Graduate School Class Reminders

- Maintain six feet of distancing
- Please sit in the same chair each class time
- Observe entry/exit doors as marked
- Use hand sanitizer when you enter/exit the classroom
- Use a disinfectant wipe/spray to wipe down your learning space before and after class
- Media Services: 414 955-4357 option 2

Documentation on the web

- CRAN: `http://cran.r-project.org`
- R manuals: `https://cran.r-project.org/manuals.html`
- SAS: `http://support.sas.com/documentation`
- Step-by-Step Programming with Base SAS 9.4 (SbS): `https://documentation.sas.com/api/docsets/basess/9.4/content/basess.pdf`
- SAS 9.4 Programmer s Guide: Essentials (PGE): `https://documentation.sas.com/api/docsets/lepg/9.4/content/lepg.pdf`
- Wiki: `https://wiki.biostat.mcw.edu` (MCW/VPN)

HW 2: ISO 8601 and the Proleptic Gregorian Calendar

- ISO 8601 dictates that the Proleptic Gregorian Calendar be used for dates prior to the start of the Gregorian Calendar
- Is the R `Date` class ISO 8601 compliant?
- If not, then at what date does it diverge?
- Unix/R define 1970-01-01 as date 0 counting for-/back-wards
- The Julian Period covers all of written human history starting on Monday 4713BCE-01-01 which is
  ISO 8601 date -2,440,588
- What Year-Month-Day is this in the corresponding Proleptic Gregorian Calendar?
- Hints: do this without `for` loops
  the R interpreter will take a long time for 2.4M iterations
  using `data.frames` and vectorization this is pretty fast
  for example, in `julian.R` see my implementation of the Proleptic Julian Calendar
- Use two `data.frames`: one for each Calendar
  and combine them (but you can't use `merge` for 2.4M records)
- Use `sprintf` to concatenate character vectors

Calendars, dates and ISO 8601: see Wikipedia

- The `Date` class represents dates
- The Egyptian Calendar had 365 days/year circa 3000BCE
- Leap days were added haphazardly as seen fit
- Emperor Julius Caesar solidified the calendar
- The Julian Calendar created a leap day once every 4 years (with some initial issues and corresponding corrections)
- The first leap year of the Julian Calendar (as planned) 41BCE
- The first day of the Julian Calendar is 45BCE-01-01
- The last day of the Julian Calendar is 1582-10-4 (Thursday)
- The first day of the Gregorian Calendar is 1582-10-15 (Friday)
- In 1582, dates 10-5 through 10-14 didn't occur in Roman Catholic countries (it is named after Pope Gregory XIII)
- This change returned the equinoxes to the same days as the first year of the Julian Calendar
- Gregorian Calendar Leap Year Rule
  Every year that is divisible by four is a leap year
  Except for years that are divisible by 100
  Unless they are divisible by 400, e.g., Y2K

The Julian Period, Proleptic Calendars and ISO 8601

- Proleptic Calendars adopt the rules of a Calendar
  and they run backwards in time before that Calendar starts
- The Julian Period starts with Proleptic Julian Date
  4713BCE-01-01 and covers all known recorded human history
- ISO 8601 dictates a Proleptic Gregorian Calendar
- The Julian and Gregorian Calendars have no year zero
- . . . , 2BCE, 1BCE, 1CE, 2CE, . . .
- But ISO 8601 dicates the following definition of year

Year	ISO 8601	Julian
	-4712	4713BCE
	⋮	⋮
	-1	2BCE
	0	1BCE
	1	1CE
	⋮	⋮

A brief overview of R

- Interpreted language (very slow as opposed to compiled)
- Interpreter written in C and Fortran
- Considered a multi-paradigm language
- For example, it is considered to be object-oriented and functional (as well as other paradigms)
- Object-oriented and vectorized for user-friendliness: vectors and matrices are natural choices for *objects*
- "Everything is an object"
- Dynamically Typed (as opposed to Statically Typed)
  implicitly typed by first usage
  rather than explicitly by definition/declaration
- Provides multi-threading by *forking* EXCEPT on Windows
  Windows does not have this capability so no multi-threading
  An odd choice since MS was an early licensee of UNIX
  Pioneering the port of UNIX to many CPUs as XENIX
  And MS-DOS was a "dumbed-down" Unix-like system
  If GNU Linux can clone forking, then why not MS?

A brief overview of R

- Much R functionality is written in R itself
  with calls to compiled C/C++/Fortran as needed (very fast)
- Mainly, these are R *add-on* packages as explained in
  the R FAQ; see section 5.1 in
  `https://cran.r-project.org/doc/FAQ/R-FAQ.html`
- There are two groups of packages distributed with R
- The 14 so-called *base* packages: base, compiler, datasets, grDevices, graphics, grid, methods, parallel, splines, stats, stats4, tcltk, tools and utils

  ```
  > installed.packages(priority="base")
  ```
- The 15 so-called *recommended* packages: KernSmooth, MASS, Matrix, boot, class, cluster, codetools, foreign, lattice, mgcv, nlme, nnet, rpart, spatial and survival

  ```
  > installed.packages(priority="recommended")
  ```
- Comprehensive R Archive Network: 18558 packages `https://cran.r-project.org/web/packages/index.html`
  Task Views `https://cran.r-project.org/web/views`

Introduction to R by Venables and Smith: R-intro

- Without emacs, to launch R: `$ module load R/4.0.4; R`
- With emacs, launch the latest version of R: `M-x R-4.0.4` or simply `M-x R`
- The R quit command: `q()`
- Getting help: `> ?q` or `> help("q")`
- In the `*R:~*` at the `>`, type `??` for help on help
- Almost everything is an object (except keywords)
- To see the definition of q, type `> q`
- To search R help and installed packages help
  type `> help.start()`
  VERY SLOW IF LOTS OF PACKAGES ARE INSTALLED
- Double quotes vs. single quotes: the docs shows double quotes in most places, but single quotes seem to work fine and they are easier to type: I will try to use double in my notes

Section 1: Storing and re-loading objects

- Names contain alphanumeric characters, underscore or period
- If they start with period, the second character must be a letter
- A case-sensitive language, i.e., distinct variables a and `A`
- The `source` function is convenient for loading objects like data, functions, etc.
- `source("lecture2.R")`
- However, re-creating some objects (like large data sets or complex analyses) might be very time-consuming
- DON'T DO THIS: R-intro describes saving R objects in `.RData` which is automatically re-loaded when you re-launch R in that directory: IT IS TOO EASY TO FORGET/MISUSE
- Save the objects that you want to keep and re-load as needed
- `saveRDS(object, "object.rds")`
- `object = readRDS("object.rds")`
- R loads all objects into memory so be cognizant of their size
- `object.size(object)`
- `rm(object); gc()`
- `?gc`

Section 1: View-/edit-ing objects

- You can view an object just by typing its name at the prompt:
  `> object`
- However, for large objects, this is not very user-friendly
- For large objects, it might be better to view/edit them
- If you just want to view (and NOT edit):
  ```
  > sink("object.Rout"); object; sink()
  C-x C-f object.Rout
  ```
- To edit, start an `emacsclient` by `M-x server-start`
- `> sink("/dev/null") ## shut off the R console`
- `> ls() ## notice that it produces NO output`
- `> edit(object, "object.R")`
- When you are done: `C-x #`
- `> sink() ## restore the R console`
- `> ls() ## now produces output as usual`
- N.B. the file `object.R` remains for your viewing

Section 2: R is naturally vectorized

- All `atomic` types are vectors of length zero or more: numeric, logical, character, complex and raw
- Roughly in that order of importance
- `length` function returns their current length
- Vectors can easily expand by allocatting beyond it
- > `x = 1` creates a vector of length 1
- > `(x = 1)` parentheses echo assignment values
- Like > `x = 1; x`
- Or to echo any expression > `1/x`
- > `x[2]`=0 expands to length of 2
- > `x = c(10.4, 5.6, 3.1, 6.4, 21.7)` creates a vector
- > `?c`
- see `lecture3.R`

Section 2.3: Sequences

- Integer sequences: `1:30` same as `c(1, 2, ..., 29, 30)`
- General sequences: `seq(-5, 5, by=0.2)` same as `c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0)`
- See > `?seq` for all of the possibilities

Section 2.4: Logical vectors

- Three possible values: TRUE, FALSE and NA (BEWARE!)
- The comparison operators: `<`, `<=`, `>`, `>=`, `==`, `!=`
- Two equal signs for equality like C
- Combining expressions with `|`, `||`, `&`, `&&`
- N.B. the difference between ORs `|` and `||`
  and the difference between ANDs `&` and `&&`
- `||` and `&&` SHOULD ONLY BE USED WITH SCALARS
  they stop evaluating once the result is known
  so very useful for conditional expressions
  ```
  FALSE || TRUE || FALSE
  TRUE && FALSE && TRUE
  ```
- `|` and `&` are for vector operations (similar to addition/etc.)
- Functions that return a scalar from vectors: `any` and `all`
- `TRUE` can be coerced to 1 and `FALSE` can be coerced to 0
- see `lecture3.R`

Section 2.6: Character vectors

- Working with character strings is NOT one of R's strengths
- But this is true of most languages that statistician's use
- The focus of statistics is largely on numerical computing
- Many functions inherited from the AWK language (mainly New AWK or NAWK) from the UNIX family (cloned by the GNU as GAWK)
- For example, see `sub`, `gsub` and `substr`
- Three functions commonly used (NOT inherited from AWK): `paste0`, `paste`, `nchar` and `sprintf`
- see `lecture3.R`

Section 2.7: Vector indices

- Integer indices: `x[1:3]` returns the first three items
- If there are fewer, then returns `NA`
- `x[0]` returns nothing
- `x[-1]` returns everything EXCEPT `x[1]`
- Logical indices: `x[!is.na(x)]`
- BEWARE: due to coercion of complex expressions you can accidentally request TRUE as index 1 and FALSE as index 0 (which is nothing): can be very difficult to debug because it happens without warning
- You can protect yourself by enclosing all logical index expressions within `which` function. See `?which`
- `x[which(!is.na(x))]`

Section 3: Object attributes

- All objects have a *mode*: > `mode(x)`
- But mainly of interest for atomic vectors
- Some objects have a *class*: > `class(x)`
- A capable function for object structure: > `str(x)`
- `as.` functions for manual coercion between types
- > `as.integer(x)`
- > `as.character(x)`
- All of an object's attributes, if any, can be listed: `attributes(x)`
- They can be extracted: `attr(x, "names")`
- And set: `attr(x, "names")=letters[1:5]`

Appendices

- Appendix A lists a lot of interesting functions that we will learn about
- But, `attach` is generally frowned upon today
- A short version of the help provided in Appendix B.1 is generated by `$ R --help`

Appendix B.4: Scripting with R

- It is not about “Scripting” per se
- Rather, asynchronous/background/batch R jobs
  i.e., NOT interactive
- It does provide useful information (which is why it is assigned)
- Particularly, the function `commandArgs` and `stdin` discussion
- Generally, I do something like the following
- the standard Unix way of doing things with re-direction
  building on what is shown in Appendix B.1
- `stdin` is comming from `trees.R`
- `stdout` and `stderr` are going to `trees.Rout`

```
nohup R --no-save < trees.R >& trees.Rout &
```