NYC DATA SCIENCE ACADEMY

Introduction to R Part II

Data Science Bootcamp

OVERVIEW

- **More data types**
  - Characters
  - Factors
- **Control statements**
  - Conditionals
  - Loops
- **Functions**
- **Data Transformations**

Introduction to R, Part II

- In the last class, we covered the major R data structures: vectors, matrices, data frames, and lists.
- Today we will finish the discussion of data types, and then cover the classic programming features of conditionals and loops. The definition of functions is the last piece of the basic programming features of R, and we'll finish the class with a discussion of a few data transformation functions.
- The next class will be devoted to a study of a powerful package for data transformations called "dplyr".

OVERVIEW

- **More data types**
  - **Characters**
  - Factors
- **Control statements**
  - Conditionals
  - Loops
- **Functions**
- **Data Transformations**

Data Types

- We discussed the primitive data types of:
  - numerics, logicals, and characters (strings)
- Plus the structured types:
  - vectors, matrices, lists, and data frames.
- Today we will complete the survey of data types, with:
  - More on **characters**.
  - More on **factors**, the data type for representing categorical data.

Character Manipulation

- In this section, we'll discuss some basic functions for manipulating character data:
  - **`nchar()`**: Count the number of characters.
  - **`strsplit()`**: Split the elements of a character vector.
  - **`paste()`**: Concatenate strings.
  - **`substr()`**: Create substrings of a character vector.
  - **`gsub()`**: Replace parts of strings.
  - **`grep()`**: Complete pattern matching.

Character Functions: *nchar*

- Count the number of characters:

```
fruit = 'apple orange grape banana'
nchar(fruit)
[1] 25
```

- Why did we get this result? The spaces are treated as characters. Symbols are treated the same way:

```
test = '10+$ 0f $ymb01$!'
nchar(test)
[1] 16
```

Character Functions: *strsplit*

- Split the elements of a string into a list:

```
strsplit(fruit, split=' ')
[[1]]
[1] "apple"  "orange" "grape"  "banana"
```

- Creating a vector from the returned list:

```
fruitvec = strsplit(fruit, split=' ')[[1]]
print(fruitvec)

[1] "apple"  "orange" "grape"  "banana"
```

Character Functions: *paste*

- `paste()` concatenates vector elements pairwise with the usual recycling rules, separated by the `sep` argument (default `' '`).

```
paste(fruitvec, c('A', 'B'))
[1] "apple A"  "orange B" "grape A"  "banana B"
paste(fruitvec, c('A', 'B', 'C'), sep='::')
[1] "apple::A"  "orange::B" "grape::C"  "banana::A"
```

- If the `collapse` argument is provided, the resulting vector is concatenated into one string, with the `collapse` argument as separator:

```
paste(fruitvec, collapse=',')
[1] "apple,orange,grape,banana"
paste(fruitvec, c('A', 'B'), sep=':', collapse='@')
[1] "apple:A@orange:B@grape:A@banana:B"
```

Example: Formula Creation

- `paste` is often useful when you want to create objects programmatically, such as generating a formula or creating variables on the fly. `paste0()` is like `paste()`, except it has a default of `sep=""`. Here, `'x'` is recycled.

```
n = 1:8
xvar = paste0('x', n)
xvar
[1] "x1"  "x2"  "x3"  "x4"  "x5"  "x6"  "x7"  "x8"
```

Example: Formula Creation

- Now apply `paste()` again, with `collapse = ' + '`:

```
right = paste(xvar, collapse = ' + ')
right
[1] "x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8"
form = paste('y ~', right)
form
[1] "y ~ x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8"
```

Character Functions: *substr*

- Substrings of a character string:

```
fruit = 'apple orange grape banana'
substr(fruit, 1, 5)
[1] "apple"
```

- Like many functions, substr extends elementwise to vectors:

```
fruitvec
[1] "apple"  "orange" "grape"  "banana"
substr(fruitvec, 1, 3)
[1] "app" "ora" "gra" "ban"
```

Character Functions: *gsub*

- Replace parts of a string:

```
gsub('apple', 'strawberry', fruit)
[1] "strawberry orange grape banana"
```

```
gsub('a', '?', fruit)
[1] "?pple or?nge gr?pe b?n?n?"
```

```
gsub('an', 'HAA', fruit)
[1] "apple orHAAge grape bHAAHAAa"
```

Character Functions: *grep*

- Pattern matching; find strings that contain a certain pattern in a vector:

```
grep('grape', fruitvec)
[1] 3
```

```
grep('a', fruitvec)
[1] 1 2 3 4
```

```
grep('an', fruitvec)
[1] 2 4
```

Exercise 1:

- Consider the character "R is great! I love data!".
  - Find the substring of the sentence above from the character 6 to the character 10.
  - Which word position is the word "love"?
  - Replace "great" by "wonderful".

OVERVIEW

- **More data types**
  - Characters
  - Factors
- **Control statements**
  - Conditionals
  - Loops
- **Functions**
- **Data Transformations**

Factors

- Factors represent categorical values, a usually small number of discrete values that might be used as discriminators in statistical tests.

```
species = iris$Species
class(species)
[1] "factor"
species
  [1] setosa  setosa  setosa  setosa  setosa   ...
Levels: setosa versicolor virginica
```

- Factors may appear like numerics or characters, but they are actually of type “factor”. Neither numeric nor character operations can be applied.
- You can always change a factor back to its underlying type by using `as.character` or `as.numeric`.

Levels

- In a factor, the distinct values are called *levels*. Levels are always characters (even if the underlying data are numeric).

```
levels(species)
[1] "setosa"     "versicolor" "virginica"
class(levels(species))
[1] "character"
```

- In data frames, by default, *character columns are assumed to be factors* while numeric columns are not.
- Let's look at two functions that can change a numeric vector to a factor:
  - `factor`
  - `cut`

factor

- factor(v) makes vector v a factor; each distinct value becomes a level:

```
ranks = c(4, 1, 1, 4, 3, 3, 2, 3, 2, 4)
rank_factor = factor(ranks)
rank_factor
[1] 4 1 1 4 3 3 2 3 2 4
Levels: 1 2 3 4
levels(rank_factor)
[1] "1" "2" "3" "4"
```

- Remember, factors are not numbers, even if they look like they are:

```
rank_factor + 1
 [1] NA NA NA NA NA NA NA NA NA NA
Warning message: '+' not meaningful for factors
```

Set Levels of a Factor

- Levels can be renamed by assigning values directly to `levels`(*factor*).

```
vec1 = factor(rep(c(0,1,3), c(4,6,2)))
vec1
[1] 0 0 0 0 1 1 1 1 1 1 3 3
Levels: 0 1 3

levels(vec1) = c("male", "female", "male")
vec1
[1] male   male   male   male   female female
[7] female female female female male   male
Levels: male female
```

Reorder Levels of a Factor

- We sometimes want to change the order of the levels - e.g. their order affects how they will appear in bar graphs. This can be done with `ordered()`.

```
vec2 = factor(rep(c('b','a'), c(4,6)))
vec2
[1] b b b b a a a a a a
Levels: a b
levels(vec2)
[1] "a" "b"
ordered(vec2, levels=c('b','a'))
[1] b b b b a a a a a a
Levels: b < a
```

OVERVIEW

- **More data types**
  - Characters
  - Factors
- **Control statements**
  - Conditionals
  - Loops
- **Functions**
- **Data Transformations**

Conditionals

- Typically, R code is executed sequentially, one line at a time. To perform more complex tasks, we can execute code conditionally:

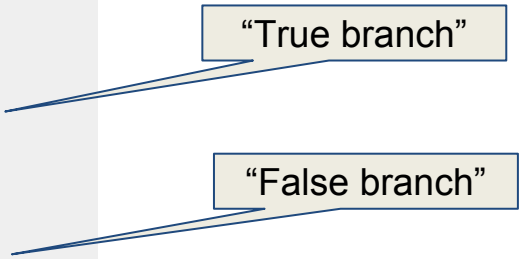
```
num = 5
if (num %% 2 != 0) {
  cat(num, 'is odd')
}
[1] 5 is odd
```

- The function `cat()` means "concatenate and print". The command inside "{}" would be executed only when the condition after **if** is satisfied. (Remember %% is remainder.)
- Note the indentation of the `cat()` statement. This is not required, but it helps make the code more readable: You can immediately see which part of the code depends upon the condition.

Conditionals

- A simple **if** is sufficient if we only need to check one condition. If our control flow is more complex, we must incorporate **else** statements for multiple branches.

```
num = 4
if (num %% 2 != 0) {
  cat(num, 'is odd')
} else {
  cat(num, 'is even')
}
4 is even
```



- *Syntax note*: The word "`else`" must appear on the same line as the closing brace of the previous branch; otherwise, R will think the `if` statement is finished and will not recognize the else on the following line.

Conditionals

- For more than two conditional branches, multiple if-else statements are necessary. We use **else if** in this case.

```
if (num %% 2 != 0) {
  cat(num, 'is odd')
} else if (num == 0) {
  cat(num, 'is even, although you may not realize it.')
} else {
  cat(num, 'is even')
}
4 is even
```

Conditionals

- If statements can be nested as much as you like:

```
num = 15
if (num %% 2 != 0) {
  if (num %% 5 == 0) {
    cat(num, 'is divisible by 5')
  } else {
    cat(num, 'is odd, but not divisible by 5')
  }
} else if (num == 0) {
  cat(num, 'is even, although you may not realize it.')
} else {
  cat(num, 'is even')
}
15 is divisible by 5
```

Execute if num is odd and divisible by 5

Execute if num is odd and *not* divisible by 5

Execute if num is even and equal to zero

Execute if num is even and *not* equal to zero

Conditionals

- If the true or false branch of an if has only one statement, it doesn't need the set braces. (Be careful: If there is an else statement, it still has to go on the same line as the previous branch.)

```
if (num %% 2 != 0)
  cat(num, 'is odd') else
  cat(num, 'is even')
15 is odd

if (num %% 2 != 0)
  cat(num, 'is odd')
else
  cat(num, 'is even')
Error: unexpected 'else' in "else"
15 is odd
15 is even
```

Aside: Vectorized Conditional

- The `if` statement is called a *control flow statement* because it affects the flow of control; it affects what statement is executed next. Since a computer can only execute one statement at a time, the condition in an `if` is a *single* logical condition.
- However, a single statement can operate on an entire vector at once - in effect, it can perform multiple operations. We have seen how a statement like "v3 = v1 + v2" can perform many additions, even though it is only a single statement.
- The `ifelse` function shown on the next slide does a kind of simultaneous conditional on an entire vector.

Vectorized Conditional

- `ifelse`(*lv*, *v1*, *v2*) takes a logical vector *lv* and vectors *v1* and *v2* and returns the vector that contains values from *v1* wherever *lv* contains TRUE, and values from *v2* where *lv* contains FALSE.

```
lv = c( T, T, F, T, F)
v1 = c( 1, 2, 3, 4, 5)
v2 = c(11,12,13,14,15)
ifelse(lv, v1, v2)
[1]  1  2 13  4 15
```

Vectorized Conditional

- For example, suppose we have two equal-length vectors and want a vector containing the larger of the values from each position:

```
v1 = c(1,2,3,4,5)
v2 = c(5,4,3,2,1)
ifelse(v1 > v2, v1, v2)
[1]  5 4 3 4 5
```

  - To break this down: "`v1 > v2`" is the usual vectorized version of `>`, yielding c(F, F, F, T, T).
  - Therefore, the result is the same as `ifelse(c(F, F, F, T, T), v1, v2)`: three values from `v2`, and two from `v1`.

Multiple Conditionals (ifelse)

- The result of an ifelse call has the same length as the first argument, and the other arguments are expanded or shrunken ("recycled") as necessary. For example, here, the second and third arguments are 1-element vectors that are expanded to six identical elements:

```
num = 1:6
ifelse(num %% 2 == 0, 'even', 'odd')
[1] "odd"  "even" "odd"  "even" "odd"  "even"
```

- That is, the first argument is the logical vector c(F,T,F,T,F,T), the second argument is extended to c('even', 'even', 'even', 'even', 'even', 'even'), and the third is extended similarly.

Multiple Conditionals (ifelse)

- ifelse calls can be nested to evaluate multiple conditions.

```
set.seed(1) #To ensure reproducible results.
age = sample(0:100, 20, replace=TRUE)
res = ifelse(age > 70, 'old', ifelse(age <= 30,
                                     'young', 'middle'))
```

  - age is a random sample of 20 ages.
  - `ifelse(age <= 30, 'young','middle')` recycles young and `middle` to give a vector of length 20 containing those two strings; ages above 70 become `middle` at this point.
  - The outer `ifelse` replaces the occurrences of `middle` by `old` for ages over 70. The result is a vector containing young for age <= 30, `middle` for age between 31 and 70, and `old` for age > 70.

Multiple Conditionals (switch)

- If you have many conditions, you might want to consider the **switch()** function. It comes in two versions. If its first argument is an integer, it just returns the corresponding following argument:

```
switch(1, 'Red', 'Orange', 'Blue')
 [1] "Red"
switch(2, 'Red', 'Orange', 'Blue')
 [1] "Orange"
switch(3, 'Red', 'Orange', 'Blue')
 [1] "Blue"
```

  - `switch(`i`,` e_1`,` e_2`, ...)` is *almost* the same as `c(`e_1`,` e_2`, ...)[`i`]`, but with one important difference: The `switch` statement evaluates *only* e_i, whereas the `c(...)[`i`]` call evaluates *all* the expressions.

Multiple Conditionals (switch)

- When the first parameter to switch is a string, the function is called with keyword arguments. The first parameter is matched with one of the keyword arguments:

```
age_type = 'middle'
switch(age_type,
       young = age[age <= 30],
       middle = age[age <= 70 & age > 30],
       old = age[age > 70]
)
[1] 37 57 66 63 69 38 50 38
```

Exercise 3: ifelse, switch

- Use `ifelse()` to decide tax by income. If income is less than $5,000, the tax rate is 10%; otherwise, it is 20%. How much tax does one need to pay if he or she makes $4,600? What about $12,300?
- The vector `Grade=c(75,93,88,80,99,75,76,92)`
- records the grades of students. Use the `switch()` to select grades that deserve an 'A': greater than 90; 'B': 80 to 89; 'C': 70 to 79:

```
Grade=c(75,93,88,80,99,75,76,92)
Choice = 'A'
<your code>
[1] 93 99 92
```

OVERVIEW

- **More data types**
  - Characters
  - Factors
- **Control statements**
  - Conditionals
  - Loops
- **Functions**
- **Data Transformations**

Loops

- Loops are one of the most important constructs in computer programming. In R, you will not need them as much as in some other languages, but it is still important to know how they work.
- Nearly all programming languages have two types of loops:
  - **for loops**: Execute a series of statements several times; the repetition count is known at the start. *For example, you might execute a statement for each item in a vector.*
  - **while loops**: Execute a series of statements several times; the repetition count is determined by a condition that cannot be evaluated ahead of time. *For example, you might execute a statement until the user says to stop.*

for Loops

- A for loop executes a series of statements once for each value in a list of values (such as a vector). For example, print a message for each NA value in a vector:

```
sign_data = read.csv('TimesSquareSignage.csv', header=TRUE)
for (x in sign_data$Width) {
    if (is.na(x)) {
        cat('WARNING: Missing width\n')
    }
}
```

- Note the *nesting* of the body of the loop inside set braces, and the *indentation* used to indicate the dependence of a statement on a condition (in both the for and if statements).

for Loops

- In this loop, the body is executed exactly as many times as the length of `sign_data$Width`, i.e., as many times as there are rows in the data frame.
- On each execution of the body (each "iteration"), the variable x takes on the next value in the vector.
- Here is the output from that loop:

```
WARNING: Missing width
WARNING: Missing width
 ...
WARNING: Missing width
WARNING: Missing width
WARNING: Missing width
```

9 lines

for Loops

- It is very common to iterate over the indices of a vector instead of the values. Here, we print a message giving the *location* of each NA:

```
sign_data = read.csv('TimesSquareSignage.csv', header=TRUE)

obs = nrow(sign_data)
for (i in 1:obs) {   # iterate over vector c(1, 2, ..., obs)
    if (is.na(sign_data$Width[i])) {
        cat('WARNING: Missing width for sign no.', i, '\n')
    }
}
```

for Loops

- The is.na test is performed sequentially on: `sign_data$Width[1]`, `sign_data$Width[2]`, `sign_data$Width[3]`, etc.
- The output is:

```
WARNING: Missing width for sign no. 2
WARNING: Missing width for sign no. 11
WARNING: Missing width for sign no. 14
WARNING: Missing width for sign no. 22
...
```

- Again: In both of our for loops, we know *as soon as the loop starts* exactly how many iterations it will have (but not necessarily what the output will be).

while Loops

- A while loop also repeats the sequence of statements in the body of the loop, but it has a different way of determining when to stop: when a condition given at the start of the loop becomes false.
- The terminating condition is checked when the loop starts; if it is TRUE, the body is executed. Then, *the condition is checked again*; it continues in this way until the condition becomes FALSE.

```
while (terminating condition) {
    body
}
```

  - It is possible that the condition will never become FALSE; this is called an *infinite loop*. This is a pretty common error in computer programming.

while Loops

- Here is a while loop that does same computation we saw earlier:

```
i = 1
while (i <= obs) {
    if (is.na(sign_data$Width[i])) {
        cat('WARNING: Missing width for sign no.', i, '\n')
    }
    i = i + 1
}
```

- Unlike the for loop, we cannot be sure this loop will terminate just by looking at the loop header, i.e. just by looking at the condition: It all depends upon having the "`i = i + 1`" statement incrementing `i`; if we forgot that, `i` would never change and this would be an infinite loop.

while Loops

- A breakdown of the execution of this while loop:
  - Start by setting `i` to 1.
  - On each iteration, increment `i`, so it takes on values 1, 2, 3, etc.
  - Eventually, `i` becomes greater than obs, the condition is false, and the loop terminates.
- The output is the same as the previous for loop:

```
WARNING: Missing width for sign no. 2
WARNING: Missing width for sign no. 11
WARNING: Missing width for sign no. 14
WARNING: Missing width for sign no. 22
...
```

for and while Loops

- Why would we use a while loop? The for loop seems simpler and, more importantly, it creates no risk of an infinite loop.
- Remember, in cases where the number of iterations cannot be known ahead of time, while loops are needed instead of for loops.
- A for loop executes a number of times equal to the length of the vector in the loop header, and a while loop executes until the termination condition is false.
- What if we wanted the loop to terminate immediately at some point?

The break Statement

- The break statement causes a loop to terminate immediately whenever it is reached within a loop. Consider the following:

```
sum = 0
number = 1
while (TRUE) {          #This creates an infinite loop...
  if (sum > 10) {
    break               #...but the break statement saves us!
  }
  sum = sum + number
  number = number + 1
  print(paste("sum is:", sum, "number is:", number))
}

[1] "sum is: 1 number is: 2"
[1] "sum is: 3 number is: 3"
[1] "sum is: 6 number is: 4"
[1] "sum is: 10 number is: 5"
[1] "sum is: 15 number is: 6"
```

Loop Efficiency

- One of the main reasons to write loops is to iterate over a vector or other data structure. In R, you will need to write for and while loops much less often than in most languages because many operations can adapt to vectorization straightaway!
- For example, let's try to square every element in a vector:

```
s = c()
v = 1:5
for (i in 1:length(v)) {
   s[i] = v[i]^2
}
```

- But here is a much simpler way:

```
s = v^2
```

Loop Efficiency

- Furthermore, the vectorized operations are often much faster than using loops. Let's explore this a little further...
- We create the function `prime.checker` which determines whether a number x is prime. It tests if x is divisible by 2 or by any odd number less than or equal to its square root. If x cannot be divided by any of those numbers, it is prime:

```
prime.checker = function(x) {
    if (x %in% c(2, 3, 5, 7)) return(TRUE)
    if (x %% 2 == 0 | x == 1) return(FALSE)
    xsqrt = round(sqrt(x))
    xseq = seq(from = 3, to = xsqrt, by = 2)
    return(all(x %% xseq != 0))
}
```

Loop Efficiency

- We will try to find all the primes less than 10000. We remark on the following:
  - The notation **1e4** is scientific notation for 10000.
  - It is a common trick to initialize an empty vector (**x1 = c()** in this case), and append elements to it through the iterative process (adding a logical type into it in this case).
  - The function `system.time()` reports the time it takes to finish a given process.

Loop Efficiency

- The first attempt:

```
system.time({
    x1 = c()
    for (i in 1:1e4) {
        y = prime.checker(i)
        x1[i] = y
    }
})
user    system      elapsed
0.240    0.036      0.277
```

Loop Efficiency

- Instead of initializing **x1 = c()**, below we use **x2 = logical(1e4)**. The difference is that we specify the type and the length of the vector, so R does not need to keep adding to it, which will improve efficiency:

```
system.time({
    x2 = logical(1e4)
    for (i in 1:1e4) {
        y = prime.checker(i)
        x2[i] = y
    }
})
user    system      elapsed
0.192   0.001       0.194
```

Loop Efficiency

- For most higher level languages, functional programming is a good way to avoid loops and improve efficiency. We went through a family of "apply" functions in the last class, which take a function as an argument and applies it to every entry in a vector (a numeric vector from 1 to 10000 in this case):

```
system.time({
    x3 = sapply(1:1e4, prime.checker)
})
user    system     elapsed
0.173    0.001      0.174
```

When to Use Explicit Loops

- When loop iterations depend on previous iterations, it is more difficult to avoid the explicit loop structure. Here, we use an explicit loop to find the Fibonacci numbers below 1000.

```
i = 2
x = c(1, 1)
while (x[i] < 1e3) {
  x[i+1] = x[i-1] + x[i]
  i = i + 1
}
x = x[-i]
print(x)
[1]  1   1   2   3   5   8  13  21  34
[10] 55  89 144 233 377 610 987
```

Exercise 4: for Loops

- Write a for loop to sum the squares of the integers from 1 to 100.
- Write a for loop to sum the squares of the integers in a vector v.
- Write a nested for loop: For each element in a vector v of positive integers, calculate the sum of the square of the integers from 1 to that element:

```
v = c(2,3,4,5)
... your loop ...
5
14
30
55
```

Note: When printing multiple lines, either use `print` - but that only takes one argument - or use `cat`, in which case the last argument should be `'\n'` (the special character for a new line).

Exercise 5: while loops

- Write a while loop to sum the squares of the integers from 1 to 100..
- Take our loop for calculating square roots by Newton's method and add a cat function call to print, for each iteration, the number of the iteration and the value of r, starting with r = 100:

```
<your loop>
Iteration 1 r = 50
Iteration 2 r = 26
Iteration 3 r = 14.92308
Iteration 4 r = 10.81205
Iteration 5 r = 10.0305
```

Exercise 6:

- Write a loop to compute the weighted average of `1:50000` with weights `w = seq(50000, 1, -1)/(sum(1:50000))`. How much time does it take?
  - The weighted average of vector v (values) and w (weights) is the sum of the pairwise products of elements of v and w: $v_1w_1 + v_2w_2 + \ldots v_nw_n$.
- You can compute the weighted average more easily using pointwise vector multiplication and the `sum()` function; how much time does that take?

OVERVIEW

- **More data types**
  - Characters
  - Factors
- **Control statements**
  - Conditionals
  - Loops
- **Functions**
- **Data Transformations**

Defining Functions

- We have seen some examples of defining functions, but now we'll discuss how to do this in detail.
- Functions have the form:

```
name = function(arguments) {
    body
    return(expression)
}
```

  - *arguments*: Values passed to the function, on which it does its calculations. There can be several, separated by commas.
  - *body*: A series of zero or more statements.
  - *return*: Expression giving the value to be provided as output.

Defining Functions

- For instance, this function calculates the area of a circle from its radius:

```
calc_area = function(r) {
    area = pi * r^2
    return(area)
}
calc_area(4)
[1] 50.26548
```

- It can also be written in one line:

```
calc_area = function(r) {
    return(pi * r^2)
}
```

Defining Functions

- For more complicated functions we can introduce multiple arguments with different types.
- For example, suppose we want to construct a function that computes the standard deviation of a numeric vector. Of course, we need to provide the numeric vector as an argument to the function.
- Besides the vector, we may also specify if we want the *sample* standard deviation (divide the sum of squared errors by the sample size minus 1) or the *population* standard deviation (divide by the sample size). That can be specified as another (non-numeric) argument to our function.
- Let's take a look...

Defining Functions

```
SDcalc = function(x, type) {
    n = length(x); mu = mean(x)
    if (type == 'sample') {
        stdev = sqrt(sum((x-mu)^2)/(n-1))
    }
    if (type == 'population') {
        stdev = sqrt(sum((x-mu)^2)/(n))
    }
    return(stdev)
}
```

```
SDcalc(1:10, 'sample')
[1] 3.02765
SDcalc(1:10, 'population')
[1] 2.872281
```

Local Variables in Function Definitions

- You can define variables in the body of a function, as we did above. These variables are *local* - they exist only inside the function:

```
SDcalc(1:10, 'sample')
[1] 3.02765
mu
Error: object 'mu' not found
```

- If you had a variable defined with the same name outside of the function, that variable definition would be undisturbed:

```
mu = 200
SDcalc(1:10, 'sample')
[1] 3.02765
mu
[1] 200
```

Keyword Parameters

- In R, you can always give the arguments to a function by using their names instead of their positions. Here are some legal calls to SDcalc:

```
SDcalc(x = 1:10, type = 'sample')
[1] 3.02765
SDcalc(type = 'sample', x = 1:10)
[1] 3.02765
SDcalc(1:10, type = 'population')
[1] 2.872281
```

- This is not as helpful if we need to give all the arguments anyway; however, it becomes useful if there are *default parameters* in the function's definition...

Defining Functions

- R functions allow you to specify default parameters. Here, we specify **`type = 'sample'`** in the argument list. The user *can omit this argument* in a function call; if so, the function works as if `'sample'` were passed:

```
SDcalc = function(x, type = 'sample') {
    ... same as before ...
}
```

```
SDcalc(1:10)
[1] 3.02765
SDcalc(1:10, type='population')
[1] 2.872281
```

- It is not uncommon for functions to have many arguments, so default parameters are a real convenience.

Hidden Arguments

- Notice that we called the functions `mean()` and `sum()` inside `SDcalc()`. In the documentation, we can see `mean()` and `sum()` take some arguments we didn't provide. However, there may be cases where we want those arguments to be propagated from the SDcalc call to the calls of `mean()`.

Defining Functions

- As we mentioned, applying SDcalc to a vector with NA returns NA:

```
test = c(1:10, NA)
mean(test)
[1] NA
SDcalc(test, type='sample')
[1] NA
```

- If we change the function we wrote, we can fix this by passing the hidden argument, the na.rm parameter; it gets passed to `mean()` and `sum()`.

```
# if we change SDcalc, we can do:
SDcalc(test, type='sample', na.rm=TRUE)
[1] 2.872281
```

- How can we do this?

Defining Functions

- To facilitate the usage of those “hidden” arguments, R provides a special parameter '...', which gives us access to embedded function parameters.; the '...' in `mean()` and `sum()` indicates the new parameters should also be passed to those functions:

```
SDcalc = function(x, type = 'sample', ...) {

    n = length(x); mu = mean(x, ...)

    if (type == 'sample') {
        stdev = sqrt(sum((x-mu)^2, ...)/(n-1)) }
    if (type == 'population') {
        stdev = sqrt(sum((x-mu)^2, ...)/(n)) }

    return(stdev)
}
```

Creating Custom Operators

- Another nifty feature of R is that you can create your own binary operators by defining functions. For example, we can define a set operator to find the intersection of two sets:

```
a <- c('NPR', 'New York Times', 'MSNBC')
b <- c('Wall Street Journal', 'NPR', 'Fox News')

'%int%' = function(x, y) {
  intersect(x, y)
}
a %int% b
[1] "NPR"
```

Aside: A Quick Look at Recursion

- Recursion is a technique by which a function calls itself. An intuitive way to define a function that calculates a factorial - closely mimicking the mathematical definition - is the following:

```
Fac1 = function(n) {
    if (n == 0) {
       return(1)
    }
    return(n * Fac1(n-1))
}
Fac1(10)
[1] 3628800
```

- Recursion is an advanced technique. A full treatment is beyond today's scope - but it is still something you should be aware of.

OVERVIEW

- **More data types**
  - Characters
  - Factors
- **Control statements**
  - Conditionals
  - Loops
- **Functions**
- **Data Transformations**

Data Transformations: Split

- "Data transformation" is a very general term that could be used to reference many operations on data. In data science and engineering, it generally refers to non-numerical operations that rearrange and combine data.
- We will see many data transformations in R. In a future class, we will look at `dplyr`, an entire package devoted to transformations.
- While we will learn how to combine data frames using `dplyr`, here we introduce a widely used functions for splitting data frames apart.

Split into Groups

- The `split()` function divides the data into groups defined by the factor specified in the second argument:

```
iris_split = split(iris, iris$Species)
class(iris_split)
[1] "list"

attributes(iris_split)
$names
[1] "setosa"    "versicolor"    "virginica"

str(iris_split) #Output on next slide...
```

Split into Groups

```
List of 3
 $ setosa    :'data.frame': 50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
  ..$ Sepal.Width : num [1:50] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
  ..$ Petal.Length: num [1:50] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
  ..$ Petal.Width : num [1:50] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ versicolor:'data.frame': 50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 7 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 ...
  ..$ Sepal.Width : num [1:50] 3.2 3.2 3.1 2.3 2.8 2.8 3.3 2.4 2.9 2.7 ...
  ..$ Petal.Length: num [1:50] 4.7 4.5 4.9 4 4.6 4.5 4.7 3.3 4.6 3.9 ...
  ..$ Petal.Width : num [1:50] 1.4 1.5 1.5 1.3 1.5 1.3 1.6 1 1.3 1.4 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 2 2 2 2 2 2 ...
 $ virginica :'data.frame': 50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 ...
  ..$ Sepal.Width : num [1:50] 3.3 2.7 3 2.9 3 3 2.5 2.9 2.5 3.6 ...
  ..$ Petal.Length: num [1:50] 6 5.1 5.9 5.6 5.8 6.6 4.5 6.3 5.8 6.1 ...
  ..$ Petal.Width : num [1:50] 2.5 1.9 2.1 1.8 2.2 2.1 1.7 1.8 1.8 2.5 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 3 3 3 3 3 3 3 3 3 3 ...
```

Split Example

- We can use split to calculate statistics within groups. We calculate averages for Petal.Length within species:

```
avg_petallen = function(x) {
    mean(x$Petal.Length)
}

lapply(iris_split, avg_petallen)
$setosa
[1] 1.462

$versicolor
[1] 4.26

$virginica
[1] 5.552
```

Unsplit

- The function `unsplit()` reverses the split operation:

```
iris_unsplit <- unsplit(iris_split, iris$Species)
class(iris_unsplit)
```

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

```
iris_unsplit[c(1, 51, 101), ]
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1            5.1         3.5          1.4         0.2    setosa
51           7.0         3.2          4.7         1.4    versicolor
101          6.3         3.3          6.0         2.5    virginica
```

- How does this work…?

Unsplit

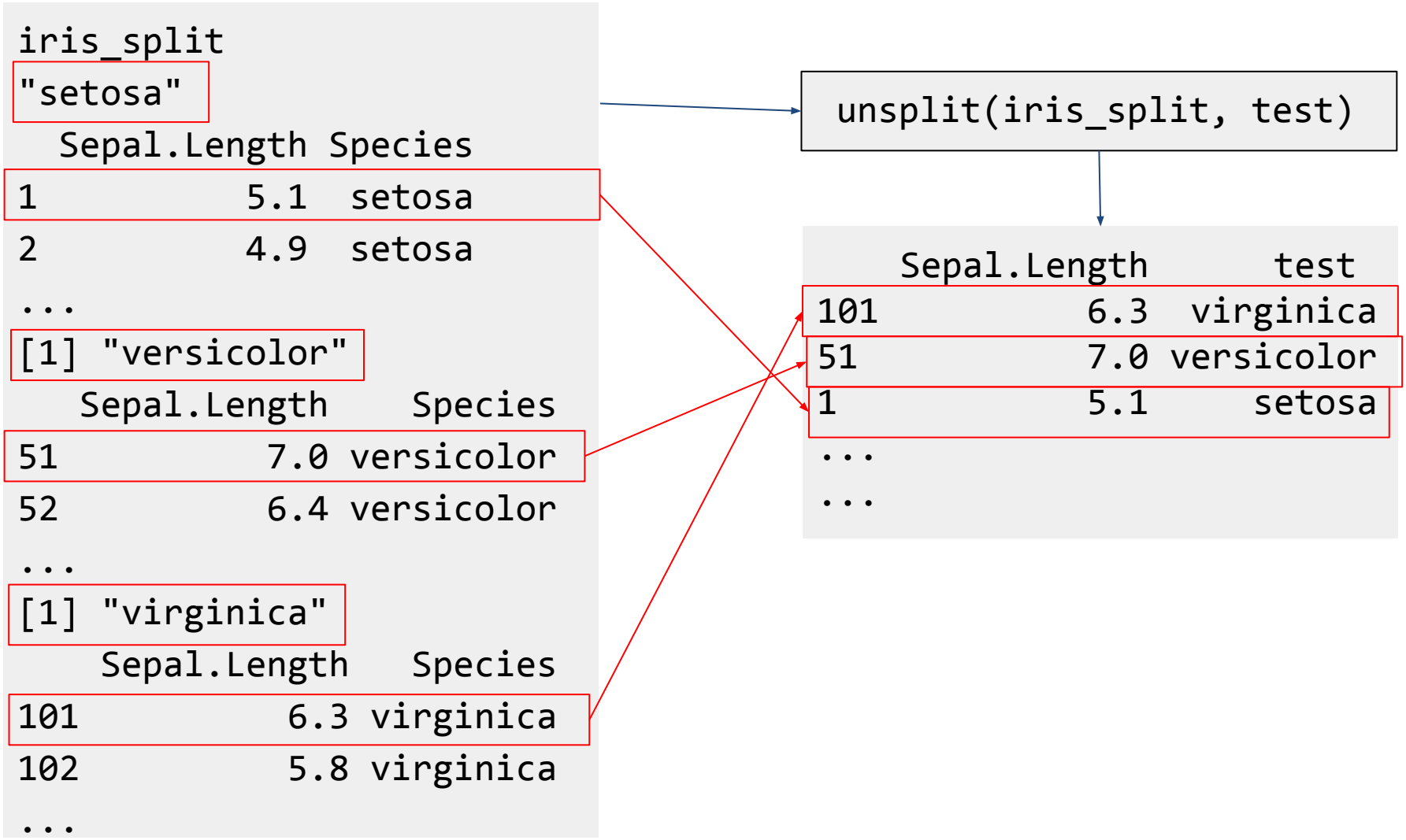
- To see how the function `unsplit()` works, let's try changing the second argument and see what happens:

```
test = rep(c('virginica', 'versicolor', 'setosa'),50)
head(unsplit(iris_split, test))
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
101          6.3         3.3          6.0         2.5  virginica
51           7.0         3.2          4.7         1.4 versicolor
1            5.1         3.5          1.4         0.2     setosa
102          5.8         2.7          5.1         1.9  virginica
52           6.4         3.2          4.5         1.5 versicolor
2            4.9         3.0          1.4         0.2     setosa
```

- The function `unsplit()` matches the list name with each element of the `test` vector.

Unsplit


iris_split
"setosa"
Sepal.Length Species
1 5.1 setosa
2 4.9 setosa
...
[1] "versicolor"
Sepal.Length Species
51 7.0 versicolor
52 6.4 versicolor
...
[1] "virginica"
Sepal.Length Species
101 6.3 virginica
102 5.8 virginica
...
unsplit(iris_split, test)
Sepal.Length test
101 6.3 virginica
51 7.0 versicolor
1 5.1 setosa
...
...

Exercise 8: split and unsplit

- Split the iris data set by Species.
- For each species, categorize the Petal Length into three levels, long, short and middle as we did before. Add this categorical feature as a new column in each data frame in the list (we have seen that the list obtained from the `split` function has data frames in its entries).
- Use `unsplit()` to turn it back into a data frame.

Appendix

- Appendix of concepts, exercises, and solutions

Exercise 2: if Statement

- A year Y is a leap year if it is divisible by 4, but not by 100; however, if Y is divisible by 400, it is a leap year. Write an if statement to print “Y is a leap year” or “Y is not a leap year”. Write it in two ways:
  1. There are three conditions to test: Y divisible by 4; Y divisible by 100; Y divisible by 400. The first method is to use each of these as the condition of an if, properly nesting the if’s to get the correct result.
  2. The second method is to combine the conditions using boolean operations & and |, and have a single if with a complicated condition.

Exercise 7: Function definitions

- Write a function, `tax_calc`, which takes *income* as an argument and computes the tax according the rule: the tax rate for the first $50,000 is 10%, and everything above $50,000 is taxed at 20%

```
tax_calc(60000)
[1] 7000
```

- You can calculate factorials without recursion. Define Fac2(i) to do this. There are numerous ways to do it, so we leave that up to you.
- Define an operator %~% between two data frames, which returns `TRUE` if the data frames have the same number of rows and columns, otherwise `FALSE`. (Hint: You can get the dimensions of a data frame with function `dim`; you might also check out function `all`.)

Aside: Approximating Square Roots

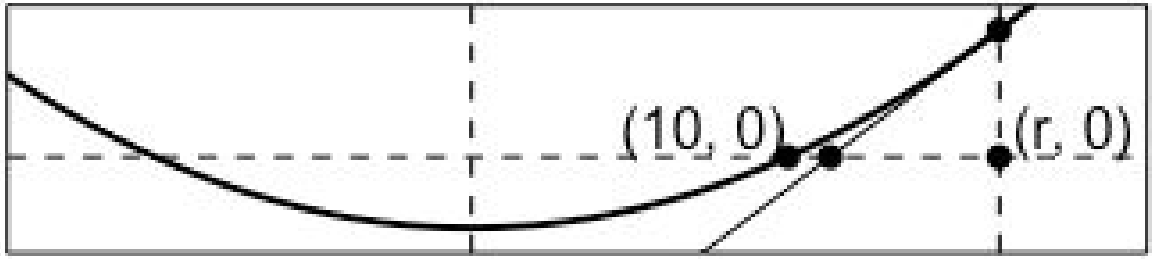
- This calculation is a famous example of an *iterative process*: one that continues until a value *close enough* to a desired answer is reached.
- This is *Newton's method* for square roots: Given a positive number x: for any value r, (r + x/r)/2 is closer to the square_root(x) than r is. E.g. suppose x = 100 and r = **50**: (r + x/r)/2 = **26**. With r = 26, (r + x/r)/2 = **14.9**. With r = 14.9, (r + x/r)/2 = **10.81**.)
- This loop keeps performing that calculation until r is as close as we want:

```
r = x/2
while (abs(x - r^2) > .001)
   r = (r + x/r)/2
```

- This *cannot be done in a for loop*, because we don't know how many iterations we'll need beforehand.

Aside: Newton's method

- For the curious, here's why this works: Look at the curve of $y = x^2 - 100$, and draw the tangent at a point r:



- As you can see, wherever we start, the tangent intersects the x-axis at a point closer to the square root of 100. The value (r + 100/r)/2 is actually r - f(r)/f′(r), where f′(r) is the slope of the tangent. This calculation can be shown to give the point of intersection of the tangent line with the x-axis.
- For more on Newton's method, see the wikipedia page (among many other sources).

Exercise 1 Solution

- Consider the character "R is great! I love data!".

```
vec="R is great! I love data!"
substr(vec,6,10)
[1] "great"
```

```
grep('love', unlist(strsplit(vec, split=' ')))
[1] 5
```

```
gsub('great', 'wonderful', vec)
[1] "R is wonderful! I love data!"
```

Exercise 2 Solution

```
if (Y %% 400 == 0) {
  cat(Y, 'is a leap year')
} else if (Y %% 100 == 0) {
  cat(Y, 'is not a leap year')
} else if (Y %% 4 == 0) {
  cat(Y, 'is a leap year')
} else
  cat(Y, 'is not a leap year')
```

```
if (Y %% 4 == 0 & (Y %% 100 != 0 | Y %% 400 == 0))
  cat(Y, 'is a leap year') else
  cat(Y, 'is not a leap year')
```

Exercise 3 Solution

```
num = 4600
ifelse(num<5000, num*0.1, num*0.2)
num = 12300
ifelse(num<5000, num*0.1, num*0.2)
```

```
Grade=c(75,93,88,80,99,75,76,92)
Choice = 'A'
switch(Choice, A=Grade[Grade >=90],
               B=Grade[Grade>=80 & Grade<90],
               C=Grade[Grade>=70 & Grade<80])
[1] 93 99 92
```

Exercise 4 Solution

```
sum=0
for (i in 1:100) {
   sum = sum + i^2
}
sum
[1] 338350
```

```
sum=0
for (x in v)
   sum = sum + x^2
```

```
for (x in v) {
  sum = 0
  for (i in 1:x) {
    sum = sum + i^2
  }
  print(sum)
}
```

Exercise 5 Solution

```
sum=0; i=1
while (i<=100) { sum = sum+i^2; i = i+1 }
sum
[1] 338350
```

- The stopping rule of the while loop depends on sum, which is a product of the loop. You can stop a while loop when its product meets some criteria, but it's not easy to do that for a for loop.

Exercise 6 Solution

```
w=seq(50000, 1, -1)/(sum(1:50000))
value=1:50000
weighted_mean=0
system.time({
for(i in 1:50000){
    weighted_mean = weighted_mean+w[i]*value[i]
  }
})
 user  system elapsed
0.035   0.003   0.038
```

```
system.time(sum(value*w))
 user  system elapsed
0.001   0.000   0.000
```

Exercise 7 Solution

```
tax_calc <- function(income){
  below_50K = min(income, 50000)
  above_50K = max(0, income-50000)
  return(below_50K * .10 + above_50K * .20)
}
```

```
Fac2 = function(i) {
  return(prod(1:i))
}
Fac2 = function(i) {
  fac = 1
  while (i > 0) {
    fac = fac * i
    i = i-1
  }
  return(fac)
}
'%~%' = function(d1, d2) {
   return(all(dim(d1) == dim(d2)))
}
```

Exercise 8 Solution

```
iris_list=split(iris, f=iris$Species)
```

```
# labels=c('Short','Middle','Long') # for cut soln
for (i in 1:3) {
  tmp = iris_list[[i]]$Petal.Length
  range=quantile(tmp, c(0,0.25,0.75,1))
  iris_list[[i]]$category = ifelse(tmp < range[2],
                                   'Short',
                                ifelse(tmp < range[3],
                                    'Long', 'Middle'))
  # iris$category=cut(tmp,breaks=range,  #using cut
  #                 labels=labels, include.lowest=TRUE)
}
```

```
unsplit(iris_list, iris$Species)
```