Lecture stuff Monday

Cmd1 data

Cmd2 output\_data

🡪 not a good approach since cmd1 could not work or pass along corrupted data

better:

cmd1 data && cmd2 output\_data

&&: looks at the **exit code** of cmd1

or cmd2 data; cmd2 … works as well but not in case of dependencies

Be a good UNIX citizen:

sysexit(:)

exit code #Put your error code here

echo $HOME (print home dir)

echo $? (exit code of the last run program)

3 Pillars of UNIX:

stdin (0): “the keyboard” input

stdout (1): “the terminal” output

stderr (2): “also the terminal” used for errors/warnings

Piping connects stdout from one process stdin to another e.g.

Sort data | head

Redirection!

myprog <infile > outfile

Always write warnings to stderr (sysstderr in python)

Raw2pure infile | analyse.py | present\_graph

Drawbacks here: in piping write stderr so you get feedback at several points of your pipeline so you could even choose to ignore a warning or react to it

Looking for ‘Lasse’ in output

grep Lasse output

analyse | grep Lasse > good stuff

How to catch patterns from stderr?

Noisy\_cmd input 2>&1 > outputfile | grep Lasse

2>&1: Redirects stream2 (stderr) to stream1 (stdout)

Get rid of noise!

Noisy\_cmd input 2> /dev/null

/dev/null is kind of a black hole

One general problem with unix is that everything is based on text files which can be annoying when dealing with a lot of data. This adds parsing steps and temporary files to the pipelines and makes it unnecessarily messy in times.

Iteration

For loop:

For x in a b c; do

Echo $x;

done

Commonly you want to loop over a large number of files or items so you would use:

File globbing:

For xx in \*.fasta; do

Analyse $x &; #the & here means put the job in the background

done

for i in $(seq | 10); do

echo $;

done

for i in $(ls \*.fasta); do

…;

done

coming back to using & e.g.

$ emacs & 🡪 makes emacs run in the background and gives you immediate control over the terminal again

$ jobs # gives you an overview over the current jobs

$ fg # brings emacs to the front

ctrl-z

$ bg #put latest haltet job in the background

Ways to write a shell script:

Source filename.sh

Imagine a file like:

heads.sh

For f in $\*; do head $f; done #$\* refers to all args, $1 to the first arg, $2 to the second arg

**#!/bin/bash** tells unix the the script is a stand-alone program and will be run as a subbash

but you need to make a file executable for that (in case it is not) and do it with:

$chmod +x heads.sh #this makes the file executable (for everyone in the system)

Collect executables in bin directories!

Instruct the operating system to look for your programs in your bin directories or the specific ones you want it to look in

Echo $PATH gives you all directories where the OS looks for executable scripts. To manipulate that one can do the following:

export PATH=$PATH:~/bin #add my bin directory to the PATH

for long term: use ~/.bashrc or bash\_profile

Lecture stuff Tuesday: Version control

Long ago:

Classical version control system is a data base containing one’s source files that come in various versions and all this is placed in a central repository placed on a big server.

A set of users start working with the files (each sort of has an image of this). They basically make a snap version of that.

Then:

Git (slang for being an idiot) follows the idea that a central repository might not be optimal. So, it introduced a distributed repository which means that each user holds its own repository. And still all users have the so called staging area that you can sync with the repository and other people’s repositories. And there is still have a big repository like GitHub (framesoftware).

Git add – add a file to your staging area

Git commit – commit from staging area to repository

Git diff – check for differences between versions

Git status – check status of files in the staging area

Git push – copies changes from local repository to the remore one

Git pull – copies a remote repository to a local repository

Git init – create a new repository/ initiate a new repository

Git clone – copies an entire remote repository

Cat gpcr.tab | gawk ‘{print NF}’ | sort –n | uniq –c

Tail –n +2 gpcr.tab | cut –f6| sort | uniq | wc –l #get number of organisms

Tail –n +2 gpcr.tab | cut –f6| sort –u| wc -l

Sed 1d gpcr.tab| cut –f6| sort –u| wc –l

A couple of good one-liners:

Sed (use it to replace one pattern with another, character by character or line by line):

Cut f1 | Sed ‘s/s/f/g’

awk condition {code}

cat gpcr.tab | gawk ‘/human / {print $1}’

$0: whole line $1: first field $NF: last field

cat gpcr.tab | gawk ‘/human/ {print $4 “\t” $3}’ > out.tab

gawk –F’,’ # tells it to expect a different delimiter than usual (usually tab)

print length ($5) #how many characters are in field 5

Assignment 1b discussion:

1.Cat gator | grep –c ‘>’ | wc –l

Cat gator.fa | gawk ’/>/{print NF}’ | wc –l

Cat gator.fa | gawk ‘/^>/{a = a + 1}END{print a}’

2.cat gator.fa | grep –v ‘^>’ | tr –d “\n” | wc –c (or wc –m)

3.cat gator.fa |gawk ‘/^>/{print(“\n”, $0)} /^[^>]/{printf(“%s”, $1)}’ | sed 1d

gawk ‘/^>/{print a; a = “”; print $0; next}{a=a$0}END{print a}’

4. cat gator.fa |gawk ‘/^>/{print(“\n”, $0)} /^[^>]/{printf(“%s”, $1)}’ | sed 1d | grep –v ‘^>’ | grep ‘[KR]$’ | wc –l

cat protein.fa | gawk ‘/^>/{print seq; swq = “”; next}{seq=seq $0}END{print seq}’ | gawk –F’[KR]’ ‘ {for (i = 0; ++I <= NF;) if(length($i)>6) print $i}’

parsing field data

1.tr –d “\”” or ‘”’

sed ‘s/”//g’

2.| awk –F , ‘$4~/Mitochondria/&&/$4~/Cytosol/{print}’ | wc –l

3. | awk –F , ‘$4~/Mitochondria/&&/$4~/Cytosol/{print $2}’

composition assignment4:

how to think:

input seq

output [A, C, G, T] [0-1]

unit testing: what input does not work (input only ACGT)

if it adds up to 1, check empty string, check if wrong input, and do easy manual check ups

instead of UNION one could also use the following to combine queries:

SELECT s.quant, p.person FROM Survey s, Person p WHERE s.quant = p.quant;

SQL:

3.Select accession from protein where length(sequence)>1000

4.Select distinct s.name from familymembers fm, protein p, species s where fm.protein=p.accession and fm.family=’NHR3’ and p.species=s.abbrev;

4b. or use a subselect

select \* from

(select \* from protein where….)

Will create a new table

5.Select species, count(sequence) FROM protein GROUP BY species;

5b. select s.name, count(\*) from protein p, species s where p.species = s.abbrev group by s.name order by count(\*) decs;

MySCL:

Select \* From x limit1 #would give you only the header

Assignment:

3.select count(gene\_id) c, biotype from gene group by biotype order by c desc;

4.select g.description, count(transcript\_id) transcripts from gene g, transcript t where g.stable\_id in (‘ENSG00000012048’, ‘ENSG00000139618’) and g.gene\_id=t.gene\_id group by g.description;

5.

Independent project:

Apply your new skills in independent project. Use best practices (defined by Stafford Noble)

Requirement:

Lab notebook (contains commands, details, software commands, temporary results) – PDF

Report (simple, short, ‘journal structured’) – PDF

Link to your code (GitHub or other repositories)

Example projects:

1.Alignment trimming(Data🡪MSA🡪(trim)🡪MSA’🡪(infer)🡪tree

trimming is supposed to remove noisy data!: tools GBlocks,TrimAL(has data)

Does the alignment trimming work? Are phylogenies actually improving?

2.Are protein domains preferably within exons?

Is there an evolutionary advantage for protein domains to reside within exons not breaking exon borders?

Data at Ensembl’s BioMart

3.Signal peptide prediction

Using old ‘Lukas-data’. Lukas wrote Phobius a software prediciting transmembrane regions.

Use: SciKit Learn module in Python

Deadline: 20th of October.

Steve McConell: Code Complete

Rules of thumb:

A function should do one thing and do it well.

Separate computing (calculations) and interaction (read, write, ask user)

Any unit of code (e.g. a function) should fit one single screen

Choose informative names

A function should only depend on its parameters (avoid global variables)

Consider your return values! (e.g. don’t return numbers as strings, avoid “special values” – use exceptions instead, or use ‘None’, use multiple return values)

Make and Snakemake:

For f in \*.fasta; do out=$(basename $f) out=${out% .\*} out=$out”\_aligned.fasta” muscle –in $f –out $ out; done

This is not the best solution since it needs tests and might have dependencies and bears a few caveats

Solution: make (create a rulefile called Makefile, 1976)

F1.pdf: f1.tex

Pdflatex f1 # run this command to make the file f1.pdf that depends on f1.tex

F2.pdf: f2.tex

Pdflatex f2

F3.pdf: f3.tex

Pdflatex f3

$ make

pdflatex1

even better:

%.pdf: %.tex

pdflatex $^

$make f1.pdf

pdflatex f1.tex

even better:

all: f1.pdf f2.pdf f3.pdf f4.pdf

%.pdf: %.tex

pdflatex $^

$make

pdflatex f1.tex

This is a very simple example! In terms of pipelines one could start thinking about where the tex file is generated from so add something like:

%.tex: %.dat

cmd $^-o $@

For example Abyss assembler is a makefile.

Downside of makefile: it is not so good for databases since it assumes simple dependencies and a very straight forward pipeline. Hence, it is not good to deal with a less stringent data flow.

Therefore, a variety of workflow languages exist like Snakemake, Common Workflow Language, Taverna, Galaxy, etc…

Pick one people are using around you to get help if needed ;) Galaxy has a GUI also

Snakemake:

Python based: configure, extend, and modify using python

Works at Uppmax

Example:

Running BWA with Snakemake

Rule bwa\_map:

Input:

“data/genome.fa”

“data/samples/{sample}.fastq”

output:

“mapped\_reads/{sample}.bam”

shell:

“bwa mem {input} | samtools view –Sb - > {output}”

Run with:

$ snakemake mapped\_reads/A.bam

There is a snakemake tutorial: snakemake.readthedocs.io (snakemake.bitbucket.org)