Variables - Reprise

Discrete Attribute:

Has only a finite or countably infinite set of values
Examples: zip codes, genotypes, gender, colors in red light, smoker
Often represented as integer variables.
Note: binary attributes are a special case of discrete attributes

Continuous Attribute:

Has real numbers as attribute values
Examples: temperature, height, or weight.
Practically, real values can only be measured and represented using a finite number of digits.
Continuous attributes are typically represented as floating-point variables.

Tan, Steinbeck, Kumar

Variables - Reprise

Categorical:

Nominal

Description: The values are different names and provide enough information to distinguish one from another

Examples: ID numbers, colors, eye color, zip codes

Operations: mode, contingency correlation, Chi-square test

Ordinal

Description: The values provide enough information to order objects

Examples: rankings (e.g., taste of potato chips on a scale from 1-5), grades, height in {tall, medium, short}

Operations: median, percentiles, rank correlations, run tests, sign tests

Tan, Steinbeck, Kumar

Variables - Reprise

Numeric:

Interval

Description: The differences between values are meaningful

Examples: Calendar dates, time, PH, temperature in Celsius or Farenheit

Difference between 5pm and 4pm is the same as 4am and 3am

Operations: mean, standard deviation, Pearson's correlation, t and F tests

Ratio

Description: Both differences and ratios make sense

Examples: temperature in Kelvin, monetary quantities, counts, age, mass, length, electrical current

Operations: geometric mean, harmonic mean, percent variations

Be careful - "color" is usually categorical - nominal but what about in a phyics experiment ? Its part of a spectrum and might be measured in terms of some continuous quantity ?

Tan, Steinbeck, Kumar

Variables - Reprise

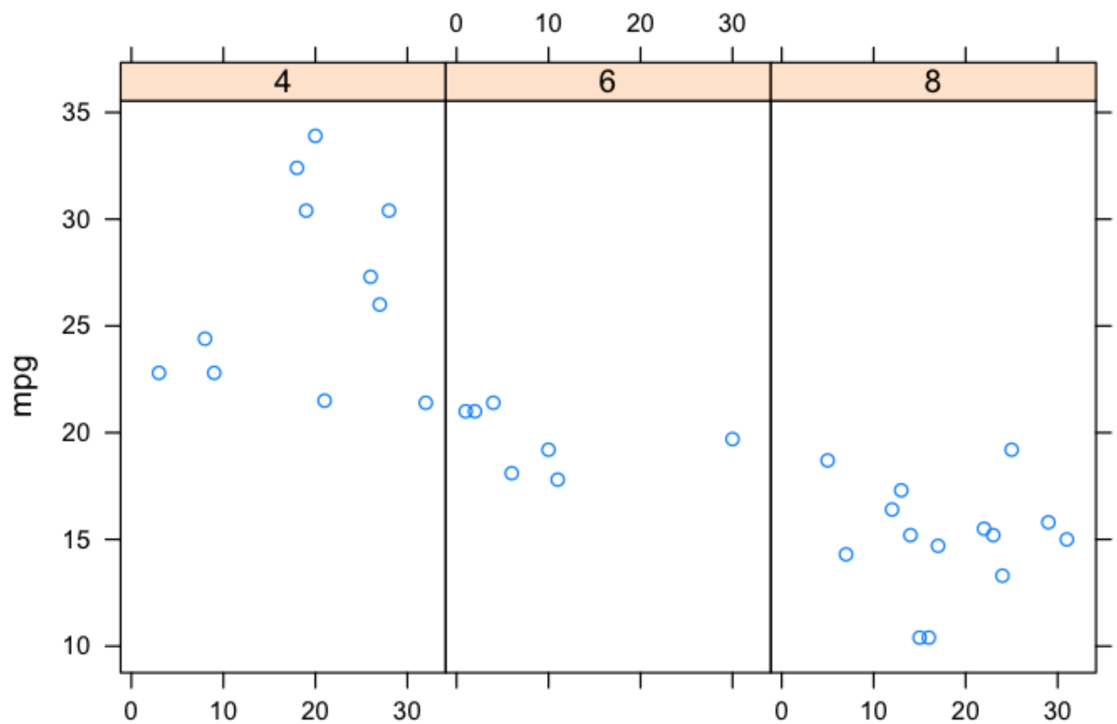
```
> head(mtcars,15)
                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4          21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710         22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive     21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant            18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360         14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D          24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230           22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280           19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C          17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE         16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL         17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC        15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
```

Variables - Reprise

Which variables are
Continuous ? Discrete ? Categorical ?

Variables - Reprise

```
xyplot(mpg~1:nrow(mtcars)|factor(cyl),data=mtcars)
```




```
aggregate(mpg~cyl,data=mtcars,mean)
  cyl      mpg
1   4 26.66364
2   6 19.74286
3   8 15.10000
```

Variables - Objects

Everything, (vector, factor, matrix, array, list , data.frame), is an **object**, which also has a **type** and belongs to a **class**:

```
3+5
[1] 8

typeof(3)
[1] "double"

class(3) # “class” and “mode” can be used interchangeably
[1] "numeric"

typeof(`+`)
[1] "builtin"
```

`Use the "str" function() - It is a good summary command`

```
str(3)
num 3
```

Variables - Objects - Numeric

The four primary variable classes are: **numeric**, character, factor, and dates. Its important to know how to manipulate these and , if necessary, convert between them.

Don't rush through these basic concepts as you will almost always have to change the type of a given variable to apply a function or statistical procedure.

```
var1 = 3
var1
[1] 3

sqrt(var1)
[1] 1.732051

var1 = 33.3

str(var1)
 num 33.3
```

Variables - Objects - Numeric

The four primary variable classes are: **numeric**, character, factor, and dates. Its important to know how to manipulate these and , if necessary, convert between them.

Don’t rush through these basic concepts as you will almost always have to change the type of a given variable to apply a function or statistical procedure.

```
myvar = 5

myvar + myvar # Addition
[1] 6

myvar - myvar      # Subtraction
[1] 0

myvar * myvar # Multiplication
[1] 9

myvar / myvar # Division
[1] 1

myvar ^ myvar # myvar raised to the power of myvar
[1] 3125
```

Variables - Objects - Numeric

Its worth pointing out that there is a distinction between integers and real values. Don't worry too much about this now but keep it in mind. If you really want an integer then you have to "ask" for it:

```
aa = 5

class(aa)
[1] "numeric"

str(aa)
 num 5

aa = as.integer(aa)  # We use a "coercion" function here

class(aa)
[1] "integer"

aa = 5.67

as.integer(aa)   # Truncates the value - note it doesn't round.
[1] 5
```

Variables - Objects - Character

Character strings are possible also. This variable type is for when you wish to store informative labels about something. Generally speaking string variables are "descriptive" whereas numeric variables are quantitative.

```
var.one = "Hello there ! My name is Steve."
var.two = "How do you do ?"

var.one
[1] "Hello there ! My name is Steve."

nchar(var.one)     # Number of characters present
[1] 31

toupper(var.one)
[1] "HELLO THERE ! MY NAME IS STEVE."

mydna = c("A","G","T","C","A")

# See BioConductor http://www.bioconductor.org/

str(mydna)
 chr [1:5] "A" "G" "T" "C" "A"

mydna
[1] "A" "G" "T" "C" "A"
```

Variables - Objects - Character

Character strings are possible also. This variable type is for when you wish to store informative labels about something. Generally speaking string variables are "descriptive" whereas numeric variables are quantitative.

```
paste(var.one,var.two)
[1] "Hello there ! My name is Steve. How do you do ?"

paste(var.one,var.two,sep=":")
[1] "Hello there ! My name is Steve.:How do you do ?"

strsplit(var.one," ")
[[1]]
[1] "Hello"  "there"  "!"      "My"     "name"   "is"     "Steve."


patientid = "ID:011472:M:C"  # Encodes Birthday, Gender, and Race

strsplit(patientid,":")
[[1]]
[1] "ID"     "011472" "M"      "C"

bday = strsplit(patientid,":")[[1]][2]  # Get just the birthday
```

Variables - Objects - Dates

Dates are an important data type in R. In many cases dates are treated as characters when printing. However, dates can be operated upon arithmetically. If you work a lot with dates then consider working with the “zoo” package which handles Time Series data quite well.

```
Sys.Date()
[1] "2011-08-01"

Sys.Date()+1
[1] "2011-08-02"

class(Sys.Date())
[1] "Date"

string = "2011-04-27”
class(string)
[1] "character"

as.Date(string)
[1] "2011-04-27"
```

Variables - Objects - Dates

If your input dates are not in the standard format, a format string can be composed using the elements shown in Table . The following examples show some ways that this can be used:

```
as.Date("03/17/1996")
Error in charToDate(x) :
  character string is not in a standard unambiguous format

as.Date("03/17/1996",format="%m/%d/%Y")
[1] "1996-03-17"

as.Date('1/15/2001',format='%m/%d/%Y')
[1] "2001-01-15"

as.Date('April 26, 2001',format='%B %d, %Y')
[1] "2001-04-26"

as.Date("2012-10-27")
[1] "2012-10-27"
```

Code	Value
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated)
%B	Month (full name)
%y	Year (2 digit)
%Y	Year (4 digit)

Variables - Objects - Dates

Once you get your date formatted you can access parts of it easily:

```
my.date = as.Date("2012-10-27")
my.date - 1
[1] "2012-10-26"

format(my.date,"%Y")   # Note all of these are character strings
[1] "2012"

format(my.date,"%b")
[1] "Oct"

format(my.date,"%y")
[1] "12"

format(my.date,"%b %d")
[1] "Oct 27"

format(my.date,"%b %d, %Y")
[1] "Oct 27, 2012"
```

Code	Value
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated)
%B	Month (full name)
%y	Year (2 digit)
%Y	Year (4 digit)

Variables - Objects - Dates

The difftime function let's us pass character strings to it.

```
difftime("2005-10-21 08:32:58","2003-8-15 09:18:05")

Time difference of 797.9687 days
```

Here is one way to deal with date strings say from an Excel file. Let's say that the dates are in month/day/year format:

```
strptime(c("03/27/2003","03/27/2003","04/14/2008"),format="%m/%d/%Y")
"2003-03-27" "2003-03-27" "2008-04-14"
```

Variables - Objects - Dates

```
rdates
   Release       Date
1      1.0 2000-02-29
2      1.1 2000-06-15
3      1.2 2000-12-15
4      1.3 2001-06-22
5      1.4 2001-12-19
6      1.5 2002-04-29
7      1.6 2002-10-10
8      1.7 2003-04-16
9      1.8 2003-10-08
10     1.9 2004-04-12
11     2.0 2004-10-04

mean(rdates$Date)
[1] "2002-05-20"

range(rdates$Date)
[1] "2000-02-29" "2004-10-04"

rdates$Date[11] - rdates$Date[1]
Time difference of 1679 days
```

	A	B
1	2007-09-05	$ 590,00
2	2007-09-15	$ 791,00
3	2007-10-30	$ 982,00
4	2007-08-28	$ 889,00
5	2007-10-05	$ 210,00
6	2007-09-17	$ 349,00
7	2007-10-09	$ 693,00
8	2007-11-18	$ 337,00
9	2007-10-23	$ 734,00
10	2007-09-03	$ 10,00
11	2007-10-18	$ 331,00
12	2007-09-28	$ 751,00
13	2007-10-24	$ 488,00
14	2007-11-20	$ 479,00
15	2007-11-07	$ 946,00
16	2007-11-11	$ 665,00
17	2007-11-21	$ 711,00
18	2007-09-01	$ 625,00
19	2007-09-05	$ 797,00

```
Data Manipulation with R – Phil Spector pp 64
```

Variables - Objects - Logical

Logical variables are those that take on a TRUE or FALSE value. Either by direct assignment or as the result of some comparison:

```
some.variable = TRUE

some.variable = (4 < 5)
```

This is an important type because it will eventually allow us to construct expressions to be used in if statements for programming

```
if (some_logical_condition) {
   do something
}

if ( 4 < 5 ) {
  print(“Four is less than Five”)
}
```

Variables - Objects - Logical

```
if ( 4 < 5 ) {
  print("Four is less than Five")
}

my.var = ( 4 < 5)

if (my.var) {
  print("four is less than five")
}

if (! my.var ) {
   print("four is greater than five")
}

my.var = (4 < 5) & ( 4 < 6 )  # Logical AND operator
```

Both expressions must be TRUE in order for the combined expression to return TRUE.

```
my.var
[1] TRUE

my.var = (4 < 5) | ( 4 < 6 ) # Logical OR operator.
my.var = TRUE
```

Only one of these expressions needs to be TRUE for the entire expression to be TRUE

Variables - Objects - Interrogating

It is a common operation to want to interrogate variables to see what they are (or aren't). There is an entire family of "is" functions to accomplish this type of activity. It is also common to "coerce" variables into another type. There is an entire family of "as" functions to accomplish this.

Interrogation	Coercion	Type of Operand
`is.array()`	`as.array()`	Arrays
`is.character()`	`as.character()`	Character
`is.data.frame()`	`as.data.frame()`	Data Frames
`is.factor()`	`as.factor()`	Factors
`is.list()`	`as.list()`	Lists
`is.logical()`	`as.logical()`	Logical
`is.matrix()`	`as.matrix()`	Matrix
`is.numeric()`	`as.numeric()`	Numeric
`is.vector()`	`as.vector()`	Vector

Variables - Objects - Interrogating

It is a common operation to want to interrogate variables to see what they are (or aren't).

```
my_int = as.integer(5)

is.integer(my_int)  # These are good for use in programming loops
[1] TRUE

is.numeric(my_int)
[1] TRUE

is.character(my_int)
[1] FALSE

is.logical(my_int)
[1] FALSE
```

Variables - Objects - Interrogating

It is also common to “coerce” variables into another type. There is an entire family of “as” functions to accomplish this.

```
my_int = as.integer(5)

as.character(my_int)
[1] "5"

as.integer(as.character(my_int))
[1] 5

my_number = 12.345

as.character(my_number)
[1] "12.345"

as.logical(1)
[1] TRUE

as.character(as.logical(1))
[1] "TRUE"
```

Variables - Objects - Interrogating

These types of functions show up frequently when users write their own functions. The "is" functions assist with checking the arguments for the correct type.

```
my.func = function(x) {

       if (!is.numeric(x) ) {
          stop("Hey. I need a numeric vector here")
       } else {
          return(mean(x))
       }
}
```

Vectors

R implements vectors as a fundamental object type so get familiar with them. It is useful for storing data of all the same type in a variable / data structure. You can put in variables of type integer, logical, real, and character.

```
1:10

rnorm(10)

y = 5.4     # A single assignment

y = 1:10    # A vector with 10 elements (1 .. 10)

y = c(1,2,3,4,5,6,7,8,9,10) # Same as above yet using the “c” function

y = scan()   # Allows you to enter in elements from the keyboard
1: 10
2:  9
3:  8

..
1:  1
```

Vectors - Indexing

Let's say we have measured the heights of some people and want to stash it in a vector. Bracket notation is the key to working vectors !

```
height = c(59,70,66,72,62,66,60,60)  # create a vector of 8 heights

Let's check out the various ways we can index into the vector

height[1:5]             # Get first 5 elements
[1] 59 70 66 72 62

height[5:1]             # Get first 5 elements in reverse
[1] 62 72 66 70 59

height[-1]              # Get all but first element
[1] 70 66 72 62 66 60 60

height[-1:-2]           # Get all but first two elements
[1] 66 72 62 66 60 60

height[c(1,5)]               # Get just first and fifth elements
[1] 59 62
```

Vectors - Logicals

Let's say we have measured the heights of some people and want to stash it in a vector:

```
height = c(59,70,66,72,62,66,60,60) # create a vector of 8 heights
```

We can apply logical tests to a vector to find elements that satisfy a condition set

```
height
[1] 59 70 66 72 62 66 60 60

height == 72         # Test for values equal to 72
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE

height[height == 72]
[1] 72

# SAME AS

logical.vector = (height == 72)

logical.vector
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE

height[ logical.vector ]
```

Vectors - Logicals

Let’s say we have measured the heights of some people and want to stash it in a vector:

```
> height = c(59,70,66,72,62,66,60,60)    # create a vector of 8 heights
```

Comparisons can be combined like a database query:

```
# Note use of the "&" / and operator

height[height > 60 & height < 70]
66      62      66

height[height > 60 & height <= 70]
70      66      62      66
```

Vectors - compared to for loop

It is important to point out that using logical operations within brackets eliminates the need to write a "for loop" every time you want to do some summary information on a vector. Which would you rather do:

This:

```
height[height > 60 & height < 70]
66      62      66
```

OR:

```
for (ii in 1:length(height)) {
    if (height[ii] > 60 & height[ii] < 70) {
       print(height[ii])
   }
}

     66
     62
     66
```

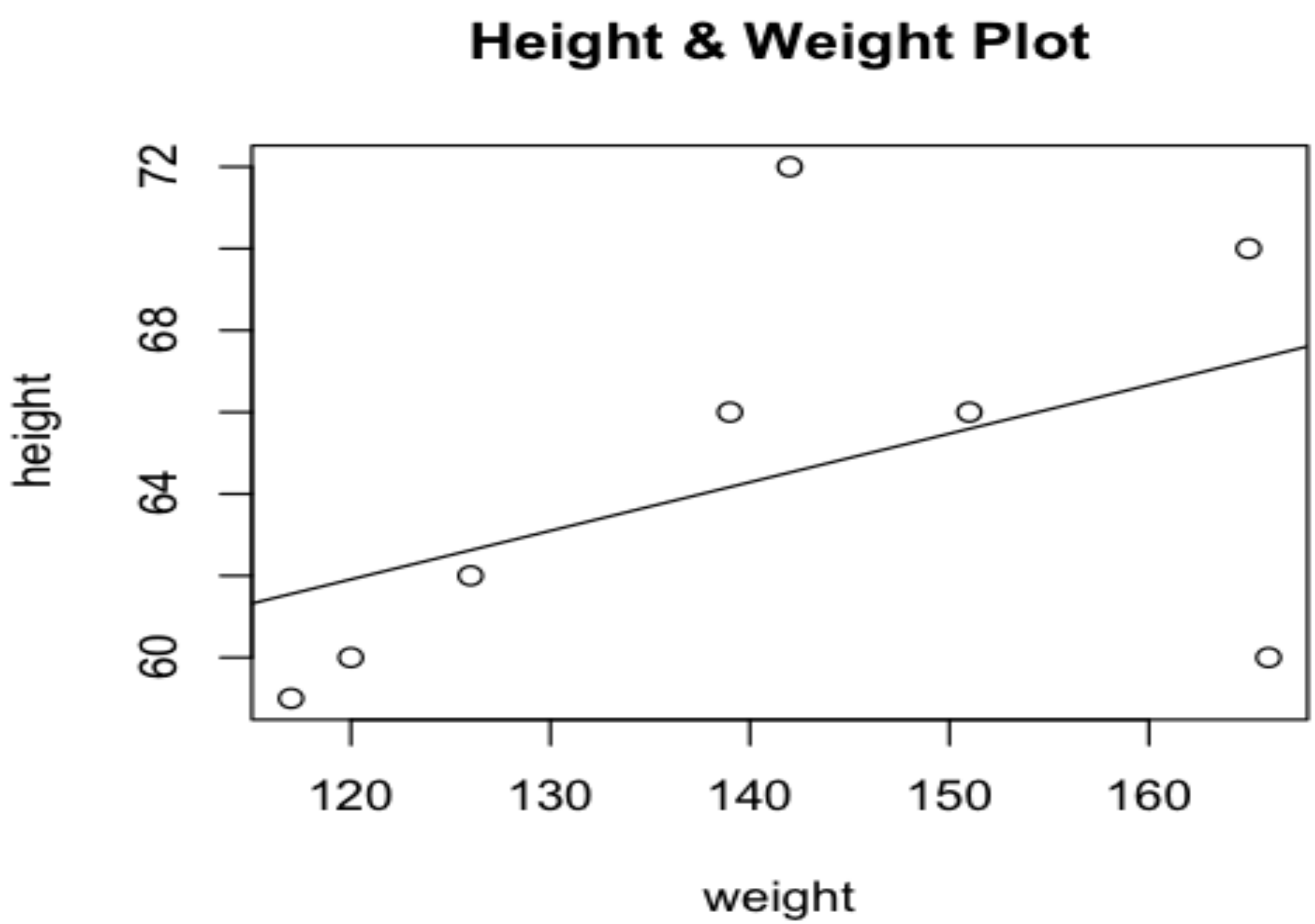
Vectors - Arithmetic

```
weight = c(117,165,139,142,126,151,120,166)   # weight (in lbs)

weight/100
[1] 1.17 1.65 1.39 1.42 1.26 1.51 1.20 1.66

sqrt(weight)
[1] 10.81665 12.84523 11.78983 11.91638 11.22497 12.28821 10.95445 12.88410

weight^2
[1] 13689 27225 19321 20164 15876 22801 14400 27556

sum((weight-mean(weight))^2)/(length(weight)-1)  # The variance formula
[1] 363.9286

var(weight)
[1] 363.9286
```

Supplemental - Vectors - lm

```
height = c(59,70,66,72,62,66,60,60)

weight = c(117,165,139,142,126,151,120,166)
# Get 8 weight measurements

cor(height,weight)            # Are they correlated ?
[1] 0.46295

plot(weight,height,main="Height & Weight Plot")  # Do a X/Y plot

res = lm(height ~ weight)     # Do a linear regression

abline(res)          # Check out the regression line
```

Supplemental - Vectors - lm



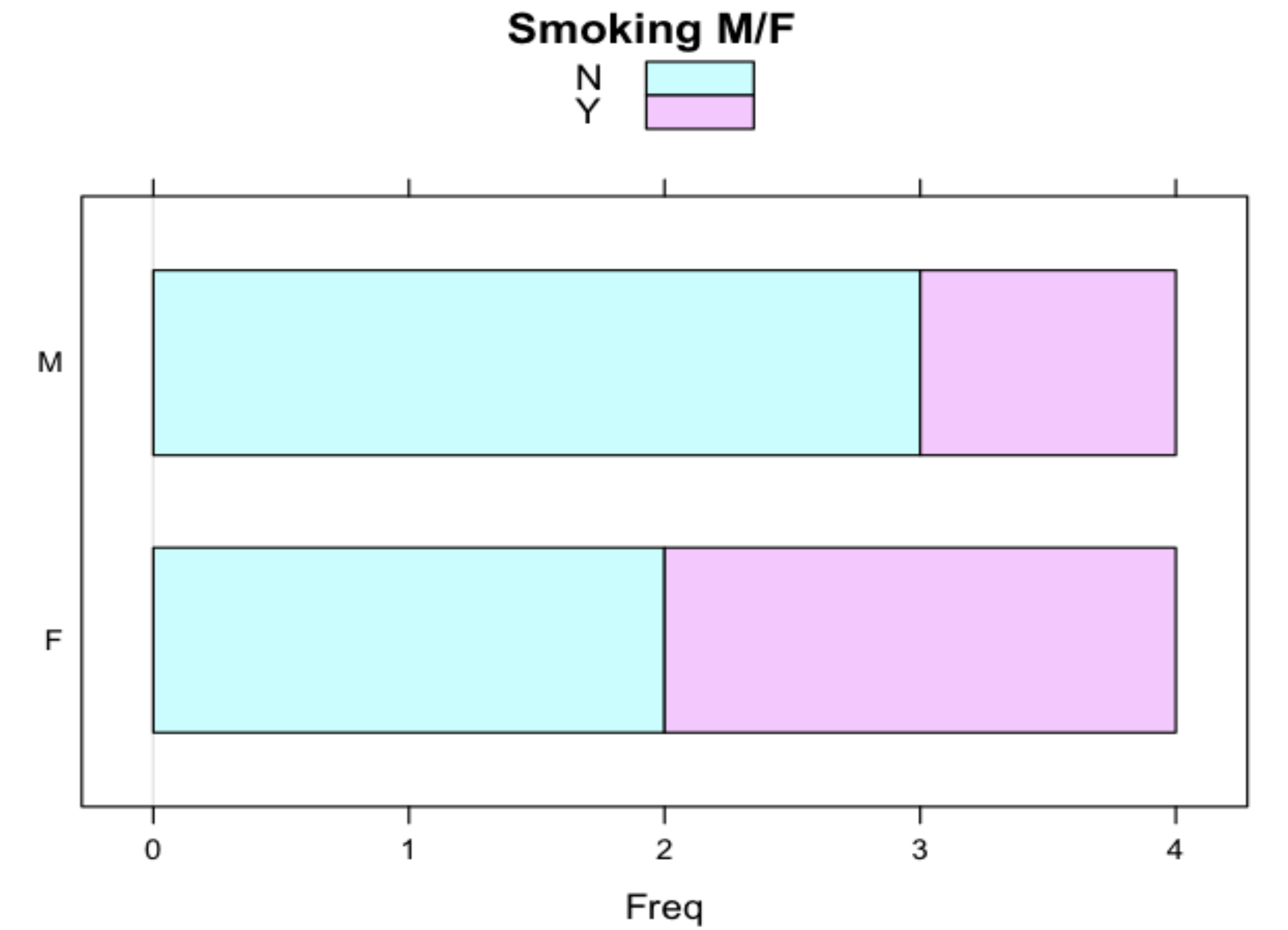
Vectors - Arithmetic

```
weight = c(117,165,139,142,126,151,120,166)   # weight (in lbs)

new.weights = weights + 1      # Vector Addition

new.weights
[1] 118 166 140 143 127 152 121 167

append(weights,new.weights)    # Combines the two vectors
[1] 117 165 139 142 126 151 120 166 118 166 140 143 127 152 121 167

c(weight,new.weights)              # Equivalent to the above


weight/new.weights        # Vector division

[1] 0.9915254 0.9939759 0.9928571 0.9930070 0.9921260 0.9934211 0.9917355 0.9940120
```

Vectors - character vectors

```
gender = c("F","M","F","M","F","M","F","M")  # Get their gender

smoker = c("N","N","Y","Y","Y","N","N","N")  # Do they smoke ?

table(gender,smoker) # Let's count them

      smoker
gender N Y
     F 2 2
     M 3 1

prop.table(table(gender,smoker))
      smoker
gender     N     Y
     F 0.250 0.250
     M 0.375 0.125

library(lattice)

barchart(table(gender,smoker),auto.key=TRUE,main="Smoking M/F")
```

Vectors - character vectors



Vectors - length and recycling

An important attribute of a vector is its length. To determine its length (or set it) one uses the “length” function.

```
y = 1:10

length(y)  # Length of the entire vector
[1] 10
```

When two vectors are combined in some fashion to form a third, the resulting vector takes on the length of the longest vector:

```
vec1 = 1:5

vec2 = c(1,3)

vec1 + vec2       # The shorter vector (vec2) is recycled
[1] 2 5 4 7 6

Warning message:
In vec1 + vec2 :
  longer object length is not a multiple of shorter object length
```

Vectors - naming elements

You can name the elements of the vector. Since we have been using only numeric data this might not add a lot of information. Though it actually does - especially when you work with character vectors. We use the “names” function for this.

Vectors - naming elements

You can name the elements of the vector. In this example, let’s say we have measured some heights of eight people.

```
height = c(59,70,66,72,62,66,60,60)

# Let's also create a character vector that contains the names of people
# whose heights we measured

my.names = c("Jacqueline","Frank","Babette",
             "Mario","Adriana","Esteban","Carole","Louis")

names(height) = my.names
height

Jacqueline    Frank     Babette      Mario    Adriana    Esteban     Carole      Louis
   59          70          66          72        62         66          60         60
```

Vectors - the which command

The which command allows us to determine which element number satisfies a given condition. If the element has a name then we can see it listed.

```
height > 60
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE

which(height > 60)
  Frank Babette   Mario Adriana Esteban
      2       3       4       5       6

height[which(height > 60)]
[1] 70 66 72 62 66

Note that the element names do not interfere with numeric evaluations

mean(height)
[1] 64.375
```

Vectors - naming elements

The "paste" function allows us to rapidly generate label names for observations in cases where we don't have original names or id.

```
new.names = paste("ID",1:8,sep="_")

new.names
 [1] "ID_1"  "ID_2"  "ID_3"  "ID_4"  "ID_5"  "ID_6"  "ID_7"  "ID_8"

names(height) = new.names

height
ID_1 ID_2 ID_3 ID_4 ID_5 ID_6 ID_7 ID_8
  59   70   66   72   62   66   60   60
```

Vectors - missing values

```
gender = c("F","M","F","M","F","M","F","M")  # Get their gender

smoker = c("N","N","Y","Y","Y","N","N","N")  # Do they smoke ?

length(gender)       # Gives current length of vector
[1] 8

length(gender) = 10  # Sets length of the vector

gender           # NA represents a missing value

 [1] "F" "M" "F" "M" "F" "M" "F" "M" NA  NA
```

Vectors - missing values

```
is.na(gender)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE

which(is.na(gender))   # Which elements contain missing values
[1]  9 10

which(!is.na(x))       # Which elements don't have missing values
[1] 1 2 3 4 5 6 7 8

gender[!is.na(gender)] # Get elements which aren't missing
[1] "F" "M" "F" "M" "F" "M" "F" "M"

gender[9:10] = "-" # Set all Nas to "-" but probably should leave NAs

[1] "F" "M" "F" "M" "F" "M" "F" "M" "-" "-"
```

Vectors - some common functions

`sum()`	`prod(x)`	Sum/product of vector elements
`cumsum(x)`	`cumprod(x)`	Cumulative sum/prod of elements
`min(x)`	`max(x)`	Returns min / max of vector
`mean(x)`	`median(x)`	Returns mean / median of x
`var(x)`	`sd(x)`	Gives variance / standard deviation
`cov(x,y)`	`cor(x,y)`	Covariance / Correlation of x,y
`range(x)`		Finds range of vector
`quantile(x)`		quantiles of the vector x
`fivenum(x)`		Returns fivenum value

`length(x)`	Returns number of elements in x
`unique(x)`	Returns only unique elements of x
`rev(x)`	Returns x reversed
`sort(x)`	sorts the vector x
`match(x)`	First position of an element in x
`union(x,y)`	Union of x and y
`intersect(x,y)`	Intersection of x and y
`setdiff(x,y)`	Elements of x not in y
`setequal(x,y)`	Test if x and y contain the same elements

Vectors - some common operators

Relational Operators

```
Equal to                    ==      if (myvar == “test”) {print(“EQ”)}
                            ==      if (mnynum == 3)     {print(“EQ”)}
Not equal to                !=      if (myvar != “test”) {print(“NE”)}
Less than or equal to       <=      if (number <= 5)     {print(“LTE”)}
Less than                   <       if (number < 10)     {print(“LT”)}
Greater than or equal to    >=      if (number >= 10)    {print(“GTE”)}
Greater than                >       if (number > 12)     {print(“GT”)}
```

Boolean Operators

```
And                         &       if ((myvar == “test”) & (num <= 10) ){
                                       print(“Equal and less than”)
                                    }

Not                         !       if (!complete.cases(myvec)) {
                                       print(“Non complete cases”)
                                     }

Or                          |       if ((num > 3) | (num < -3)) {
                                      print(“Only one of these has to be true”)
                                    }
```

Vectors - functions

```
mean(height)          # Get the mean
[1] 64.375

sd(height)       # Get standard deviation
[1] 4.897157

min(height)           # Get the minimum
[1] 59

range(height)         # Get the range
[1] 59 72

fivenum(height)       # Tukey's summary (minimum, lower-hinge, median,
                        upper-hinge, maximum)
[1] 59 60 64 68 72

length(height)        # Vector length
[1] 8

quantile(height)           # Quantiles
  0%  25%  50%  75% 100%
  59   60   64   67   72
```

Vectors - functions

```
my.vals = rnorm(10000,20,2)

max(my.vals)
[1] 28.94032

which.max(my.vals)
[1] 2570

my.vals[ which.max(my.vals) ]
[1] 28.94032

min(my.vals)
[1] 12.49251

my.vals[ which.min(my.vals) ]
[1] 12.49251

x = 1:16
x[x %% 2 == 0]                 # Find all the odd numbers from 1 to 16
[1]  2  4  6  8 10 12 14 16
```

Vectors - functions

Suppose we have x defined as follows. We want to find the sum of all the elements that are less than 5.

```
x = 0:10

x[ x < 5 ]
[1] 0 1 2 3 4

sum( x[x<5] )
[1] 10
```

Vectors - functions

Here is another vector. What if we wanted to compute the sum of the three largest values ? This would be easy by visual inspection but let's do it using some functions. This way we could use it on a vector that is possibly millions of elements long.

```
x = c(20,22,4,27,9,7,5,19,9,12)

sort(x)
 [1]  4  5  7  9  9 12 19 20 22 27

rev(sort(x))
 [1] 27 22 20 19 12  9  9  7  5  4

rev(sort(x))[1:3]
[1] 27 22 20

sum(rev(sort(x))[1:3])
[1] 69
```

Vectors - sample

The sample function takes a sample of a specified size from the elements of a given vector using either with or without replacement.

```
LETTERS       # A built-in character vector with the alphabet

[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R"
"S" "T" "U" "V" "W" "X" "Y" "Z"

sample(LETTERS, 26, replace=F)
[1] "Q" "J" "V" "I" "H" "A" "K" "W" "U" "E" "M" "D" "G" "O" "S" "Y" "L" "C"
"Z" "B" "N" "F" "X" "T" "P" "R"

sample(LETTERS, 26, replace=TRUE)
[1] "G" "V" "C" "M" "J" "B" "K" "Q" "M" "D" "V" "H" "D" "E" "C" "O" "B" "K"
"V" "Y" "S" "C" "S" "C" "N" "J"

sample(LETTERS,8,replace=FALSE)
[1] "S" "G" "U" "M" "F" "V" "O" "B"
```

Vectors - sample

```
my.coins = c("Heads","Tails")        # Create a coin vector

sample(my.coins,5,replace=TRUE)      # 5 coin tosses
[1] "Tails" "Tails" "Heads" "Tails" "Heads"


my.vec = sample(my.coins,100,replace=TRUE)

my.vec
[1] "Heads" "Tails" "Heads" "Heads" "Tails" "Heads" "Tails" "Tails" "Heads"
..
[100] "Tails"

table(my.vec)
my.vec
Heads Tails
   55    45

my.heads = my.vec[my.vec == "Heads"] # Gives us all the Heads

length(my.heads) / length(my.vec) * 100  # gives percentage of Heads
```

Vectors - sample

```
my.coins = c("Heads","Tails")     # Create a coin vector


# LET'S SIMULATE 1,000,000 TOSSES AND TABULATE

faircoin = table(sample(my.coins,1000000,replace=TRUE))

 Heads  Tails
500072 499928

# NOW LET'S CHEAT AND RIG THE COIN

unfaircoin = table(sample(my.coins,1000000,
                          replace=TRUE,prob=c(.75,.25)))

unfaircoin

 Heads  Tails
749811 250189
```

http://www.sigmafield.org/comment/21

Vectors - sample

```
# Does faircoin represent a fair coin ? Yes

chisq.test(faircoin, p=c(.5,.5))

    Chi-squared test for given probabilities

data:  faircoin
X-squared = 0.3069, df = 1, p-value = 0.5796


# Is unfaircoin "fair" ? Of course not

chisq.test(unfaircoin, p=c(.5,.5))

    Chi-squared test for given probabilities

data:  unfaircoin
X-squared = 249622.1, df = 1, p-value < 2.2e-16
```

Supplemental - Vectors - sample

```
# LET'S DO A SIMPLE BOOTSTRAP EXAMPLE


# Generate 1,000 values from a normal dist, mu=10

my.norm = rnorm(1000,10)

# Sample with replacement, collect means

mean(sample(my.norm,replace=TRUE))
[1] 10.01396

mean(sample(my.norm,replace=TRUE))
[1] 9.963395

..
..

mean(sample(my.norm,replace=TRUE))

Do this 1,000 times then do quantile of all the means according
to .95 confidence
```

Supplemental - Vectors - sample

```
# LET'S DO A SIMPLE BOOTSTRAP EXAMPLE

my.norm = rnorm(1000,10)  # Generate 1,000 values from a normal dist, mu=10

# NOW USE THE REPLICATE FUNCTION TO GENERATE 1,000 MEANS

quantile(replicate(1000, mean(sample(my.norm, replace = TRUE))),
+           probs = c(0.025, 0.975))
     2.5%     97.5%
 9.927472 10.044173

# COMPARE TO T.TEST

t.test(my.norm)$conf.int
[1]  9.923378 10.044916
```

http://www.sigmafield.org/comment/21

Vectors - characters

Let's look back at the character vectors:

```
char.vec = c("here","we","are","now","in","winter")

nchar(char.vec)
[1] 4 2 3 3 2 6

rev(char.vec)   # Reverses the vector
[1] "winter" "in"     "now"    "are"    "we"     "here"


char.vec[-1]   # Omit the first element
[1] "we"     "are"    "now"    "in"     "winter"

char.vec = c(char.vec,"Its cold")   # Append the vector

[1] "here"     "we"       "are"      "now"      "in"       "winter"   "Its cold"
```

Vectors - characters

R has support for string searching and manipulation. This is important for managing sequencing data. Let's start with some basics.

```
char.vec = c("here","we","are","now","in","winter")

grep("ar",char.vec)
[1] 3

char.vec[3]
[1] "are"

grep("ar",char.vec,value=T)
[1] "are"

grep("^w",char.vec,value=TRUE) # Words that begin with “w”
[1] "we"     "winter"

grep("w",char.vec, value=TRUE)                 # Any words that contain w
[1] "we"     "now"    "winter"

grep("w$",char.vec, value=TRUE)              # words that end with “w”
[1] "now"
```

Vectors - characters

R has support for string searching and manipulation. This is important for managing sequencing data. Let's start with some basics.

```
char.vec = c("here","we","are","now","in","winter")

char.vec[ -grep("ar",char.vec)]  # All words NOT containing "ar"
[1] "here"   "we"     "now"    "in"     "winter"


-grep("ar",char.vec)
[1] -3

char.vec[-3]

gsub("here","there",char.vec)  # We can change words too !
[1] "there"  "we"     "are"    "now"    "in"     "winter"

gsub("^w","Z",char.vec) # Replace any "w" at the beginning of a word to Z

[1] "here"   "Ze"     "are"    "now"    "in"     "Zinter"
```

Vectors - DNA character strings

We can search within a character vector for some specific characters. Let's find all the occurrences of the "G" string:

```
dna = c("A","A","C","G","A","C","C","C","G","G","A","T","G","A","C","T","G",
"A","A","C")

dna
"A" "A" "C" "G" "A" "C" "C" "C" "G" "G" "A" "T" "G" "A" "C" "T" "G" "A" "A"
"C"

grep("G",dna)          # Extracts the elements numbers
[1]  4  9 10 13 17

dna[ grep("G",dna) ]
[1] "G" "G" "G" "G" "G"

OR MORE SIMPLY

grep("G",dna, value = TRUE)
[1] "G" "G" "G" "G" "G"

length(grep("G",dna, value = TRUE))  # 5 occurrences of G
[1] 5
```

Vectors - DNA character strings

We can search within a character vector for some specific characters. Let's find all the occurrences of the "C" string:

```
set.seed(188)

dna = sample(c("A","C","G","T"),20,T)

dna
 [1] "A" "A" "C" "G" "A" "C" "C" "C" "G" "G" "A" "T" "G" "A" "C" "T" "G" "A"
"A" "C"

grep("C",dna, value = TRUE)
[1] "C" "C" "C" "C" "C" "C"


length(grep("C",dna, value=T))
[1] 6
```

Vectors - DNA character strings

Let's look at some special cases.

```
dna = c("A","A","C","G","A","C","C","C","G","G","A","T","G","A","C","T","G",
"A","A","C")

dna
 [1] "A" "A" "C" "G" "A" "C" "C" "C" "G" "G" "A" "T" "G" "A" "C" "T" "G" "A" "A" "C"

my.str = paste(dna,collapse="")
[1] "AACGACCCGGATGACTGAAC"


length(my.str)
[1] 1                # Not what you expected ?

my.str
[1] "AACGACCCGGATGACTGAAC"

rev(my.str)                   # What's going on ?
[1] "AACGACCCGGATGACTGAAC"

str(my.str)                      # Its now just a character string not a vector
 chr "AACGACCCGGATGACTGAAC"
```

Vectors - character strings

Let's look at some special cases.

```
my.str = paste(dna,collapse="")
[1] "AACGACCCGGATGACTGAAC"


substr(my.str,1,1)
[1] "A"

substr(my.str,1,2)
[1] "AA"

substr(my.str,1,3)
[1] "AAC"

substr(my.str,1,4)
[1] "AACG"

gsub("TG","G",my.str)
[1] "AACGACCCGGAGACGAAC"
```

Vectors - character strings

Let's look at some special cases.

```
my.str
[1] "AACGACCCGGATGACTGAAC"

substr(my.str,2,8)
[1] "ACGACCC"

substr(my.str,2,8) = "TTTTTTT"

my.str
[1] "ATTTTTTTGGATGACTGAAC"
```

Supplemental - Vectors - character strings

Let's look at some special cases.

```
nchar(my.str)
[1] 20

for (ii in 1:nchar(my.str)) {
    cat(substr(my.str,ii,ii))
}
AACGACCCGGATGACTGAAC


for (ii in nchar(my.str):1) {
    cat(substr(my.str,ii,ii))
}
CAAGTCAGTAGGCCCAGCAA

# Recipe to get the "collapsed" string back into a vector with separate elements for
each letter

unlist(strsplit(my.str,""))
 [1] "A" "A" "C" "G" "A" "C" "C" "C" "G" "G" "A" "T" "G" "A" "C" "T" "G" "A" "A" "C"
```

Vectors - characters

<table>
<tr><th>POSIX</th><th>Non-standard</th><th>Perl</th><th>Vim</th><th>ASCII</th><th>Description</th></tr>
<tr><td>[:alnum:]</td><td></td><td></td><td></td><td>[A-Za-z0-9]</td><td>Alphanumeric characters</td></tr>
<tr><td></td><td>[:word:]</td><td>\w</td><td>\w</td><td>[A-Za-z0-9_]</td><td>Alphanumeric characters plus "_"</td></tr>
<tr><td></td><td></td><td>\W</td><td>\W</td><td>[^A-Za-z0-9_]</td><td>Non-word characters</td></tr>
<tr><td>[:alpha:]</td><td></td><td></td><td>\a</td><td>[A-Za-z]</td><td>Alphabetic characters</td></tr>
<tr><td>[:blank:]</td><td></td><td></td><td></td><td>[\t]</td><td>Space and tab</td></tr>
<tr><td></td><td></td><td>\b</td><td>\< \></td><td>(?<=\W)(?=\w)|(?<=\w)(?=\W)</td><td>Word boundaries</td></tr>
<tr><td>[:cntrl:]</td><td></td><td></td><td></td><td>[\x00-\x1F\x7F]</td><td>Control characters</td></tr>
<tr><td>[:digit:]</td><td></td><td>\d</td><td>\d</td><td>[0-9]</td><td>Digits</td></tr>
<tr><td></td><td></td><td>\D</td><td>\D</td><td>[^0-9]</td><td>Non-digits</td></tr>
<tr><td>[:graph:]</td><td></td><td></td><td></td><td>[\x21-\x7E]</td><td>Visible characters</td></tr>
<tr><td>[:lower:]</td><td></td><td></td><td>\l</td><td>[a-z]</td><td>Lowercase letters</td></tr>
<tr><td>[:print:]</td><td></td><td></td><td>\p</td><td>[\x20-\x7E]</td><td>Visible characters and the space character</td></tr>
<tr><td>[:punct:]</td><td></td><td></td><td></td><td>[\]\[!"#$%&'()*+,./:;<=>?@\^_`{|}~-]</td><td>Punctuation characters</td></tr>
<tr><td>[:space:]</td><td></td><td>\s</td><td>\s</td><td>[\t\r\n\v\f]</td><td>Whitespace characters</td></tr>
<tr><td></td><td></td><td>\S</td><td>\S</td><td>[^ \t\r\n\v\f]</td><td>Non-whitespace characters</td></tr>
<tr><td>[:upper:]</td><td></td><td></td><td>\u</td><td>[A-Z]</td><td>Uppercase letters</td></tr>
<tr><td>[:xdigit:]</td><td></td><td></td><td>\x</td><td>[A-Fa-f0-9]</td><td>Hexadecimal digits</td></tr>
</table>

Vectors - characters

Metacharacter	Meaning
?	The ? (question mark) matches the preceding character 0 or 1 times only, for example, colou?r will find both color (0 times) and colour (1 time).
*	The * (asterisk or star) matches the preceding character 0 or more times, for example, tre* will find tree (2 times) and tread (1 time) and trough (0 times).
+	The + (plus) matches the previous character 1 or more times, for example, tre+ will find tree (2 times) and tread (1 time) but NOT trough (0 times).
{n}	Matches the preceding character, or character range, n times exactly, for example, to find a local phone number we could use [0-9]{3}-[0-9]{4} which would find any number of the form 123-4567. **Note:** The - (dash) in this case, because it is outside the square brackets, is a **literal**. Value is enclosed in braces (curly brackets).
{n,m}	Matches the preceding character at least n times but not more than m times, for example, 'ba{2,3}b' will find 'baab' and 'baaab' but NOT 'bab' or 'baaaab'. Values are enclosed in braces (curly brackets).