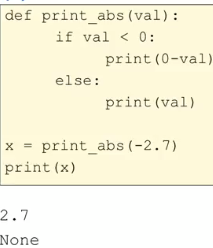
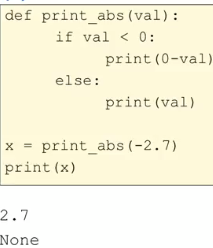
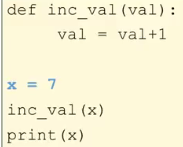
***Background in Python + Unix***

*Python*

* Jupyter 🡪 originally the combination of 3 languages, Julia, Python and R.
* These days, Jupyter supports over 40 programming languages.
* NumPy and Pandas libraries are incredibly valuable for doing data science in Python.
* Python is powerful + surprisingly fast for **interpretive language,** which aregenerally slower b/c they'd run on top of an **interpreter** rather than being compiled directly from the machine on which they're running.
* Think of this as having a middle man between the code + the machine, which isn't there for a language like C or C++.
* Python works well with other languages + is often used as a **blue language**, running in between components written in other languages.
* Python plays well with other languages in Jupyter notebooks as well.
* Python, because it's interpreted, can run everywhere w/ Python installed
* B/c Python is open source, you can install it anywhere w/out worrying about licenses.
* Python has a number of features, like **dynamic typing** (no explicit types needed for variable **declaration**) + automatic memory management, which make it both easy to learn + read
* Everything in Python is an **object**
* “x = 3” causes a PyIntObject to be created, which hold the value of the object, in this case 3, along w/ other details for Python to work w/ under the hood (including the type of object, # of references to the object, etc.)
* When you say x = 3, Python creates a PyIntObject with a value of 3 + have your variable x point directly to that object.
* x is created on the **stack** (holds local values + is managed by the program) + the PyIntObject is created on the **heap** (holds dynamically-created data + is managed by the OS)
* If we then say x = 4.5, The **garbage collector** in Python automatically frees the space associated w/ that 1st PyIntObject b/c nothing is pointed to it anymore, and now we have a PyFloatObject
* If ever curious to know if 2 variables are pointing to the same object, use the **is** command (False if x points to a PyIntObject and y points to a PyFloatObject)
* To test for *numeric* equality, use the **==** operator.
* x.lower() 🡪 lower **method**
* Strings CANNOT be changed 🡪 using x.lower on a string returns a new value, but the x variable still holds the original value
* To change it, re-declare x w/ the new value 🡪 x = x.lower() 🡪 x now points to new lowercase object
* If a function doesn't return anything, it effectively returns type None in Python.

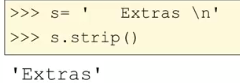
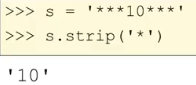
 



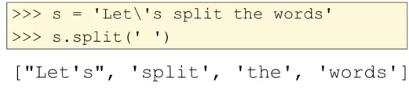
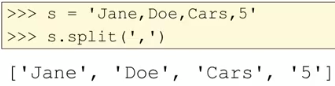
* This returns 7 b/c x is assigned the value 7 🡪 variable x is created + *points* to an PyIntObject w/ value 7
* When we call our function on the argument x, a *new* variable **val** is created, which points to the *same* PyIntObject as x (**copying the reference**) + they’re NOT the same variable
* Val is a *copy* of the pointer = points to same value as x
* When assigning val = val + 1, we are reassigning val to point to a new PyIntObject with value of 8, so *x has not changed at all + when we print x, we get the original value*
* When function inc\_val ends, out **temp variable** val is terminated, and only x is left, unchanged
* Never want a **global var** to have same name as a function **parameter**, though many places do not recommend using global variables *at all*

*String Functions*

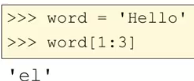
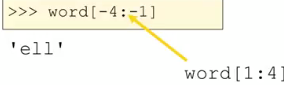
* Strings are **immutable** = cannot be changed 🡪 assign return values of calling methods on strings to new variables to save changes (or use In place = true for some)
* .**lower**(), .**upper**(), .**capitalize**() [1st char is capital]
* Concatenate w/ plus signs, replicate with asterisks
* **strip**(s[, chars]) 🡪 removes leading + trailing chats 🡪 omitting chats arg removes whitespace

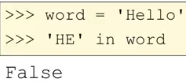
* Can **split** into smaller substrings

* **Slice** to get specific chars (w/ negative indices being equal to length of string – arg (5 – 4 = 1)

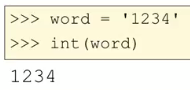
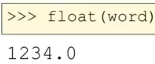
* Tell if substring is w/in string

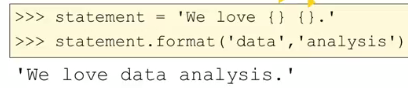
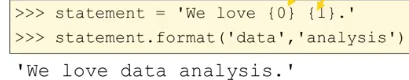
* Use find to find *where* substring is (get start indice)

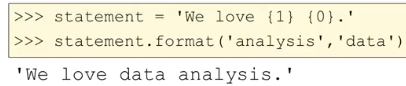


* Convert to numbers w/ int() and float(), but only if the string is a number

* Want to insert strings into larger string w/ format() + its placeholders



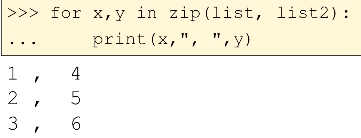
*Lists*

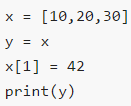
* Like arrays 🡪 resizable + has array implementation under the hood



* 1st loop is easier, more readable, less error-prone
* Are **mutable** 🡪 can be changed
* **.append()** 🡪 add elements to end of list
* **.pop(i)** 🡪 removes element at index i + returns it as well
* All remaining indices occurring after this index i shift to the left (become 1 less)
* Can remove specific values w/ **.remove(xx)**
* Can add 1 list to the end of another (merge) w/ **.extend(list2)f**
* Append 🡪 adds entire list as 1 element, extend 🡪 adds each list2 element as own element
* **.zip(list1,list2)** creates a paring of 2 lists 🡪 good for working w/ multiple lists at the same time



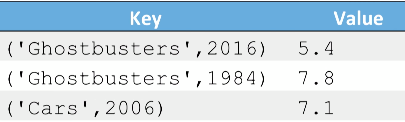
* For each x in list1 each y in list2, print x comma y
*  🡪 returns [10, 42, 30] b/c y points to same list object as x (not separate entities)
* i.e. we set up a list and made x point to it, then y points/refers to the *same list in memory­*
* x[1] = 42 changes the list in memory that both variables are pointing to
* To make y refer to its own copy of this list, do **y = list(x)**

*Tuples*

* **Tuples =** way of storing data in *immutable* manner 🡪 valuable for situations in data analysis
* Usually used to put together info that is in some way connected (like a record in a data frame)
* 
* **Notice multiple data types**
* Can index + loop over tuples like lists, but cannot change elements
* This is important b/c we can trust it to never change 🡪 Immutability is important in 2 key ways:
* Data which can change makes **parallel computing** (a key facet of data science) harder b/c we have to make sure everyone working on the problem sees those changes
* If data *can’t* change, we can just make copies of it to CPU nodes + no one has to worry about it changing on them while they’re computing
* Tuples can be used as the **keys** in **dictionaries**
* So the dictionary can organize based on the initial key value w/out worrying about a key being changed by someone w/ a reference to it data which can change

*Dictionaries*

* 1 of the most useful Python data structures 🡪 has **key-value (KV) pairs**
* Can ask if a key is in a factionary, and look up a value by its key (dictionaries are very fast at lookups)
* Only restriction on types 🡪 key must be immutable (value could be a list, or even another dictionary)
* *Any object is fine as a value in a KV pair, and everything in Python is an object*
* Can use a tuple as a key (b/c they’re immutable) to make unique keys

’



* Use keys as indices



* Can use **len()** to see how many KV pairs are in a dictionary
* To add to a dict, just write a new key as the index + assign it a value



* Would reassign this value if this key was already in the dict
* Dict’s are an **unordered collections** 🡪 no inherent ordering + we can’t trust its own internal ordering not to change at any time
* They’re very fast b/c they’re unordered
* If we need data ordered, dictionaries might not be ideal for the problem



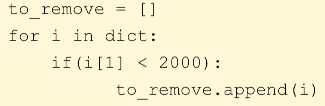
* Be careful using bracket access to get values b/c if the key doesn’t exist, we get an error
* 2 approaches mitigate this:
* Use **.get()** 🡪 if key doesn’t exist, get back value **None (test w/ x == None)**
* Test w/ **in 🡪** 
* Use these 2 methods to see if a key is in a dict before trying to access it
* Can return + remove items w/ **dict.pop(key)** 🡪 **mutates** the data structure such that the KV pair is no longer there
* Can also just use **del dict[(key)]** if we don’t want the value returned
* Can iterate over elements in dictionary
* Get all keys:



* Get all KV pairs:



* **.items()** returns a list w/ key and value
* *Cannot mutate a dict while iterating* 🡪 bad habit in any programming language to mutate a data structure while iterating through it b/c the underlying data structure implementation may change its structure of the data when adding/removing from it
* Ex: want to loop through a dict and remove everything meeting certain criteria 🡪 2 steps
* Find keys matching criteria via a loop 🡪 add that key to a list



* Remove items by iterating through the *list*, NOT the dict

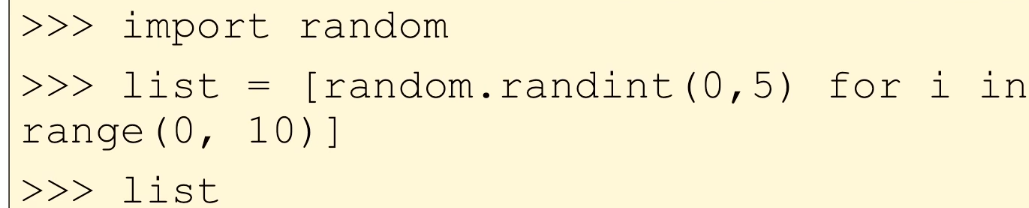


*Comprehension*

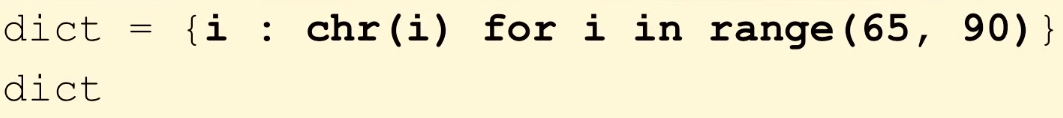
* Can builds lists + dictionaries easily w/ **comprehension**
* Ex: list of all squares from 1-10:

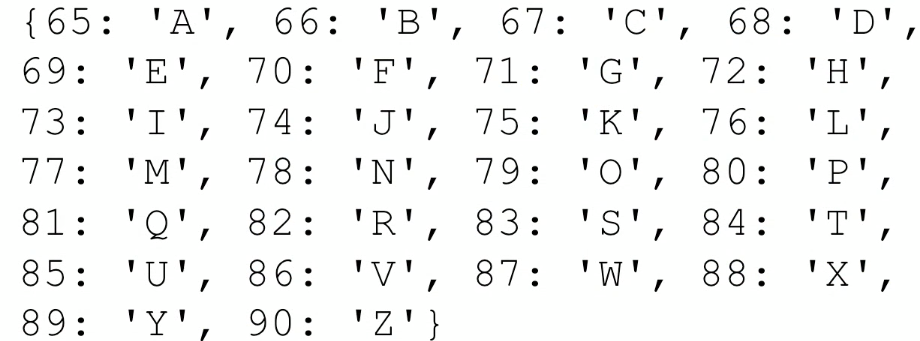


* Range(1,11) provides i w/ values 1-10, and we square each + store in the list



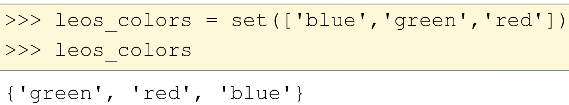
* Get a random int between 0-5 ten times
* Can build dictionaries as well, but while specifying both the key and the value
* Ex: ASCII chars w/ corresponding letters:



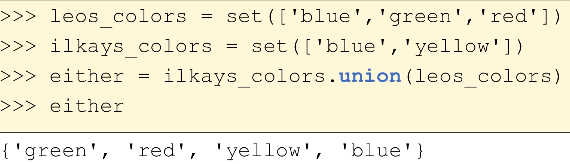


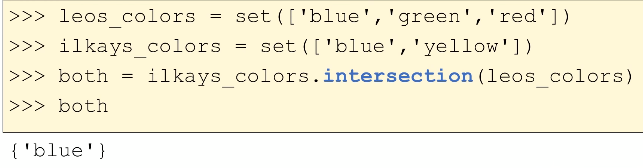
*Sets*

* **Sets =** support a # of useful math operations + only allow *unique* elements
* 3 useful qualities
* Unordered like dictionaries 🡪 allows them to be fast for a of key operations
* Hold unique elements 🡪 no dupes
* Support useful set operations like **UNION** and **INTERSECTION**
* **Pass lists into set() as an arg**



* Notice results are in a different order b/c sets are unordered
* Add to set w/ **set.add(‘value’**), so long as value is unique
* Remove w/ **.remove(‘value to remove’)** or **.discard(‘value to remove’)**
* Discard is better b/c if value isn’t there, it dose nothing, while .remove() causes an error
* **Set1.union(set2)** gives unique results out of both of the sets

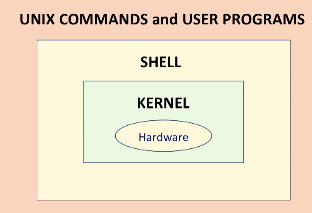


* Useful to find ALL unique items in 2 separate sets
* Can also do this w/ **set 1 | set 2** 🡪 OR operator
* **Set1.intersection(set2)** finds all unique items that are in BOTH sets
* 
* Can also do this w/ **set 1 & set 2** 🡪 AND operator

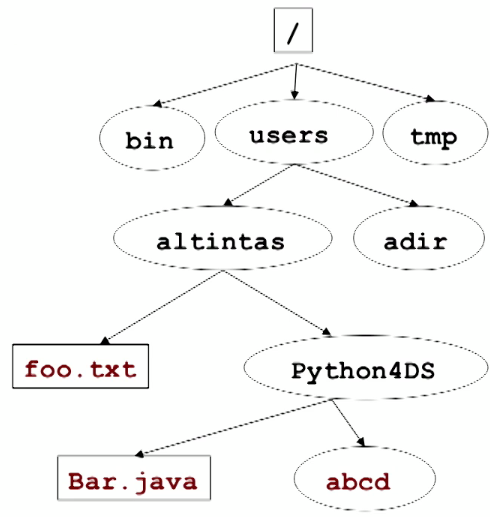
***UNIX***

*Intro*

* Most OS’s today are built on top of UNIX, except those based on Windows OS
* Ex: Various Linux distributions, Mac OSx, iOS, Android
* Unix is widely adopted in industry + the backend of many data + computing systems utilize UNIX, such as those at Facebook + Google
* Unix provides a v. powerful dev environment built upon the composability of **commands =** small utility programs that do 1 thing very well + are executed by the **shell**
* Users can build their own commands + build programs using these commands + **scripting**
* Unix is relevant to data science b/c in addition to being a powerful OS, it provides commands for data search, subsetting, + transformation
* Through effective use of commands on the command line, we can do quick manipulation + analysis of data, which is v. helpful in exploratory analysis + data prep
* Can also chain multiple commands together w/ a **pipeline**
* Also, many DS tools come w/ a **command line interface (CLI)** which requires interaction w/ the CL + executes through the Unix **shell**
* 3 main parts of the Unix OS:
* **Programs > Shell > Kernel (containing hardware)**

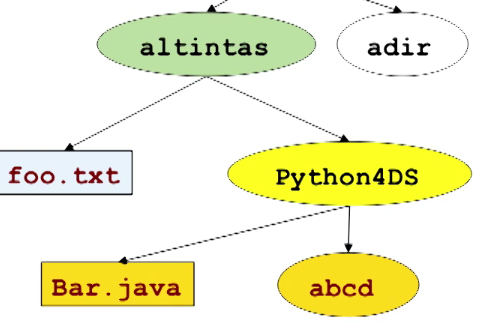


* **Shell** = interface between user + **kernel** that acts as the CL interpreter for Unix
* Allows user commands + other user-developed programs to be executed by the kernel
* Automatically starts whenever a user logs into a Unix-type system
* Accepts commands + makes **system calls** for those commands to be executed
* Also provides a programing/shell scripting interface w/in the shell environment
* Each time a user sends command to the shell, the shell communicates the command to the kernel which in turn creates a process w/ a unique identifier to run it
* *\* Everything in Unix is represented by a* ***process*** *or a file, even hard drive volumes*
* This is why there is no concept of **name drive** like in Windows
* **Process =** an executing program IDed by a unique process identifier (**PID**)
* **Files** = collections of data created by user running process that are organized in a **directory** (special files that contain other files) structure that
* **Directory structure =** a tree **hierarchy** made up of **paths** to specific files
* **Paths** = separate each node of the directory tree via a slash character starting at the **root directory**





* **Absolute path** – can be reached from anywhere in the Unix system
* **Relative (to the WD) path** – can be reached via the **working directory (**ex: altintas)



* *\* Be careful to check which dir you’re in when working w/ relative paths via* ***pwd (print wd)***
* Can reach foo via **foo.txt** or **./foot.txt** where **./** = refers to the wd
* To access foo.txt from wd of Python4DS, we use **../foo.txt** where **../** refers to the level above our current wd (**parent directory)** so we don’t have to change our wd
* Each Unix dir has this special utility to reach the/refer to its parent dir
* TILDE ~ refers to the **home directory** which can reach any file under the home dir (ex: users)
* ~/foo.txt or ~ altintas /foo.txt

*Common Commands*

* **cd** = change dir
* **cd ~** = go to home dir **cd ..** = go to parent dir **cd .** = go to cwd
* **ls** = list files in current dir
* **ls -** 🡪 list al files in *long form* (who created them, what time, how big they are, and access rights)
* can ls . and ls .. to see contents of cwd and parent of cwd
* **ls – a** 🡪 shows hidden files + dirs.
* **ls -la** 🡪 who all files + dirs. including hidden ones in long form
* **ls path/path/folder** 🡪 list all files in certain dir
* ls /Users/altintas/ ls ~ ls ..
* **clear** = clears current shell screen
* **cat ‘file name.txt’** 🡪 shows text w/in that file
* **man ‘command name’** = show manual page for a command
* **mkdir ‘folderName’** = create a dir
* \* = a wildcard char
* **ls \*.txt** 🡪 show all files w/ txt extension
* **cp ../fruits.txt .** 🡪 copies fruit text file from parent dir. to cwd
* **mv ../fruits.txt .** 🡪 moves only copy of fruit text file from parent dir. to cwd
* ls .. 🡪 no longer see fruits in that dir
* 3 file descriptors that the shell starts when they start up, which are always ready for commands to use
* **Standard Input (stdin)** = default source for a user to enter inputs (usually the keyboard)
* Could be switched to another file via a file redirection
* **Standard Output (stdout) =** where programs output/print their functional/primary outputs
* Usually to shell/terminal (default\_ but can be switched to another process or file via a file redirection
* Should be kept for the correct outputs of a command/program
* **Standard Error (stderr)** = output channel for error messages or any other non-functional outputs coming out of programs (like **trace print statements**)
* Default is also terminal window

*Redirecting Standard IO*

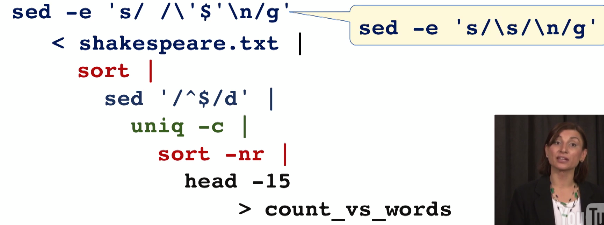
* In Unix + Unix-like systems, each process started from the CL has 3 **file descriptors** associated with it
* **FD 0** for stdin 🡪 normally connected to keyboard
* **FD 1** for stdout 🡪 normally connected to terminal from which application was launched
* **FD 2** for stderr 🡪 normally connected to terminal from which application was launched
* file descriptors can be *redirected* + connected to other files, IO of other process, and more
* To redirect:
* stdin to come from a txt file, use **command < filename**
* stdout to go to a txt file, use command > file1
* or explicitly define the file descriptor **command 1> file2**
* stderr to go to a txt file, explicitly use the file descriptor in **command 2> filename**
* **$ command 2> file4 > file5** 🡪 redirects command’s output to file 5 + its error messages to file 4
* **$ command > file1 2> file2 < file3** 🡪 get stdin from file 3, redirect command’s output to file 1 + its error messages to file 2
* Can see redirections don’t need to be in order so long as the redirection operators is kept w/ the correct file name
* Even the command name can come last
* Doubling these operators will **append** to a file instead of *rewriting*
* *Command >> file1* 🡪 we redirect stdout to file 1 but instead of rewriting to it, we append to it
* *Command 2>> file1* 🡪 we redirect stderr to file 1 but instead of rewriting to it, we append to it
* Command > file 2>&1 🡪 redirect both stdout + stderr into *same* file
* **more *filename.txt*** 🡪 only displays how much of the text file that will fit into the Unix shell
* **cat *filename.txt*** 🡪 returns whole file
* **sort fruits.txt** 🡪 sorts contents in alphabetical
* save it in another file via redirection **sort fruits.txt > fruits\_sorted.txt**
* check the file w/ **cat < fruits\_sorted.txt**
* Get only unique lines w/in a file w/ **unique fruits\_sorted.txt**
* Save **unique fruits\_sorted.txt > fruits\_unique.txt**
* check the file w/ **cat < fruits\_unique.txt**
* To count # of lines in a file (i.e. # of unique fruits in unique\_fruits.txt) do **wc –l** for *word count –lines*
* See who is logged into a Unix system at the time use **who**
* Can save this info w/ **who > names.txt**
* Can chain commands (**composite command)** via a semicolon 🡪 **pwd; ls –l**
* Chained commands outputs can be saved but the chained commands must be w/in parenthesis
* **(pwd; ls –l) > output.txt**
* Can suppress redirected outputs by sending them to a **null device** 🡪 won’t be seen in stdout (terminal screen) or in a file

*Pipes and Filters*

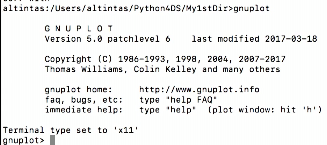
* Can **pipe** outputs of commands to other commands instead of doing multi-step work like above
* Stdout of 1 command into stdin of another
* **Filters** can transform what is piped into new commands
* **Filters =** programs placed after a Unix pipe that take inputs (the outputs from a previous command) + transforms them in some way
* **grep** 🡪 searches for lines containing a given string or a pattern w/in a given input stream
* **more/ less** 🡪 shows as much as what fits in the shell window
* **sort**
* **uniq**
* **Cat foo.txt | wc** 🡪 pipes output of *cat* into *wc* (word count)
* In most cases, the receiving commands will be triggered to run w/out taking full advantage of the output of the previous command
* **ls –la | more** 🡪 shows as much details for all files including hidden files that show in the shell
* go through the rest of the lines via the space bar
* **cat file | wc 🡪** displays lines, words, and characters in the file
* **man cat | grep file** 🡪 looks for the string “file” w/in the manual page for cat command + returns only those lines
* **ls –l | grep txt | wc** 🡪 looks for the string “txt” w/in all returned files in long form + returns those + *then* returns the # of lines from the grep output which = the # of text files in the CWD
* double pipes are used for a sort of small script
* **who | sort > current\_users.txt** 🡪 get all current users logged into the Unix system and sort them and place this into a new file
* Ex:
* **sort fruits.txt | uniq | wc –l** 🡪 all unique lines from the sorted list of fruits from fruits.txt
* **man cat | grep file | wc –l 🡪** counts how many times the word “file” occurs in the cat command manual page
* **ps** 🡪 command to report process status w/in all terminals
* In every Unix-based system, there's a process w/ **process ID, PID**, = 1, which becomes a parent process, **PPID**, for many other processes.
* PID 1 is the **grandparent** of all processes in the Unix system,
* The kernel (root) itself, has PID = 0
* In addition to a unique PID, each processes also have a PPID which tells which process started it.
* PPID = PID of the process's parent.
* Parent processes start new processes as **sub processes**.

*Useful Commands for Data Science*

* Unix commands, when chained together, facilitate complex data manipulations.
* Filtered commands, in general, give data analysts a quick way to inspect + transform data.
* When working w/ txt files or Unix command outputs, we mainly do txt manipulation and/or searches
* **cat, grep, wc, sort, uniq**
* **head/tail 🡪** list 1st/last n lines of txt file/input stream
* **head -5 fruits\*** 🡪 get 1st 5 lines from all files that start w/ “fruits”
* **(head -2 \*fruits\*, tail -2 \*fruits\*) | cat**
* **(head -2; tail -2) < fruits.txt | cat** 🡪 get 1st + last 2 lines from fruits.txt
* **cut** 🡪 can cut a portion of each line of a file
* **paste** 🡪 merge contents into 1 file
* **paste \*fruits\* > all\_fruits.txt** 🡪 puts contents of each file next to each on same line (1st line of 1st file, then 1st line of 2nd file until all files done, then onto 2nd line)
* good for creating tabular data + therefore good for data science
* **sed 🡪** stream editor used to perform basic txt transforms on an input stream
* **find 🡪** quick searches in the file system/hierarchy
* Ex: What are the top words in Shakespeare’s works?
* **sed –e ‘s/ /\ ‘$’ \n/g’ < Shakespeare.txt | sort | sed ‘ /^$/d’ | uniq –c | sort –nr | head –l5 > top\_shakespeare\_words.txt**
* take input stream of Shakespeare.txt into **sed** + convert each space (s/) between words into a new line character (s/ into \n), since each line has many words
* Then we have a stdout of words as their own lines, with some blank lines that existed prior (between paragraphs, for example)
* Sort this stdout + then remove the remaining blank lines (‘ /^$\d’)
* Then find the unique words + include their counts (-c)
* Do a numerical sort (sort –nr) + get the top 15 lines from this sort command + place into file



* Which users run the most processes on the Unix system?
* **ps –aef | cut –c3-5 | sort | uniq –c | sort –nr | head -3 > top\_users.txt**
* get all details from all processes in all Unix terminals
* then cut out the user ID’s and sort them 🡪 cut out + return characters 3 through 5
* Count how many times each unique ID occurs, sort them numerically + store the top 3
* Transform fruits.txt into all caps for further processing
* **tr ‘[a-z]’ ‘[A-Z]’ < fruit.txt > fruit\_all\_caps.txt**
* **tr** transforms/converts lowercase letters into uppercase
* *This is much faster and done more efficiently w/ Python compared to Unix*
* We will then use **gnuplot** tool to plot our exploratory analysis findings



* Set the **term** to PNG to save the output image from the graph into a PNG file



* Name the output file



* Can set width of bars, or set to histogram syle, and give it a solid fill and more style formatting



* Now plot the Shakespeare word count file



* Exit gnuplot and check the directory for the PNG



