Computational Sociology

Programming fundamentals

Dr. Thomas Davidson

Rutgers University

February 1, 2021

Plan

- Course updates
- Recap on data structures
- Programming fundamentals
  - Boolean logic
  - If-else statements
  - Loops
  - Functions
  - Pipes
- Github

Course updates

Homework

- Homework 1 will be released by 5pm tomorrow.
  - Due next Tuesday (2/9) at 5pm Eastern.
  - Homework will be released and submitted using Github (more at the end of lecture)
- Please complete Google Form on Slack to share your Github credentials

Course updates

Homework

- Three more homework assignments
  - Web-scraping and APIS (due 2/23)
  - Natural language processing (due 3/30)
  - Machine learning (4/20)
- Homework 5 (agent-based modeling) removed from syllabus

Course updates

Final project information

- New timeline for final projects (60%)
  - Phase 1: Paper proposal (5%) (due 3/2)
  - Phase 2: Preliminary data collection (5%) (due 3/23)
  - Phase 3: Preliminary analysis (5%) (due 4/13)
  - Phase 4: Presentations (5%) (In class on 5/3)
  - Final submission (40%) (due 5/12)

Course updates

Final project information

- Use the next few weeks to think about project ideas and potential data sources
- Attend office hours or schedule a meeting with me to discuss your idea before submission of the proposal

Recap

Data structures

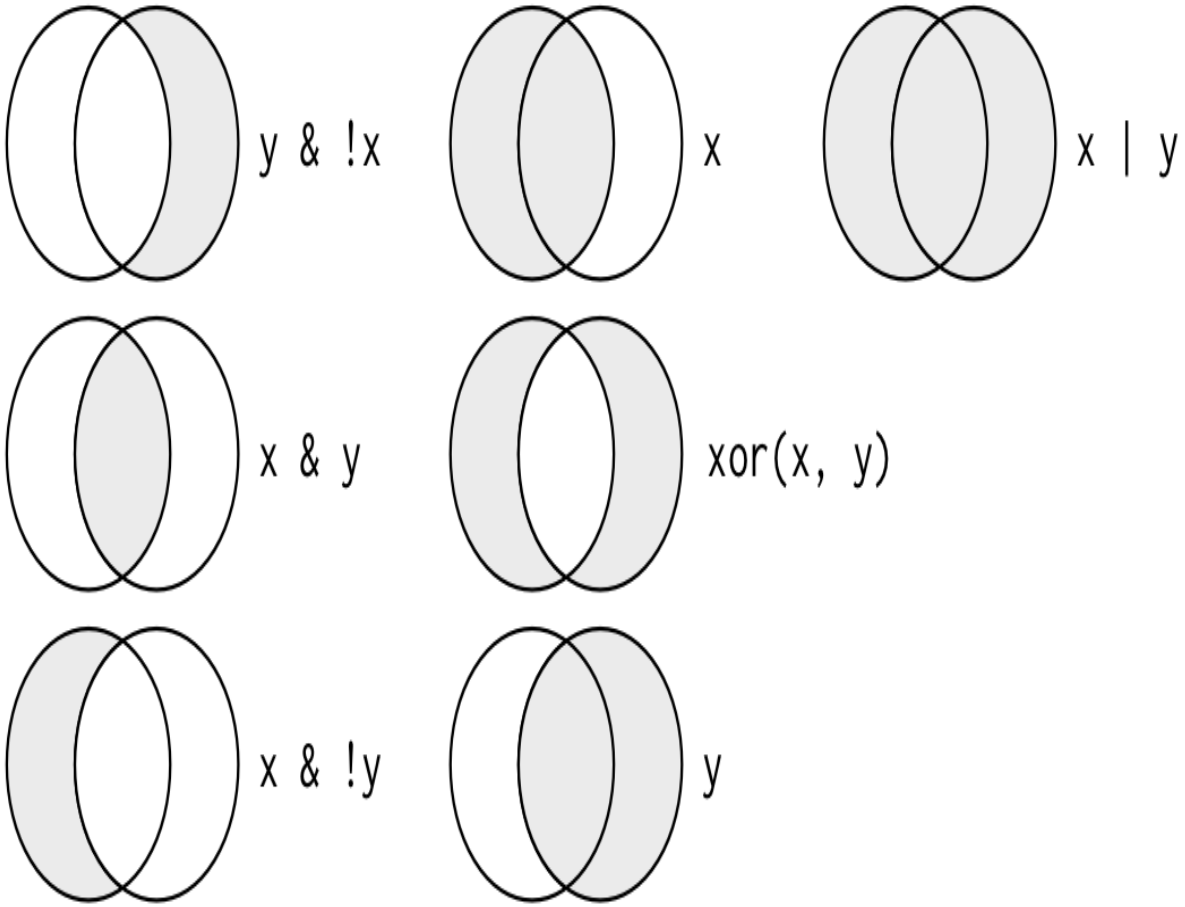
- Basic data types
- Vectors
- Lists
- Matrices
- Data frames and tibbles

Boolean logic

Operators

- There are three possible operators in Boolean logic
  - AND (`&`)
  - OR (`|`)
  - NOT (`!`)
- TRUE and FALSE are Boolean values, hence the type `logical`

Boolean logic



Boolean logic

```
TRUE == TRUE
```

```
## [1] TRUE
```

```
TRUE != FALSE
```

```
## [1] TRUE
```

```
TRUE == !FALSE
```

```
## [1] TRUE
```

```
!TRUE != !FALSE
```

```
## [1] TRUE
```

Boolean logic

```
TRUE | FALSE
```

```
## [1] TRUE
```

```
TRUE & FALSE
```

```
## [1] FALSE
```

```
TRUE & FALSE == FALSE
```

```
## [1] TRUE
```

Boolean logic

```
TRUE | FALSE & FALSE
```

```
## [1] TRUE
```

```
FALSE | TRUE & FALSE
```

```
## [1] FALSE
```

See https://stat.ethz.ch/R-manual/R-patched/library/base/html/Logic.html for more on logic in R.

Boolean logic

Uses

- These logical operations can be applied to Boolean values or used more generally.
  - Conditionals and functions
  - Selecting from or filtering a table
  - Querying a database

If-else statements

- We often encounter situations where we want to make a choice contingent upon the value of some information received.
- If-else statements allow us to chain together one or more conditional actions.
  - e.g. If time is between 4:10-6:50pm, attend computational sociology class meeting. Else, do something else.

If-else statements

The basic syntax. The `if` is followed by a conditional statement in parentheses. If the condition is met, then the code in the braces is executed.

```
x <- TRUE

if (x == TRUE) {
    print("x is true")
}
```

```
## [1] "x is true"
```

If-else statements

In this case we have a vector containing five fruits. We use sample to randomly pick one. We can use an if-statement to determine whether we have selected an apple.

```
fruits <- c("apple", "apple", "orange", "orange", "apple")

f <- sample(fruits, 1)

if (f == "apple") {
    print("We selected an apple")
}
```

```
## [1] "We selected an apple"
```

If-else statements

In the previous example we only have an if-statement. If the condition is not met then nothing happens. Here we add an else statement.

```
fruits <- c("apple", "apple", "orange", "orange", "apple")

f <- sample(fruits, 1)

if (f == "apple") {
    print("We selected an apple")
} else {
    print("We selected an orange")
}
```

```
## [1] "We selected an apple"
```

Note that R can be quite fussy about the syntax. If `else` is on the line below then the function throws an error.

If-else statements

What about this case where we have another fruit? If we only care about apples we could modify the output of our else condition.

```
fruits <- c("apple", "apple", "orange", "orange", "apple", "pineapple")

f <- sample(fruits, 1)

if (f == "apple") {
    print("We selected an apple")
} else {
    print("We selected another fruit.")
}
```

```
## [1] "We selected another fruit."
```

If-else statements

We could also use else-if statements to have a separate consideration of all three.

```
f <- sample(fruits, 1)

if (f == "apple") {
    print("We selected an apple")
} else if (f == "orange") {
    print("We selected an orange")
} else if (f == "pineapple") {
    print("We selected a pineapple")
}
```

```
## [1] "We selected an apple"
```

Loops

- Often when we program we need to complete the same operation many times. One of the common approaches is to use a loop.
- There are three main kinds of loops
  - For-loops
    - Iterative over an entire sequence
  - While-loops
    - Iterate over a sequence while a condition is met
- For-loops are more common, but there are cases where while-loops are necessary.

For-loops

Here is a simple example where we use a loop to calculate the sum of a sequence of values.

```
s <- 0L  # value to store our sum

for (i in 1:10000) {
    # for i from 1 to 100
    s <- s + i  # add i to sum
}

print(s)
```

```
## [1] 50005000
```

For-loops

```
s = 0

for i in range(1,10001):
  s += i

print(s)
```

```
## 50005000
```

The syntax varies slightly across programming languages but the basic structure is very similar, as this Python example shows. Note that for-loops and other functions in R tend to use braces around the operations. We will see this again when we look at functions.

For-loops

```
is_orange <- logical(length(fruits))  # a vector of logical objects

i = 1  # In this case we need to maintain a counter
for (f in fruits) {
    if (f == "orange") {
        is_orange[i] <- TRUE
    }
    i <- i + 1  # increment counter by 1
}

print(is_orange)
```

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

Often we may want to modify something based upon the the contents of whatever we are iterating over and the position. For example, let's say we want to create a vector representing whether each fruit in our fruits vector is an orange.

For-loops

Loops can also easily be nested. Here we start a second loop within the first one and use it to populate a matrix.

```
M <- matrix(nrow = 5, ncol = 5)

for (i in 1:5) {
    for (j in 1:5) {
        M[i, j] <- i * j
    }
}

print(M)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
## [4,]    4    8   12   16   20
## [5,]    5   10   15   20   25
```

While-loops

A while loop ends when a condition is met. Make sure the condition will eventually be false to avoid an infinite loop!

```
i = 1  # iterator
while (i < 5) {
    print(i)
    i <- i + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

While-loops

In this example we iterate over fruits until we get a pineapple.

```
i = 1  # iterator
f <- fruits[i]  # define initial value
while (f != "pineapple") {
    print(f)
    i <- i + 1  # increment index
    f <- fruits[i]  # get next f
}
```

```
## [1] "apple"
## [1] "apple"
## [1] "orange"
## [1] "orange"
## [1] "apple"
```

Functions

- A function is a customized sequence of operations
- We use functions to make our code modular and extendable
- There are thousands of functions built into R and available for packages, but sometimes it is useful to create our own
  - *R4DS* contains the following heuristic:
    - "You should consider writing a function whenever you've copied and pasted a block of code more than twice"

Functions

Here is an example of a simple function that returns the mean of a vector of values.

```
avg <- function(values) {
    return(sum(values)/length(values))
}

avg(c(5, 6, 6, 4, 3))
```

```
## [1] 4.8
```

We define the function by using the `function` command and assigning it to the name `avg`. The content in the parentheses is called the `argument` of the function. The `return` statement tells the function what output to produce.

Functions

Here is the same function in Python.

```
def avg(values):
  return(sum(values)/len(values))

avg([5,6,6,4,3])
```

```
## 4.8
```

Again you can see that the syntax is slightly different, for example the `def` command is used to define a function on the left hand side, followed by the name.

Functions

Testing

- It is important to test functions to ensure they work as expected
  - Ensure the function will only process valid inputs
  - It is good practice to handle incorrect inputs
    - There are many ways a function could behave that would not raise errors in R but could still be problematic
  - Write unit tests to ensure the function works as expected
    - Make sure to handle "edge cases", inputs that require special handling

Functions

Testing

```
avg(c("a", "b", "c"))
```

```
## Error in sum(values): invalid 'type' (character) of argument
```

Functions

Testing

```
avg(c())
```

```
## [1] NaN
```

Functions

Testing

The function can be modified to return a message if input is incorrect. Note the use of two `return` statements within the function.

```
avg <- function(values) {
    if (!is.numeric(values)) {
        return("Input must be numeric.")
    } else {
        return(sum(values)/length(values))
    }
}
```

Functions

Testing

```
# Unit tests
avg(c("a", "b", "c"))
```

```
## [1] "Input must be numeric."
```

```
avg(c())
```

```
## [1] "Input must be numeric."
```

```
avg(c(2.6, 2.4))
```

```
## [1] 2.5
```

Functions

Testing

While the return statement did avoid errors, we still have to be careful. The error message we printed could cause downstream errors if not handled. In this case we prevent an error by rejecting invalid inputs. See `help("tryCatch")` for more on error handling.

```
output <- avg(c("a", "b", "c"))
output/2  # we are attempting to divide a string
```

```
## Error in output/2: non-numeric argument to binary operator
```

Pipes

- Pipes are a tool designed to allow you to chain together a sequence of operations
- The pipe is designed to improve the readability of complex chains of function
- Implemented in the `magrittr` package but loaded in `tidyverse`

Pipes

Note how pipes allow us to chain operations from left to right, rather than nesting them from inner to outer. In this case we take a sequence from 1 to 10, get the square root of each value, sum the rots, then print the sum.

```
library(magrittr)

print(sum(sqrt(seq(1:10))))  # nested functions
```

```
## [1] 22.46828
```

```
seq(1:10) %>% sqrt() %>% sum() %>% print()  # using pipes
```

```
## [1] 22.46828
```

Pipes

We can also use pipes to do basic arithmetic using pipes. Note again the difference between the nested operations and the pipe operator.

```
((1 + 2) - 10) * 10
```

```
## [1] -70
```

```
1 %>% add(2) %>% subtract(10) %>% multiply_by(10)
```

```
## [1] -70
```

Note how magrittr provides aliases for certain mathematical operations as shown in the second line. https://stackoverflow.com/questions/27364390/chain-arithmetic-operators-in-dplyr-with-pipe#_=_

Pipes

Pipes are particularly useful we're working with tabular data. Here's an example without pipes or nesting. Each line produces an object that is then passed as input to the following line.

```
library(tidyverse)
library(nycflights13)

not_delayed <- filter(flights, !is.na(dep_delay), !is.na(arr_delay))
grouped <- group_by(not_delayed, year, month, day)
summary <- summarize(grouped, mean = mean(dep_delay))
print(summary)
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.4
##  2  2013     1     2 13.7
##  3  2013     1     3 10.9
##  4  2013     1     4  8.97
```

Pipes

In this case the expressions have been nested. This is better as we are not unnecessarily storing intermediate objects.

```
summarize(group_by(
  filter(flights, !is.na(dep_delay), !is.na(arr_delay)),
  year, month, day), mean = mean(dep_delay))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.4
##  2  2013     1     2 13.7
##  3  2013     1     3 10.9
##  4  2013     1     4  8.97
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.56
##  9  2013     1     9  2.30
```

Pipes

Here we have the same expression using a pipe. Note how much easier this is to read.

```
flights %>% filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  group_by(year, month, day) %>% summarize(mean = mean(dep_delay))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.4
##  2  2013     1     2 13.7
##  3  2013     1     3 10.9
##  4  2013     1     4  8.97
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.56
##  9  2013     1     9  2.30
## 10  2013     1    10  2.84
```

Working with tabular data

- Often we will be using intermediate objects like lists, vectors, and matrices to produce tabular data
  - We can then use tables to create visualizations and conduct statistical analysis
- The `tidyverse` contains an entire suite of functions designed for such tasks:
  - `dplyr` contains functions for basic manipulation and merges and joins
  - `tidyr` implements functions for data cleaning and shape transformations
  - `purr` contains methods to easily map functions to lists and data frames.
  - https://github.com/rstudio/cheatsheets is a great resource

Github

Overview

- Github is a version-control system
  - This allows you to easily control and manage changes to your code (similar to Track Changes in Word)
  - It can facilitate collaboration
  - Version-control helps to ensure reproducibility
  - It makes it easy to share code
- Github is *not* designed as a place to store large datasets (100Mb file size limit)

Github

Terminology

- A Github *repository* (or *repo* for short) contains all files and associated history
  - A repository can be public or private
  - Files should be organized into folders
  - Github can render Markdown files (suffix `.md` in Markdown), useful for documentation
- Github repositories exist online and you can *clone* them to your local computer

Github

Using Github

- You can interact with Github in several different ways
  - Github Desktop (recommended)
  - Through your browser (not recommended)
  - Using the command line
  - RStudio integration
    - See https://happygitwithr.com/index.html for a guide

Github

Basic commands

- Let's say you want to make changes to a repository, in this case adding a single file called `myfile.txt`:

1. Make changes to `myfile.txt` and save the file.
2. `git status` will show information about the status of your repo.
3. `git add myfile.txt` will stage the file to be added to the online repo.
   - Avoid using `git add *`
4. `git commit -m "Adding a new file"` commits the file to the repo, along with an informative message.

Github

Basic commands

5. `git push origin main` then tells Github to push the local changes to the main branch of the online repository
   - Conversely, `git pull origin main` will pull the latest updates from your main branch to your local machine
6. Now visit the web page for your repository and you should see the changes.

Github

Viewing commit histories

- You can view the history of a given file by looking at the commits
  - e.g. Let's look at the syllabus for this course https://github.com/t-davidson/computational-sociology/commits/main/syllabus/syllabus.Rmd

Github

Branches and merging

- A *branch* consists of a particular version of the repo
  - All repos start with a single branch called *main* (formerly *master*)
  - You can create separate branches for particular tasks
    - This is particularly useful for collaboration
  - You can then *merge* the branch back into main
    - But be careful of *merge conflicts*
- A *pull request* is a mechanism for merging content into a repository
  - This can enable the code to be reviewed before it is integrated
- The *issue* function can be used to note any issues with the code and to bring them to the repo owner's attention (e.g. https://github.com/tidyverse/ggplot2)

Github

Forks

- A *fork* is a copy of another repository (usually from another user)
  - This allows you to easily copy the repository and modify it without changing the original content

Github

Classroom

- We will be using a tool called *Github Classroom* for the homework assignments
  - You will receive a special template repository containing the homework
  - The submission will occur when you push the final commits to Github
    - Further instructions will be included

Github

Student Developer Pack

- If you haven't already, log in and apply for the Github Student Developer pack
  - https://education.github.com/pack
- This allows you to make unlimited private repos and gives access to many other tools

Questions?