WEEK 2,
ADVANCED R & GITHUB:

FUNCTIONS, PACKAGES, & VERSION CONTROL

+

RSTUDIO

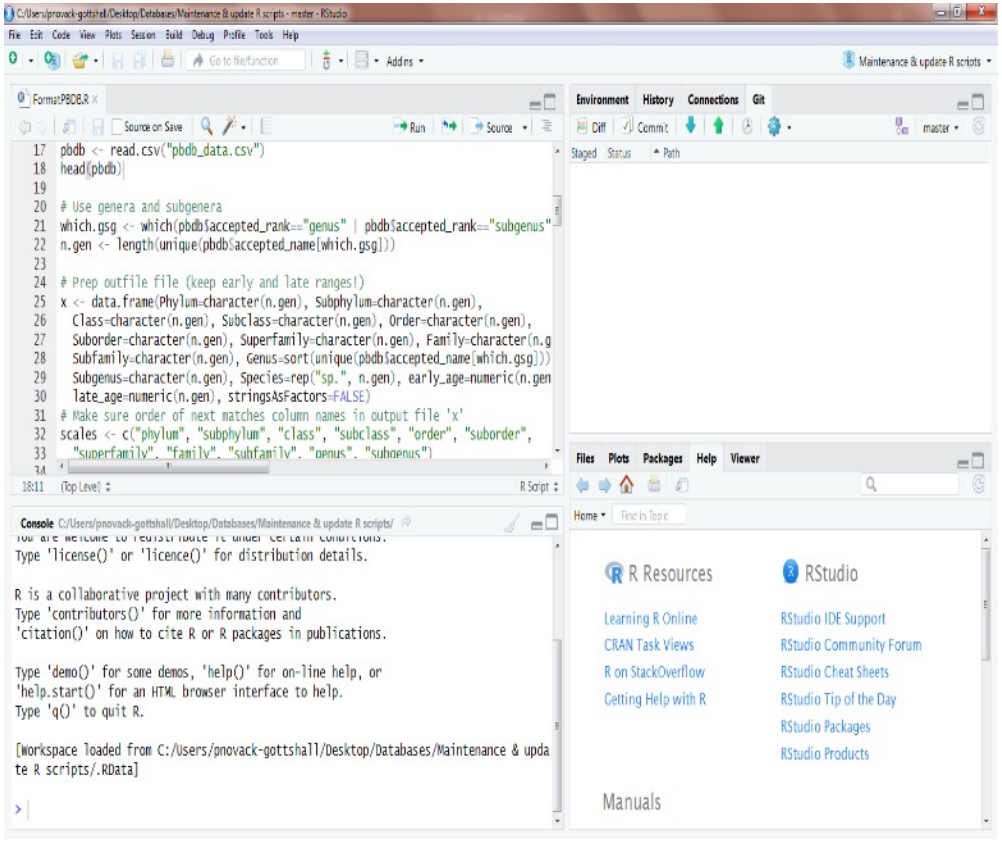
A POWERFUL INTEGRATED ENVIRONMENT FOR USING R

RStudio IDE

- Integrated development environment (IDE) for R
- Versatile open-source GUI interface
- Fully integrated to R resources
- Loaded with powerful and efficient add-ins for GitHub, building packages, and writing/formatting/trouble-shooting code
- Widely used
  - Windows, Mac OS, Linux

RSTUDIO DEMO

- Different panels
- Common shortcuts (toggle between panels, execute)
- "Modify keyboard shortcuts"
- Environment summarizes data, objects, functions
- Reflow comments and reformat code
- Tag code sections with chapter-like "levels"
- Autocompletion of functions and objects in code editor
- Built-in help and plotting
- Downside: no `alarm()`!

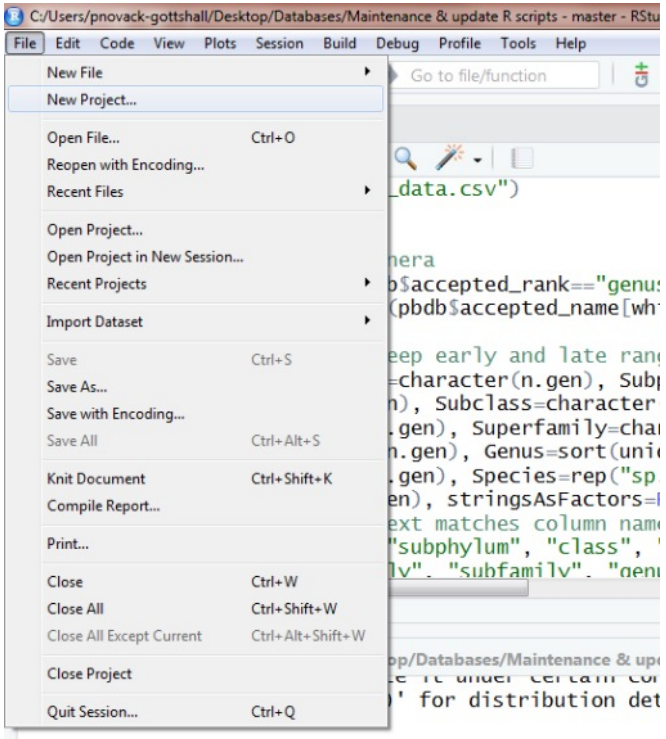


RSTUDIO PROJECTS

- Projects (.Rproj files) allow you to keep all documents, data files, code, and code history in their own folder
- Each project file can be synced to GitHub
  - If building a package, advisable to set up within its own project
  - (Note: need to create the repository on GitHub before it will sync)

Creating a new project in RStudio

- File > New project
  - New directory: Set up a "New project"
    - Choose "R package" if building a new package
  - Existing directory: If want to build a project around existing material
  - Version control: If want to create a project from a GitHub or Subversion repository



GitHub and Version Control

COLLABORATE, SHARE, AND UNDERSTAND HOW YOUR CODE HAS CHANGED

VERSION CONTROL, GIT & GITHUB

- Version control tracks changes to files
  - When changed? (timestamp)
  - Who made the change? (blame)
  - How changed? (differencing)
- Seamless documentation of how code has changed through time
  - Prevents (tracks) errors
  - Allows collaborative code development
  - Coordinates changes among different users
- Git is an efficient and commonly used version control system (VCS)
  - GitHub is an online repository for storing, exchanging, and sharing your files

BENEFITS OF GIT/GITHUB

1. Version control: it makes it easier for you and users to track changes to your code over time
2. Changes (differences) are color coded for easy documenting
3. Integrated with RStudio while building your package / project

Above available even if do not submit to GitHub!

---

4. Users can request feature changes or post issues
5. Users can download beta packages directly to R

```
library(devtools)
devtools::install_github("user/repo")
library(repo)
```

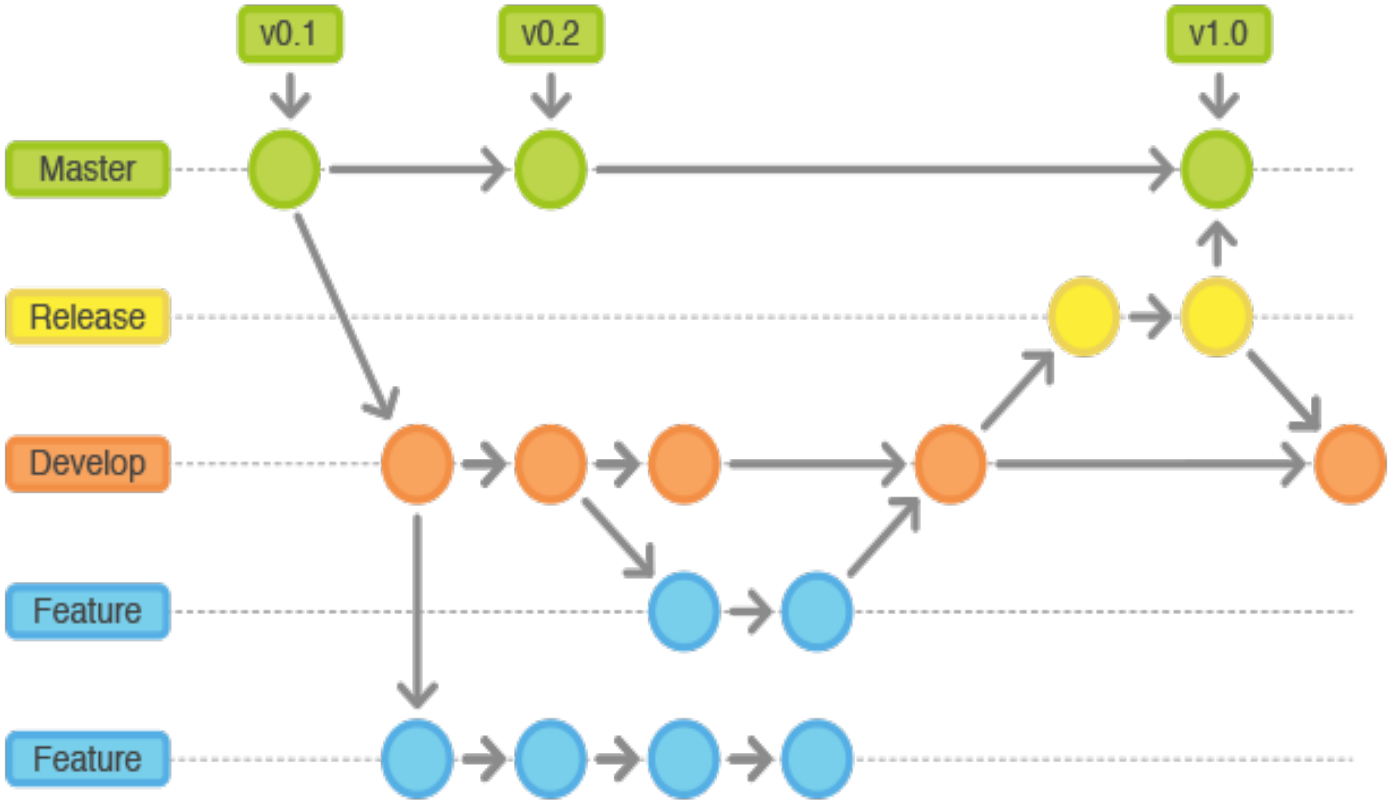
6. Easily collaborate with others who are also developing your code
7. Allows you to test each code change across different computational environments if link your repository to Travis CI
8. When submit a CRAN package, can track downloads

REPOS & BRANCHES

- Repository ("repo"): The contributed files for a project/package/platform
  - Can be local or remote
- Master (recently renamed to “main”): The central "production-ready" "public release" fully functioning GitHub repository
  - Never* work with the master (main) directly!

    (* unless you are the sole developer)
- Branch: Where you work on one part of the project
  - There can be many branches
    - One branch for each new feature
    - Can be developed by one or several users
  - When done, branches are merged back into the master
- Easier to create and merge branches via GitHub rather than RStudio (unless comfortable using shell commands)

MASTER AND BRANCHES



https://stackoverflow.com/questions/20755434/what-is-the-master-branch-and-release-branch-for

DEVELOPMENT WORK FLOW (PULL, STAGE, COMMIT, & PUSH)

1. Pull: When you download the current version from the branch/master
   - Also merges with your current code, if different
   - A fetch does not merge

   1B. Resolve any conflicts indicated with a merge "`<<<<<<< HEAD`" tag
   - You are required to resolve it before you can "push" back to the branch/master
2. Make changes to your code
   - Changes are staged to be committed later
3. Commit: A formal (documented) commitment to your staged work
   - "Commit early and often"
   - All commits are version controlled with differences
   - Annotate (briefly) what each commit does
4. Push: When you upload your "commit" to the branch or merge to the master

- Note differencing works with most text-based file types (but not Excel)

Functions in R

DO THE THINGS YOU NEED FASTER AND SIMPLER

BASIC SYNTAX OF R FUNCTIONS

- R is a function-oriented programming language
- Basic syntax:

```
fn <- function(args, …) {
    out <- …
    return(out)
}
```

- formals (arguments, args): the list of inputs
- body (expression): what is being done
- environment: each function is evaluated in its own "evaluation frame" environment
  - namespace: list of function/object names within a named environment (such as a package name)

- Exception: primitive functions, like `sum()`, lack these elements and call C code

NAMING FUNCTIONS (AND OBJECTS)

- Functions "do" things — give them "action" names
  - `mean()`, `rnorm()`, `ecospace::create_ecospace()`, `paleoTS::fit3models()`, `MASS::fitdistr()`, `vegan::rarefy()`
- Objects "are" things — give them "noun" names
  - `SpList`, `output1`, `ecospace`, `samples`, `traits`, `midpoints`
- CamelCase and Snake_Case for multiple word names:
  - Avoid all lowercase (and all UPPERCASE) if multiple words
    - `SampleOne` vs. `sampleone` vs. `SAMPLEONE`
  - Some coders recommend avoiding using periods to separate because periods mean things in some computer languages
    - `SampleOne` vs. `Sample.One`

EXAMPLE OF ENVIRONMENTS

```
f1 <- function (x) {
      a <- x^2 + 1
      return(a)
 }
x <- 2
a <- 3


f1(x)
```

```
[1] 5
```

```
a
```

```
[1] 3
```

EXAMPLE OF ENVIRONMENTS

```
mean(1:3)
[1] 2
mean <- function(x) { "hello, world" }
mean(1:3)
[1] "hello, world"
base::mean(1:3)
[1] 2
```

OUTPUT FROM FUNCTIONS

- Last assignment returned by default
  - Better to explicitly code what to return
  - `return(out)`: returns (and prints)
  - `invisible(out)`: returns without printing

- Compare:

```
x <- function(x) {return(x)}
x(2)
x <- function(x) {invisible(x)}
x(2)
```

OUTPUT FROM FUNCTIONS

- If more than one output value, typically output as a list
  - Example:

```
lm_out <- lm(Fertility ~ . , data = swiss)
str(lm_out)
```

- If output is used in other package functions, output can be given a novel class
  - Example:
    - `class(y) <- "MyNewClass"`

WARNINGS AND ERRORS

- Help users by providing informative warnings and errors!
  - Do not go crazy with exotic troubleshooting
    - Every check adds time
    - GitHub issues can alert designers to common errors (or mis-uses) among users
  - Fine to insert brief comments in your functions

- To add warnings that do not stop function:

```
if (…) warning("…")
```

- To trigger an error that stops function:

```
if (…) stop("…")
```

COMMON SCENARIOS TO TEST FOR

1. What if input has missing data (NAs)?
   - Is it appropriate to call `mean()`, `sum()`, `sd()`, etc. with `na.rm = TRUE`
2. What if user provides different data class?
   - Trigger error if wrong class, inappropriate dimensions, incorrect `names()`
   - **How does function handle NA, NULL, logicals, characters, and numbers?**
3. Case-sensitive error in arguments?
   - Use `tolower()` or `toupper()` to internally fix case
4. What if nonsensical arguments?
   - Example, input negative values for a lognormal distribution or presence of ties for a continuous distribution
   - Trigger warning, and provide informative explanation of why inappropriate!

ACTIVITY: LEARN FROM OTHERS

- With a partner, pick a favorite function and study how it works
  1. Identify:
     - `args(function) / formals(function)`
       - Is "`...`" an allowed argument? If so, how handled?
     - `environment(function)`
     - `body(function)`
     - `str(output)`
  2. What inherent trouble-shooting is included (if any)
     - Are object classes checked?
     - What conditions trigger warnings vs. errors?
  3. What is returned?
  4. What is the structure of the output?
  5. How does the function work?