Day 1: Mon 29th Nov - The Unix Shell

Instructor: Srinivasa **Rao**
Helpers: Everlyn Kamau, James Scott-Brown, Badran Elshenawy

Workshop website
https://2cjenn.github.io/2021-11-29-oxford-online/

Code of conduct
https://docs.carpentries.org/topic_folders/policies/code-of-conduct.html

Pre-workshop survey
https://carpentries.typeform.com/to/wi32rS?slug=2021-11-29-oxford-online

Ice-breaker
What is your favourite animal?
Rao: Elephant

James: cat
Aislinn - sea turtle
Claire Z - Zebra
Beth H - Platypus
Hannah - Pangolin
Valeria-Owl
Claire C - Capybara
Mehpare - Cat
Fabian - Chameleon
Everlyn - Penguin
Marsha - Fox
Amy - golden retriever

Use of Teams

- Unmute yourself and ask Qs
- Ask Qs in the chat
- Use emojis on Teams for feedback

Course files we will use today

Download from here: https://swcarpentry.github.io/shell-novice/data/shell-lesson-data.zip

$ sign - command/shell prompt
ls
ls --help (Windows)
man ls (Mac/Linux)
q to quit from man pages
pwd - your current path or folder (working directory) - print working directory
cd - change directory

Absolute vs Relative Paths

Starting from `/Users/amanda/data`, which of the following commands could Amanda use to navigate to her **home directory**, which is `/Users/amanda`?

1. `cd .`
2. `cd /`
3. `cd /home/amanda`
4. `cd ../..`
5. `cd ~`
6. `cd home`
7. `cd ~/data/..`
8. `cd`
9. `cd ..`

Aislinn - cd ../..
Claire Z - cd ../
Beth H - cd ../
Hannah - cd ~
Valeria-
Claire C -
Mehpare - cd .. and cd `~/data/..`
Fabian - cd ../

Everlyn - cd ~
Marsha - cd ~ and cd ..
Amy - cd ~, cd ../

List filenames matching a pattern

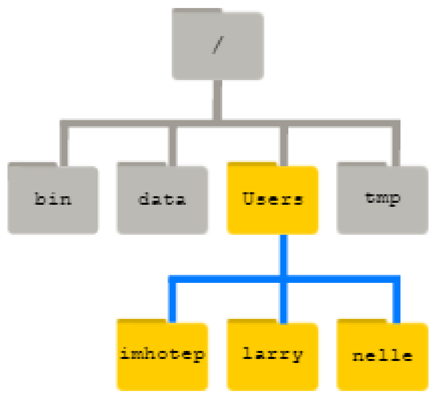
When run in the `molecules` directory, which `ls` command(s) will produce this output?

```
ethane.pdb methane.pdb
```

1. `ls *t*ane.pdb`
2. `ls *t?ne.*`

3. `ls *t??ne.pdb`
4. `ls ethane.*`

James:
Aislinn - 3
Claire Z - 3
Beth H - 3,
Hannah - 3
Valeria-
Claire C -
Mehpare - 3
Fabian - 1,3
Everlyn -
Marsha - 1 (plus more),3
Amy - 1,3



mkdir - ‘make’/create directory
nano - text editor
  ^O - to save
  ^X - to exit
cat - print contents to screen
head / tail - first /last 10 lines
Less - view files screen by screen
rm - remove/delete file or folder
  -r flag to remove directories
  -v verbose
cp - copy
mv - move
Wildcards - * matches zero or more characters, ? matches one character

wc - word/line/character count (depending on options provided)
sort - sorts text; -n option sorts numerically
| - passing output from one command as input to another
> - redirect output (to a file)

>> - redirects output and appends to specified file

```
for thing in list_of_things
do
    operation_using $thing    # Indentation within the loop is not required, but aids legibility
done
```

```
for thing in list_of_things; do operation_using $thing; done
```

Saving to a File in a Loop - Part One

In the `shell-lesson-data/molecules` directory, what is the effect of this loop?

```
for alkanes in *.pdb
do
    echo $alkanes
    cat $alkanes > alkanes.pdb
done
```

1. Prints `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, `pentane.pdb` and `propane.pdb`, and the text from `propane.pdb` will be saved to a file called `alkanes.pdb`.
2. Prints `cubane.pdb`, `ethane.pdb`, and `methane.pdb`, and the text from all three files would be concatenated and saved to a file called `alkanes.pdb`.
3. Prints `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, and `pentane.pdb`, and the text from `propane.pdb` will be saved to a file called `alkanes.pdb`.
4. None of the above.

Maybe user wanted to to append rather than overwrite, >> is needed to append

Shell scripts - record code for later use
Repeat commands at a later time
Store a long list of commands, make code more legible
$1 - in a shell script, $1 indicates the first argument given at the command line when the shell script is invoked. $2 is the second argument, and so on.

Create a shell script in creatures folder, where you extract the 5th line of a .dat file; run a for loop with this script to print the 5th line of each file.

for i in *.dat; do echo $i; head -n 5 $i | tail -n 1; done

for each_dat_file in *.dat; do echo $each_dat_file; bash exercise.sh $each_dat_file; done
exercise.sh
    head -n5 $1 *.dat | tail -n 1

HW: use tail first and then head, to do the same task

grep - searches for a given text in a file
find - finds a file on your disk drive

Day 2: Tues 30th Nov - Version Control with Git

Instructor: James Scott-Brown
Helpers: Cassandra Gould van Praag, Jennifer Collister, Badran Elshenawy

Workshop website
https://2cjenn.github.io/2021-11-29-oxford-online/

Code of conduct
https://docs.carpentries.org/topic_folders/policies/code-of-conduct.html

Pre-workshop survey
https://carpentries.typeform.com/to/wi32rS?slug=2021-11-29-oxford-online

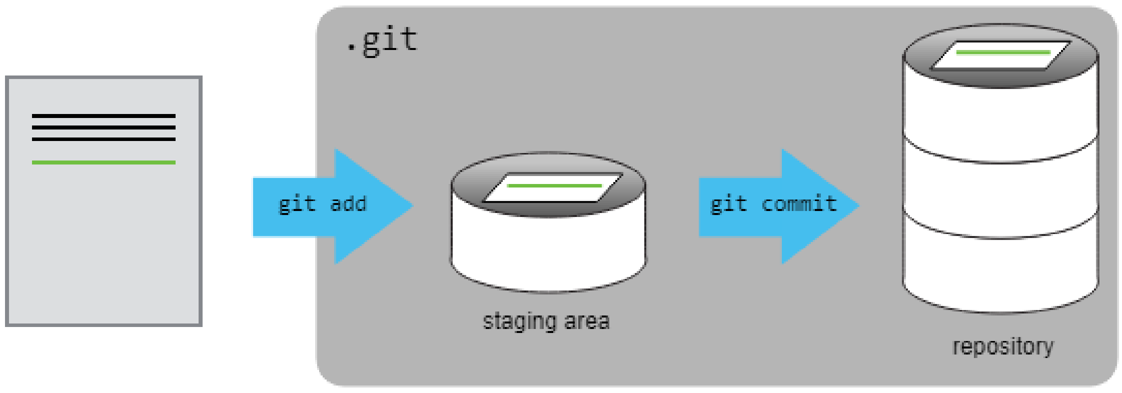
Preparation: as well as ensuring that you have installed Bash, Git and nano, please also create a (free) GitHub account. I suggest that you have Microsoft Teams and a terminal window open side-by-side during the session.

Participants:

James - pizza
Claire C - Chocolate
Hannah - cucumber
Amelia - Icecream
Mehpare - Steak
Dani - peanut butter
Fabian - Lasagna
Cass (she/her) - potatoes, all ways.
Claire Z - Chinese dough sticks
Everlyn - chicken curry
Jennifer (she/her) - chocolate
Aislinn - pasta
Marsha - mac & cheese
Clare W - crisps

Commands introduced

`git init` - initialise a new git repository in the current working directory
`git status` - check the status of our repository - including showing which branch you're on
`git checkout -b main` - create a new branch called "main" and swap to it, if your config hadn't worked and your repository initialised on master branch!
`git add` - add a file to the staging area
`git commit` - commit the changes you have staged (note: you have to "add" the files before you can "commit" them)
`git diff` - Shows changes you made relative to the index (staging area for the next commit). In other words, the differences are what you *could* tell Git to further add to the index but you still haven't.
`git diff --staged` - compares any staged changes to what's in the last commit
`git log` - log of commits made, in reverse chronological order (use **`-n`** option to show only the last n commits)



Getting out of vim

If at any point you have ":" (colon) at the bottom of your terminal, you are in Vim
Hit Escape, then press :q to exit Vim
See this stackoverflow answer for more info
https://stackoverflow.com/questions/11828270/how-do-i-exit-the-vim-editor

.gitignore

Ignore files such as data that you don't want Git to track

Useful examples of .gitignore files for various languages:

https://github.com/github/gitignore/

https://github.com/github/gitignore/blob/master/Python.gitignore

https://github.com/github/gitignore/blob/master/R.gitignore

https://github.com/github/gitignore/blob/218a941be92679ce67d0484547e3e142b2f5f6f0/Global/Windows.gitignore

https://github.com/github/gitignore/blob/218a941be92679ce67d0484547e3e142b2f5f6f0/Global/Linux.gitignore

https://github.com/github/gitignore/blob/218a941be92679ce67d0484547e3e142b2f5f6f0/Global/macOS.gitignore

SSH keys

Generally one per user account - don’t need to create a new one for every repository
Good practice to have a new key per device - when you add the key to GitHub (or equiv) you can give it an informative name to indicate which device the key is for

Remotes

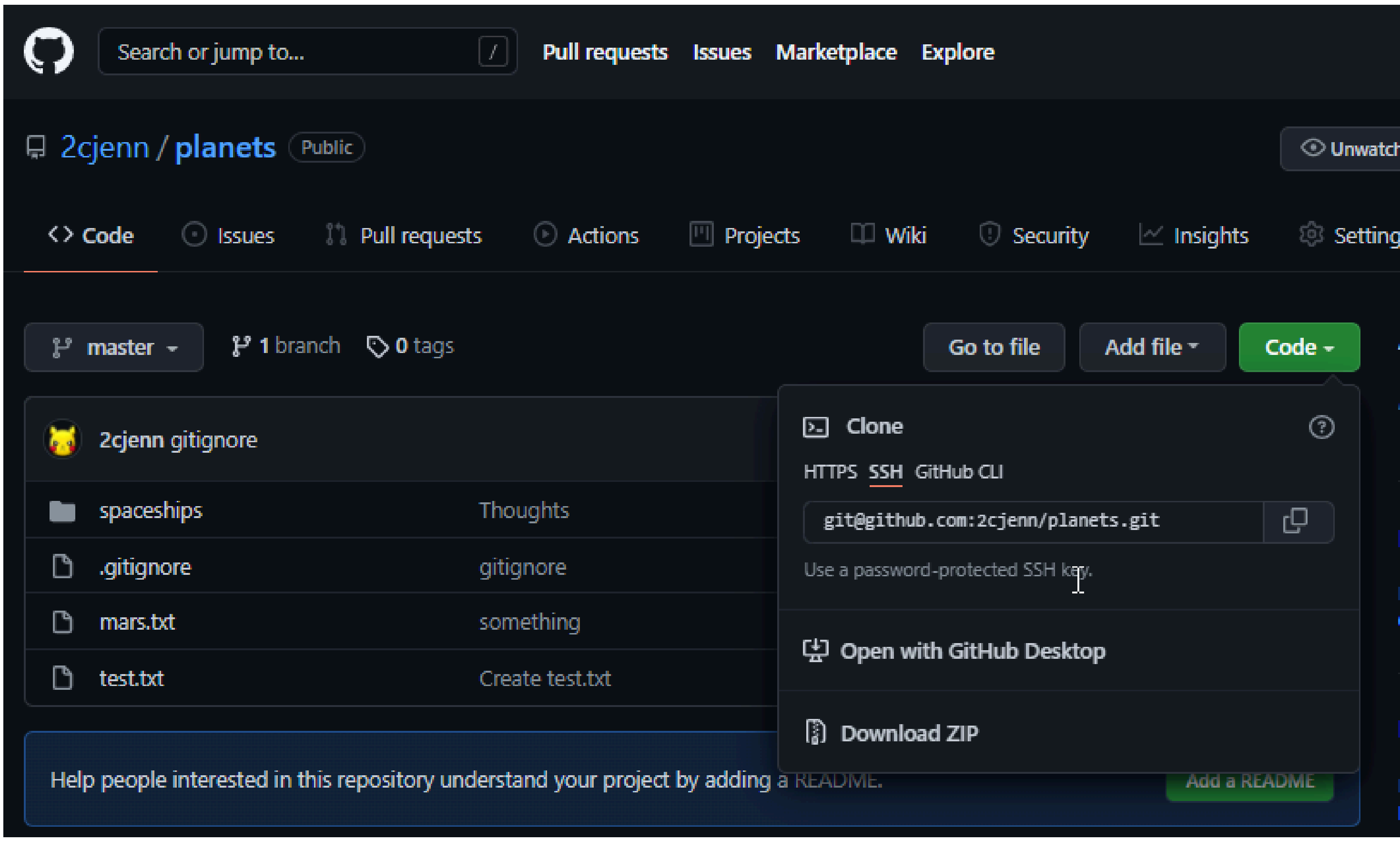
Other repositories that are on different computers.
GitHub, GitLab, etc are websites that host remote repositories

To add a new remote (called “origin”) to your local repository:
```
git remote add origin [link to remote repo eg
git@github.com:username/repo.git]
```

Be sure to use the ssh link to the repo (git@github.com:username/repo.git) rather than the https one (https://github.com/username/repo.git)
See screenshot below to see where you can find these links in an existing repository

To view the remotes set up for your repository
```
git remote -v
```
The v stands for “verbose”

To change the url for your remote
```
git remote set-url origin [new/url/here]
```



2cjenn gitignore

spaceships	Thoughts
.gitignore	gitignore
mars.txt	something
test.txt	Create test.txt

```
git@github.com:2cjenn/planets.git
```

Use a password-protected SSH key.

Help people interested in this repository understand your project by adding a README.

Working on a remote

`push` changes to the remote, **`pull`** changes from the remote

`git push origin main` - push changes on branch “main” to the remote called “origin”

Exercises

Committing Multiple Files

The staging area can hold changes from any number of files that you want to commit as a single snapshot.

1. Add some text to mars.txt noting your decision to consider Venus as a base
2. Create a new file venus.txt with your initial thoughts about Venus as a base for you and your friends
3. Add changes from both files to the staging area, and commit those changes.

bio Repository

- Create a new Git repository on your computer called bio.
- Write a three-line biography for yourself in a file called me.txt, commit your changes
- Modify one line, add a fourth line
- Display the differences between its updated state and its original state.

Recovering Older Versions of a File

Jennifer has made changes to the Python script that she has been working on for weeks, and the modifications she made this morning “broke” the script and it no longer runs. She has spent ~ 1hr trying to fix it, with no luck…

Luckily, she has been keeping track of her project’s versions using Git! Which commands below will let her recover the last committed version of her Python script called data_cruncher.py?

1. $ git checkout HEAD
2. $ git checkout HEAD data_cruncher.py
3. $ git checkout HEAD~1 data_cruncher.py
4. $ git checkout <unique ID of last commit> data_cruncher.py
5. Both 2 and 4

   Answer: 5

Reverting a Commit

Jennifer is collaborating with colleagues on her Python script. She realizes her last commit to the project’s repository contained an error, and wants to undo it. Jennifer wants to undo correctly so everyone in the project’s repository gets the correct change. The command git revert [erroneous commit ID] will create a new commit that reverses the erroneous commit.

The command git revert is different from git checkout [commit ID] because git checkout returns the files not yet committed within the local repository to a previous state, whereas git revert reverses changes committed to the local and project repositories.

Below are the right steps and explanations for Jennifer to use git revert, what is the missing command?

1. ________ # Look at the git history of the project to find the commit ID
2. Copy the ID (the first few characters of the ID, e.g. 0b1d055).
3. git revert [commit ID]
4. Type in the new commit message.
5. Save and close

   Answer: git log

Getting help

You can access short usage information about a git command using the -h option:

```
$ git diff -h
```

```
usage: git diff [<options>] [<commit> [<commit>]] [--] [<path>...]
```

This requires some explanation if you're seeing it for the first time.

Things in *angle* brackets refer to the *values* of whatever is written: e.g., `<commit>` means that you should type something representing a commit (e.g., a hash or `HEAD`), rather than literally typing `commit`. Things not in angle brackets are literal strings: `git diff` means you should type `git diff`.

Things written in square brackets are optional, so `[<options>]` means that specifying the value of options is optional. These can be nested: `[<commit> [<commit>]]` mean that you can optionally specify a commit, and that if you do then you can optionally also specify a second commit.

You can access more detailed information in the form of a manpage using either `git diff --help` or `man git diff`. You can scroll up/down by a line using the arrow keys, or down a page using the space bar. You can exit by pressing the q key.

Further resources:

The solutions to some of the most common problems are listed at Oh shit, git website. The solutions to many more problems can be found on Stack Overflow.

These two cheat sheets may be useful.

The Git *User* Manual is a good introduction that covers more than we had time to discuss today; it's roughly a book chapter in length.

The official *reference* manual contains the same man page text that you can access with commands like `git config --help` or `man config help`: it's a definitive reference, but probably not the easiest place to learn about something for the first time.

The Pro Git book is freely available online, and provides a book-length explanation of how to use Git.

One of the major topics that we didn't cover today is branching; I suggest reading about branches in either the Git *User* Manual or the Pro Git book.

Open practices

Advice on Open Source Licensing specific to the University of Oxford.

OxFoss: https://ox.ukrn.org/events/#Oxford-Free-Open-Source-Software

Community Call on Executable manuscripts

Reproducible Research Oxford

Q&A from Day 2

- say you made a typo in a message, can you change the message?
  - **`git commit --amend`**
- Exiting vim
  - https://stackoverflow.com/questions/11828270/how-do-i-exit-the-vim-editor
- Are SSH keys specific to each repository?
  - No, one per user account, can be used for multiple repositories
- My terminal doesn't let me paste
  - In “Git Bash” terminal, use Shift + Insert
  - In Command Prompt (windows) use right click
- The authenticity of host 'github.com (140.82.121.3)' can't be established.
  - This should only happen the first time you’re setting up a new ssh key to a repo, let it continue!
- why is my git asking me to sign in when I do the push command if I'm already signed in?
  - Might be using the https link rather than the ssh one (see notes in remotes section)
- I'm getting an error that host key verification failed
  - Try the following command to add the website to known hosts
  - `ssh-keyscan -t rsa github.com >> ~/.ssh/known_hosts`
- where did the git@github.com... come from?
  - in your repository on github there is a "code" button, which gives this address
  - See screenshot in remotes section
- What if you have large files with multiple conflicts? Do you have to do these merges all in Bash or is there another way?
  - You can use IDEs to resolve conflicts as well - they often have little wrappers around chunks so you can just click to keep one set of changes
  - But it's good to understand what's actually going on - git is genuinely just adding these <<<<< and >>>> lines in text to the files, and you can just edit it manually

Day 3: Weds 1st Dec - Programming with R

Instructor: Jennifer Collister
Helpers: Everlyn Kamau, Alex Sauer, Srinivasa Rao, Xiaonan Liu

Lesson page: http://swcarpentry.github.io/r-novice-inflammation/

Icebreaker:
Please type your name and favourite movie :)

Jennifer - howl's moving castle
Rao - Before Sunrise/Sunset
Aaron - Enemy
Alex - Victoria
Xiaonan Liu - Black Swan
Everlyn - point break

Valeria- Secret Life of Pets
Hannah - lord of the rings
Clare W - School of Rock
Aislinn - Toy Story
James - shawshank redemption
Danial Q - Catch Me If You Can
Claire C - Bambi
Marsha - V for Vendetta
Claire - Aladin
Claire Z - Lord of the Rings trilogy
Mehpare - N/A

CSV - Comma-separated value file
TSV - Tab-separated value file
Run line in RStudio script panel - 'Run' button / ctrl + enter / cmd + enter
<- or **=** are assignment operators in R; they are equivalent but R users prefer <- because it is distinctly different from '==' and also there is a distinction between variable assignment and argument specification (within function parentheses)

Read CSV file into R:
```
dat <- read.csv(file = "data/inflammation-01.csv", header = FALSE)
```

Variable assignment (detour):
```
weight_kg <- 55
weight_kg
# weight in pounds:
2.2 * weight_kg
```
View top few lines of data:
```
head(dat)
```
Subset data.frame:

```
dat[c(1, 3, 5), c(10, 20)]
```

This subsets the 1st, 3rd, 5th rows and 10th, 20th columns of the data.frame that we loaded from the csv file

```
dat[1:4, 1:10]
dat[5, ]
```

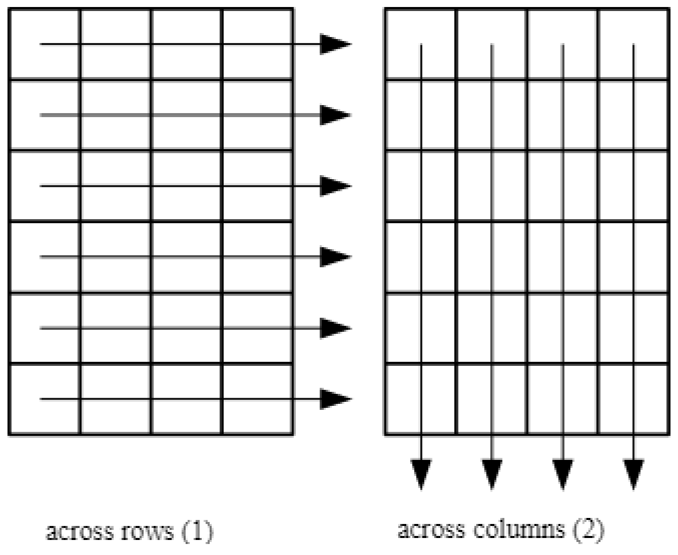
Calculations with the rows and columns of a data.frame

```
max(dat[2, ])
min(dat[, 7])
mean(dat[, 7])
median(dat[, 7])
sd(dat[, 7])
```

Summarise data.frame:

```
summary(dat[, 1:4])
```

Margins in apply function:



`?apply` - prefixing a '?' to an R function and running it will take you to the help pages

# Average inflammation of each patient

```
avg_patient_inflammation <- apply(dat, 1, mean)
```

# Average daily inflammation

```
avg_day_inflammation <- apply(dat, 2, mean)
```

Run the name of the variable to see what is stored in it:

```
avg_day_inflammation
```

Exercises

Subsetting More Data

Suppose you want to determine the maximum inflammation for patient 5 across days three to seven. To do this you would extract the relevant subset from the data frame and calculate the maximum value. Which of the following lines of R code gives the correct answer?

1. `max(dat[5, ])`
2. `max(dat[3:7, 5])`
3. `M0`
4. `max(dat[5, 3, 7])`

Marsha - 3.
Claire: 3
Clare 3
Hannah: 3
Claire Z: 3
Danial: 3
Valeria:3

Subsetting and Re-Assignment

1.

Solution:

```
patients <- seq(2, 60, 2)
days <- 1:5

dat2 <- dat
dat2[patients, days] <- dat2[patients, days]/2
dat2
```

Plotting in 'base R':

```
?plot
plot(avg_day_inflammation)
max_day_inflammation <- apply(dat, 2, max)
```

```
plot(max_day_inflammation)
min_day_inflammation <- apply(dat, 2, min)
plot(min_day_inflammation)
```

Creating functions (detour)

You can use functions to make coding quicker by putting all the repetitive bits into a one object which then you access (‘calling a function’)

```
fahrenheit_to_celsius <- function(temp_F) {
  temp_C <- (temp_F - 32) * 5/9
  return(temp_C)
}
```

Indentation in R - RStudio helpfully automatically indents code to make it ‘pretty’, i.e. more readable. But spacing and indentation doesn’t usually make a difference in R.

```
fahrenheit_to_celsius(32)
```

```
celsius_to_kelvin <- function(temp_C) {
  temp_K <- temp_C + 273.15
  return(temp_K)
}
```

```
# freezing point of water in Kelvin
celsius_to_kelvin(0)
```

```
fahrenheit_to_kelvin <- function(temp_F) {
  temp_K <- celsius_to_kelvin(fahrenheit_to_celsius(temp_F))
  return(temp_K)
}
```

Exercise

Create a Function

In the last lesson, we learned to **c**ombine elements into a vector using the `c` function, e.g. `x <- c("A", "B", "C")` creates a vector `x` with three elements. Furthermore, we can extend that vector again using `c`, e.g. `y <- c(x, "D")` creates a vector `y` with four elements. Write a function called `highlight` that takes two vectors as arguments, called `content` and `wrapper`, and returns a new vector that has the wrapper vector at the beginning and end of the content:

R code

```
best_practice <- c("Write", "programs", "for", "people", "not", "computers")

asterisk <- "***"  # R interprets a variable with a single value as a vector
                   # with one element.

# You need to create this highlight() function
highlight <- function() {
 Some stuff in here
 return()
}
# When you then run this, the desired output will appear if you’ve got it right :)
highlight(best_practice, asterisk)
```

Desired output

```
[1] "***"       "Write"     "programs"  "for"       "people"    "not"

[7] "computers" "***"
```

Marsha - finished
Hannah - finished
Claire Z - finished
Claire K - finished

Bonus exercise

If the variable `v` refers to a vector, then `v[1]` is the vector’s first element and `v[length(v)]` is its last (the function `length` returns the number of elements in a vector). Write a function called `edges` that returns a vector made up of just the first and last elements of its input:

R code

```
dry_principle <- c("Don't", "repeat", "yourself", "or", "others")
edges(dry_principle)
```

Desired output

```
[1] "Don't"  "others"
```

Marsha - finished

Testing, error handling:

```
center <- function(data, midpoint) {
  new_data <- (data - mean(data)) + midpoint
  return(new_data)
}


dat <- read.csv(file = "data/inflammation-01.csv", header = FALSE)

centered <- center(dat[, 4], 0)
head(centered)

mean(dat[, 4])
mean(centered)

sd(dat[, 4])
sd(centered)
```

NA - stands for missing data in R

Factors - categorical data, e.g. control, drug1, drug2 or group1, group2, group3 - more info here: http://swcarpentry.github.io/r-novice-inflammation/12-supp-factors/index.html

Warning - in R this indicates the output may not exactly be what you want, but R will continue executing code after a warning is encountered. In contrast, an **error** will stop execution and R will not continue when an error is encountered.

Exercise

Functions to Create Graphs

Write a function called `analyze` that takes a filename as an argument and displays the three graphs produced in the previous lesson (average, min and max inflammation over time). `analyze("data/inflammation-01.csv")` should produce the graphs already shown, while `analyze("data/inflammation-02.csv")` should produce corresponding graphs for the second data set. Be sure to document your function with comments.

Note: The RStudio plot panel only shows one plot at a time, so in order to show all three of these plots, put the following command at the start of your function
par(mfrow = c(1, 3))
This sets the parameter (par) for "multiple figure row-wise" layout to have 1 row and 3 columns.

You're aiming for something like:

```
analyze <- function(filename) {
  par(mfrow=c(1, 3)) # This tells R to display all three plots

  # Read in the data from the filename

  # Calculate the average daily inflammation
  # And plot it

  # Repeat for min and max

  par(mfrow=c(1, 1)) # This tells R to go back to the normal single plot display

# No need for a return statement in this one, the plots are the output!
}
```

Claire Z - done
Marsha - finished
Aislinn - finished
Claire K - finished
Hannah - finished

Solution

```
analyze <- function(filename) {
  # Display all 3 plots at once
  par(mfrow=c(1, 3))

  # Read in the file
  dat <- read.csv(file=filename, header=FALSE)

  # average daily inflammation
  avg_day_inflammation <- apply(dat, 2, mean)
  plot(avg_day_inflammation)

  # maximum daily inflammation
  max_day_inflammation <- apply(dat, 2, max)
  plot(max_day_inflammation)

  # minimum daily inflammation
  min_day_inflammation <- apply(dat, 2, min)
  plot(min_day_inflammation)

  # Return to default single plot
  par(mfrow=c(1, 1))
}
```

Function arguments - need not be specified by name; then the arguments are understood by R in the order of their **position**. If in doubt about the position order of arguments for a function, it's safer to use the names of the arguments!

Matching Arguments

To be precise, R has three ways that arguments supplied by you are matched to the *formal arguments* of the function definition:

1. by complete name,
2. by partial name (matching on initial *n* characters of the argument name), and
3. by position.

Arguments are matched in the manner outlined above in *that order*: by complete name, then by partial matching of names, and finally by position.

Exercise

Summing Values

Write a function called `total` that calculates the sum of the values in a vector. (R has a built-in function called `sum` that does this for you. Please don't use it for this exercise.)

R code

```
ex_vec <- c(4, 8, 15, 16, 23, 42)
total(ex_vec)
```

Desired output

```
[1] 108
```

Marsha - finished
Yangmei - finished
Aislinn - finished

Exponentiation

Exponentiation is built into R with the ^ operator:

```
2^4
```

```
[1] 16
```

```
```

Write a function called `expo` that uses a loop to calculate the same result.

R code

```
expo(2, 4)
```

Desired output

```
[1] 16
```

Processing multiple files

```
list.files(path=”data”, pattern=”csv”, full.names=TRUE)
list.files(path=”data”, pattern=”inflammation”, full.names=TRUE)
```

Q&A from Day 3

- When would you want to use <- versus = when assigning data to variables?
  - The use of the arrow operator <- is historic. In older keyboards (many decades ago now) there was actually an arrow operator key! R uses <- for assignment of variables and functions and = for arguments to functions. This was a design choice when the language S was created (R is a free and open source version of the S language).
  - Practically, most of the time it is equivalent. However, there are cases where you can get odd behaviour and it can be difficult to debug. Therefore, it is considered good R coding practice to stick to convention.
- Is temp_F arbitrary? i.e. you can call that argument whatever you want, and it will just pick up that whatever you put inside the brackets will be what is passed through the function?
  - Yes!
- Why am I not seeing any outputs from my function?
  - Remember the return()
-

Day 4: Thurs 2nd Dec - Programming with R (cont)

Instructor: Aaron Ceross
Helpers: Mcebisi Ntleki, Everlyn Kamau, James Robineau, Badran Elshenawy

Welcome and Introduction

- Getting to know each other
- Goals, expectations, and foolish assumptions

Data: http://swcarpentry.github.io/r-novice-inflammation/data/r-novice-inflammation-data.zip

Ice Breaker

Aaron: Baked Bread
Claire C - Mulled wine
Hannah - Baking
Everlyn - Lavender

Valeria:Roses
Yangmei: Lily
Claire - alcohol hand rub
Marsha - lavender
Mehpare - N/A

Data Structures

- Everything in R is an object
- Every object has a type
    - Six data types
        - Character
        - Integer
        - Numeric
        - Logical
        - Complex
        - Raw (this is not really something you will encounter in most uses of R,, it deals with byte values)
- Every object belongs to a class
- Classes define how information is contained and how it may be accessed
- Even functions are objects in R! This means you can use them in other functions and perform operations.

- There are a number of functions useful in telling us about the objects
    - typeof()
    - class()
    - attributes()
    - length()

Vectors

- There are no scalar values in R, everything belongs to a vector. A single value is a vector of length 1
- Atomic vectors have elements of all the same data types
- Atomic vectors can be coerced into different types
- If an atomic vector is being created with mixed data, R will try to coerce them to the same data type. It will do this silently!

Lists

- A special type of vector that can have heterogeneous data types
- Elements can be named!
- We access these elements using the $ operator on the list

One is a list and one is an integer

Exercise

Write down the length of xlist and the structure:

Length = 3
Str = List of 3

Length = 3
Structure = List of 3

```
length(xlist) # =3
str(xlist)
  # tells you the structure of each piece in the list
    #a is a character
    # b is integer
    # data is a data frame
```

Exercise

Write code to return the age and gender values for the first 5 patients.

```
dat[1:5, c(2, 5)]

dat[1:5, c("Gender","Age")]


sub_dat <- dat[1:5, c("Age", "Gender")]
dat[1:5, c(2,5)]
```

Exercise

Create a scatterplot showing BloodPressure for subjects not in the control group.

```
plot(dat[dat$Group != "Control", ]$BloodPressure)
plot(dat[dat$Group!="Control",]$BloodPressure)

plot(dat[dat$Group != "Control", ]$BloodPressure)
plot(dat[dat$Group!="Control",]$BloodPressure)
```

Exercise

In this dataset, values for Gender have been recorded as both uppercase M, F and lowercase m, f. Combine the addressing and assignment operations to convert all values to lowercase.

```
dat$Gender <- toupper(dat$Gender) #cheating

dat[dat$Gender == "m", ] <- "M" # not cheating
dat[dat$Gender == "f", ] <- "F" # not cheating

dat$Gender[dat$Gender=="M"]<-"m"
dat$Gender[dat$Gender=="F"]<-"f"

dat$Gender[dat$Gender=="M"]<-"m"
dat$Gender[dat$Gender=="F"]<-"f"

tolower(dat$Gender
dat$Gender[dat$Gender=="f"]<-"F"
dat$Gender[dat$Gender=="m"]<-"M"
```

Factors

You have a vector representing levels of exercise undertaken by 5 subjects

“l”, “n”, “n”, “i”, “l” ; n=none, l=light, i=intense

What is the best way to represent this in R?

a) exercise <- c(“l”, “n”, “n”, “i”, “l”)

b) exercise <- factor(c(“l”, “n”, “n”, “i”, “l”), ordered = TRUE)

c) exercise < -factor(c(“l”, “n”, “n”, “i”, “l”), levels = c(“n”, “l”, “i”), ordered = FALSE)

d) exercise <- factor(c(“l”, “n”, “n”, “i”, “l”), levels = c(“n”, “l”, “i”), ordered = TRUE)

D.
D
D
D

Project Setup

Project_folder
- /data
- /output
- /R
- analysis.R
- /figures
- /report

Places to get help
Stack Overflow (Cross-validated)
RStudio Discussion