
+         panel.xyplot(...)
+         panel.lmline(...)
+     }, data = iris)
> print(xyobj)
```

Product of print(xyobj)

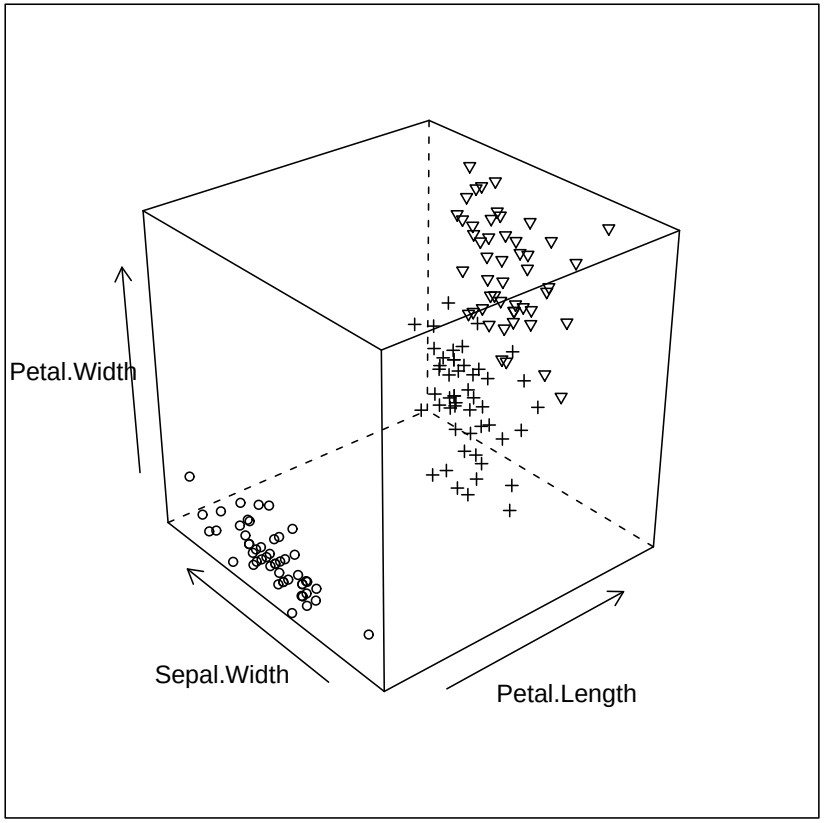


3d

Here is a 3d version of lattice graphics that adds an additional variable/dimension to the plot.

```
> cldobj <- cloud(Petal.Width ~ Petal.Length + Sepal.Width, group = Species,
+     data = iris)
> print(cldobj)
```

3d product



lattice versus base graphics

- Base graphics are great for most one and two-variable plots.
- Lattice graphics come in handy when there are one and especially if there are multiple categorizing variables (e.g.: gender, species)

Plotting geography

Plotting in R can produce a diverse set of figures. For example, if you are interested in plotting sampling locations, but don't care about the analytical capabilities of GIS, R provides a nice solution.

The files "sccoast.dat", "rivers.dat", and "polboundaries.dat"

are downloaded from NOAA's coastline extractor .

Plotting a map, data proc

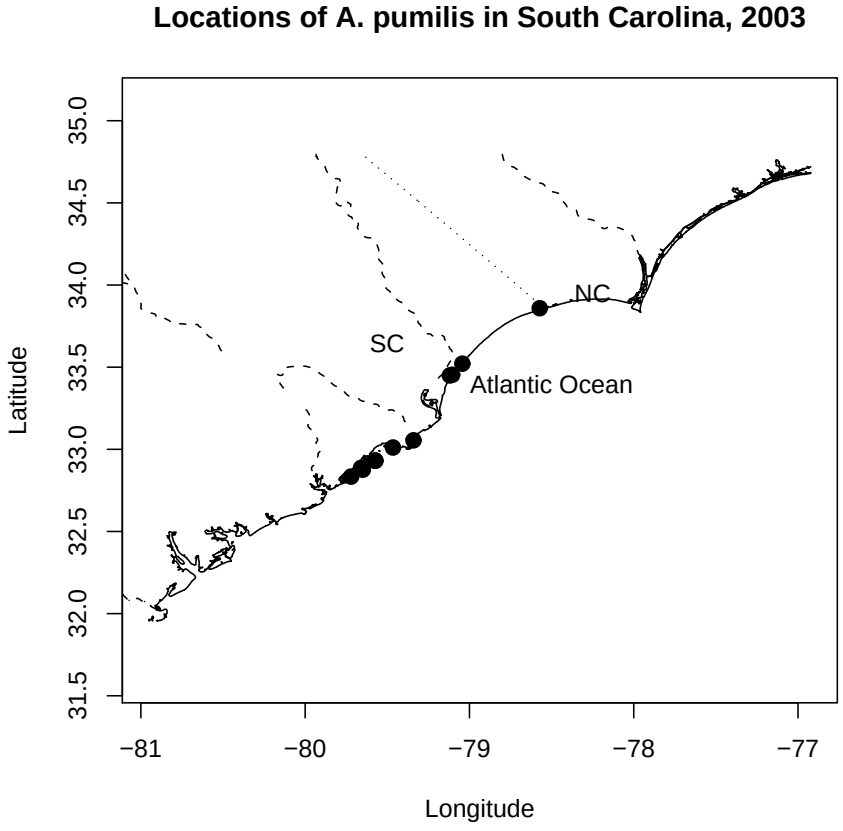
```
> coast <- read.table("sccoast.dat", header = F)
> rivers <- read.table("rivers.dat", header = F)
> polbound <- read.table("polboundaries.dat", header = F)
> sites <- read.csv("sc_census_sites.csv", header = T)
> attach(sites)
> lat <- latd + ((latm + lats/60)/60)
> long <- (-1) * (longd + ((longm + longs/60)/60))
> detach(sites)
> sites <- cbind(sites, lat, long)
> names(sites)

 [1] "location"   "latd"       "latm"       "lats"       "longd"
 [6] "longm"      "longs"      "collection" "lat"        "long"
```

Now the data are read into different dataframes in R. The following commands plot these dataframes using the `plot()` and `points()` functions. Then the map is annotated using the `text()` function.

```
> plot(coast, type = "l", lty = 1, xlab = "Longitude", ylab = "Latitude",
+     main = "Locations of A. pumilis in South Carolina, 2003",
+     asp = 1)
> points(rivers, type = "l", lty = 2)
> points(polbound, type = "l", lty = 3)
> attach(sites)

        The following object(s) are masked _by_ .GlobalEnv :

         lat long

> points(long, lat, type = "p", lty = 1, pch = 20, cex = 2)
> detach(sites)
> text(-78.25, 33.95, "NC")
```

Map product

```
The following object(s) are masked _by_ .GlobalEnv :

 lat long
```



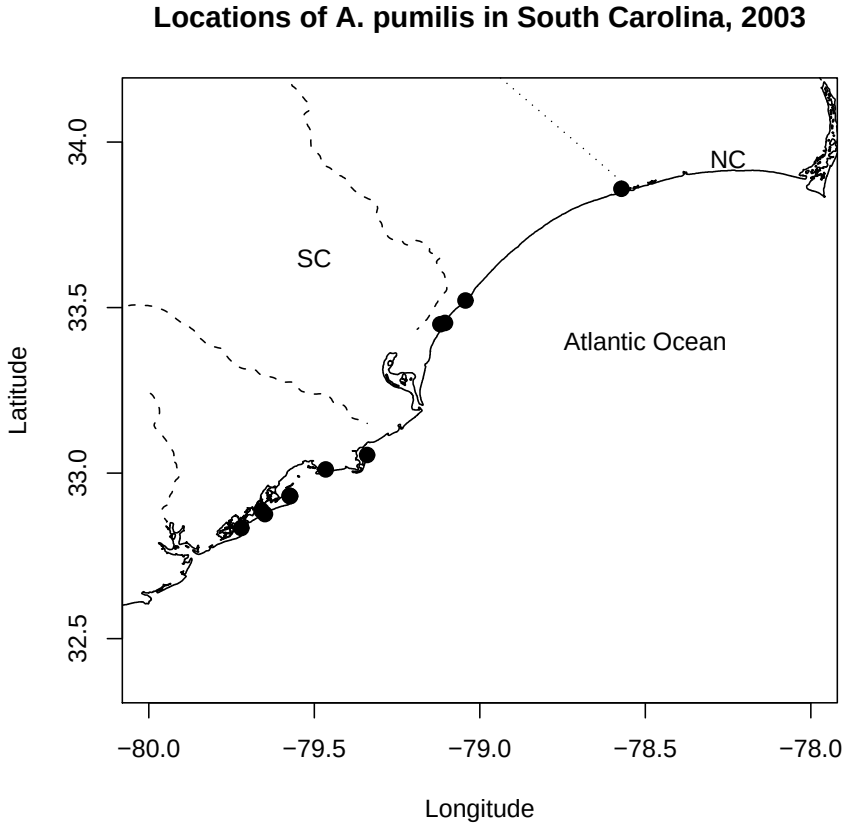
Changing limits on axes

That map is kind of small scale. You can increase the scale by just limiting the size of the plotted figure using `ylim` and `xlim` Here is an example (only the `plot()` command is changed)

```
> plot(coast, type = "l", lty = 1, xlab = "Longitude", ylab = "Latitude",
+     main = "Locations of A. pumilis in South Carolina, 2003",
+     asp = 1, ylim = c(32.5, 34), xlim = c(-80, -78))
> points(rivers, type = "l", lty = 2)
> points(polbound, type = "l", lty = 3)
> attach(sites)

        The following object(s) are masked _by_ .GlobalEnv :

         lat long

> points(long, lat, type = "p", lty = 1, pch = 20, cex = 2)
> detach(sites)
> text(-78.25, 33.95, "NC")
```

Product

```
The following object(s) are masked _by_ .GlobalEnv :

 lat long
```



Other plotting functions

There are many other functions for visualizing data in R, most of them also process data passed to them more significantly than `plot()`, `xyplot()`, so they will be dicussed in more analytical lectures later.

Help using R

Internet R resources

Almost anything that has been pulished on how to use S or S+ is equally applicable to R. One notable exception relevant to this class is the use of trellis graphics in S is a bit different than R. Many books have been published in the last decade on S and our library has a small collection. Books are expensive on a graduate student's budget and luckily there are quite a few online resources that can help. The best place to go is the R-project website .

On-line help

Once you get going with R, you will find that the online help is very useful. There are several ways to invoke the online help system from the command line:

```
> help(mean)
```

This approach gives the help page for 'mean' in the R session. You could also achieve the same end by saying `?mean` at the command prompt.

help.start()

On most systems the following command:

```
> help.start()
```

opens a web-browser that allows you to navigate through the help system.

Searching for help

There are times when you want to do something, you know it's possible, but you cannot figure out the name of the function. There are two commands that can be used in this case:

```
> apropos("mean")
```

```
 [1] "colMeans"          "kmeans"            "mean"              "mean.data.frame"
 [5] "mean.Date"         "mean.default"      "mean.difftime"     "mean.POSIXct"
 [9] "mean.POSIXlt"      "rowMeans"          "weighted.mean"
```

and

```
> help.search("mean")
```

The second of these approaches often returns more information, and it is displayed on the terminal in a formatted

Using web browser for local help s

After starting the web-browser help system using help.start(), it is possible to search through the web-pages on R.

Searching elsewhere for help

Finally, there is a mailing-list called r-help that has an active ongoing discussion of issues in R. I'd not recommend that you post to the list until you have some experience, but it may be interesting to subscribe and lurk.

You can take advantage of the list though, by using google and typing your question and 'r-help' in the search field. This searches through years of messages. Often you can find a worked out example, a pointer, or some other help this way.

References

Anderson, E. (1935). The irises of the gaspe peninsula, *Bulletin of the American Iris Society* **59**: 2–5.