Version Control

Julia Wrobel

Overview

Today, we cover:

- Debugging tools and techniques
- Git/GitHub
- Finish lab

Antibugging

Antibugging refers to strategies coders use to proactively prevent, identify, and address bugs during software development.

- **Defensive programming**: when writing a function, you can sometimes anticipate potential problems
  - Example: your function takes a file path and loads data, but a user might supply the path to a file that doesn't exist

Antibugging

Antibugging refers to strategies coders use to proactively prevent, identify, and address bugs during software development.

- **Defensive programming**: when writing a function, you can sometimes anticipate potential problems
    - Example: your function takes a file path and loads data, but a user might supply the path to a file that doesn't exist
    - Example: user passes in the wrong input type. If you know x should be positive:

```
stopifnot(x > 0)
```

If you insert this line of code in your function after assigning x and x is not positive, then your program will stop with the message:

```
Error: x > 0 is not TRUE
```

Antibugging: Conditions

Conditions communicate a problem to the user. There are three types:

- **Errors** are raised by `stop()` and terminate execution of the program
- **Warnings** arise from `warning()` and tell the user about potential problems
- **Messages** are generated by `message()` and provide information to the user

Debugging

Overview of debugging process:

- **Reproduce**: figure out how to reliably reproduce the problem
- **Hypothesize**: conjecture causes of the problem
- **Diagnose**: test each hypothesis until the problem is isolated
- **Fix**: design and apply a fix for the problem. The fix should not detract from the quality and readability of the code
- **Reflect**: Why did the mistake happen? Are there other places this mistake might have occurred?

Identifying which function gives an error - traceback()

```r
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
```

```r
f(10)
```

Whenever error occurs, you can hit traceback to see which function returned an error.

Debugging tools

- **Manual:** put print statements in different parts of your code or condition statements to check intermediate outputs
- Use **debug()** and **browser()** to step through your code

```
# Calculate value of function (x + y)/x
# for a given sequence of x and y
 f <- function(x, y){
   a <- x + y
   b <- a/x
   return(b)
 }
x = c(1, 2, 0); y = c(2, 3, 0)
```

```
f(x, y)
```

```
[1] 3.0 2.5 NaN
```

```
debug(f)
f(x, y)
```

Debugging tools

- More on debugging from Hadley Wickham, **Highly recommended read**
- Rstudio specific advice, **Highly recommended read**
- The best way to learn debugging is to practice on your own code

Debugging summary

- `print()`
  - Printing the value of variables and other signposts can help pinpoint where the program starts to fail
- `traceback()` (RStudio: error inspector):
  - Prints the **sequence of calls** leading to the last uncaught error
  - When code breaks in R, first read the error message, then type `traceback()` in your R session
- `browser()` (RStudio: breakpoints, “Rerun w/ Debug”):
  - `browser()` interrupts your program during execution and allows you interactively examine and change values of variables while you continue to execute your program step-by-step

Debugging tip

- Debugging is easier with **modular code**
  - Make sure each of your functions is not doing too many things
- **Less modular**: one function generates data **and** computes an estimator
- **More modular**: one function generates data; another takes data as input and computes the estimator

Other failures

- To investigate warnings that arise from your code:
  - Use `options(warn = 2)` to convert them to errors
  - Then you can trace their origin by calling `traceback()` to look at the call stack
- A function might get stuck and never return
  - Force it to terminate (e.g., control + C) and look at the call stack, or use other standard debugging strategies
- Code that crashes R completely indicates a bug in the underlying C code. This case is difficult to debug

Git/GitHub

Basic GitHub workflow

```
git pull

git status
git add --all
git commit -m "informative commit message!"
git status

git push
```

What is Git?

- Git is a version control system - lets you track your progress over time
- Check that you have GIT installed by typing in the terminal/bash

```
git --version
```

- See here for details on installing git and registering a GitHub account if you don't have these already

Why use (formal) version control?

- Ability to share code online
- Collaborate with multiple people on a single code bank
- Try risky things without fear of losing stable code
- Painlessly revert to older versions of code
- See history of changes made to code
- Expected skill in data science

Terminology

- **Repository**
    - Directory of all-(or most)- files included in your project
- **Commit**
    - a unique alpha-numeric identifier that references a particular state of your project
- **History**
    - the history of all commits for a project
- **Local**
    - the copy of the **repo** on your computer
- **Remote**
    - the copy of your repo online (e.g. on GitHub)

Initializing a repository

- To initialize a repository, we use the `git init` command
    - Make sure that you are in the correct directory first!
- This creates a subdirectory called `.git` that contains all files needed for version control
- Deleting this directory removes the entire history of a project

Basic git workflow

1. Create a project folder
2. Make sure there is an `.Rproj` file in the root directory
3. Open the terminal and `cd` into that directory

Execute the following `bash` code

```bash
# only once for each repo
git init

# used over and over
git status
git add --a # adds everything
git status
git commit -m "my highly informative commit message"
git status
```

When to commit

How often you make commits is totally up to you

- (And anyone you may be collaborating with)
- More harmful to commit too little vs. too much
- Commit code once it reaches a stable state
- OK to include longer commit messages if they will be helpful for you or your collaborators

What to commit

- Generally, want to include only plain text files (i.e., code) as part of your version-controlled repository
    - Commit source code and not outputs of code
- Generally, do not want to include binary files (e.g., pdf, jpeg, docx, .Rdata, .rds, .xlsx, etc...)
- No real harm in including, but can make git slower
- Be mindful of committing data. Is it allowed under DUA?

.gitignore files

- If there are files that you know for sure you do not want to track, include these files in a `.gitignore` file
- Create a new file and save it with name `.gitignore`
- Any files listed in `.gitignore` will be specifically ignored when using commands like `git status` or `git commit`.
- Use wildcards (e.g., `*.pdf`) to ignore files of a particular type found anywhere in the repository

Creating a GitHub repo

- From GitHub dashboard, click on the + symbol in the upper right-hand corner and select New repository
- Give the repository a name
- DO NOT check the "Add README file" box
- DO NOT add a .gitignore
- DO NOT choose a license
- Click Create repository

Adding a remote

Once you have created the repo on GitHub, you can link it to your local folder - I created an empty repo on Git called `empty`

Enter the following commands in the root directory of your local repo:

```
git remote add origin https://github.com/julia-wrobel/example.git
git branch -M main
git push -u origin main
```

Basic GitHub workflow

- Set up repo locally
- Set up repo remote
- Link local and remote

Each time you make changes and are ready for a new commit, do the following:

```
git pull # check that there aren't discrepancies between local and remote

git status
git add --a
git commit -m "informative commit message!"
git status # I like to check before I push

git push
```

Reproducible examples

A reproducible example allows someone else to recreate your problem by just copying and pasting R code. **Critical for**:

- Posting on StackOverflow
- Debugging with others through email
- Emailing a package author to ask why their code doesn't work
- Hadley Wickham: how to write a reproducible example