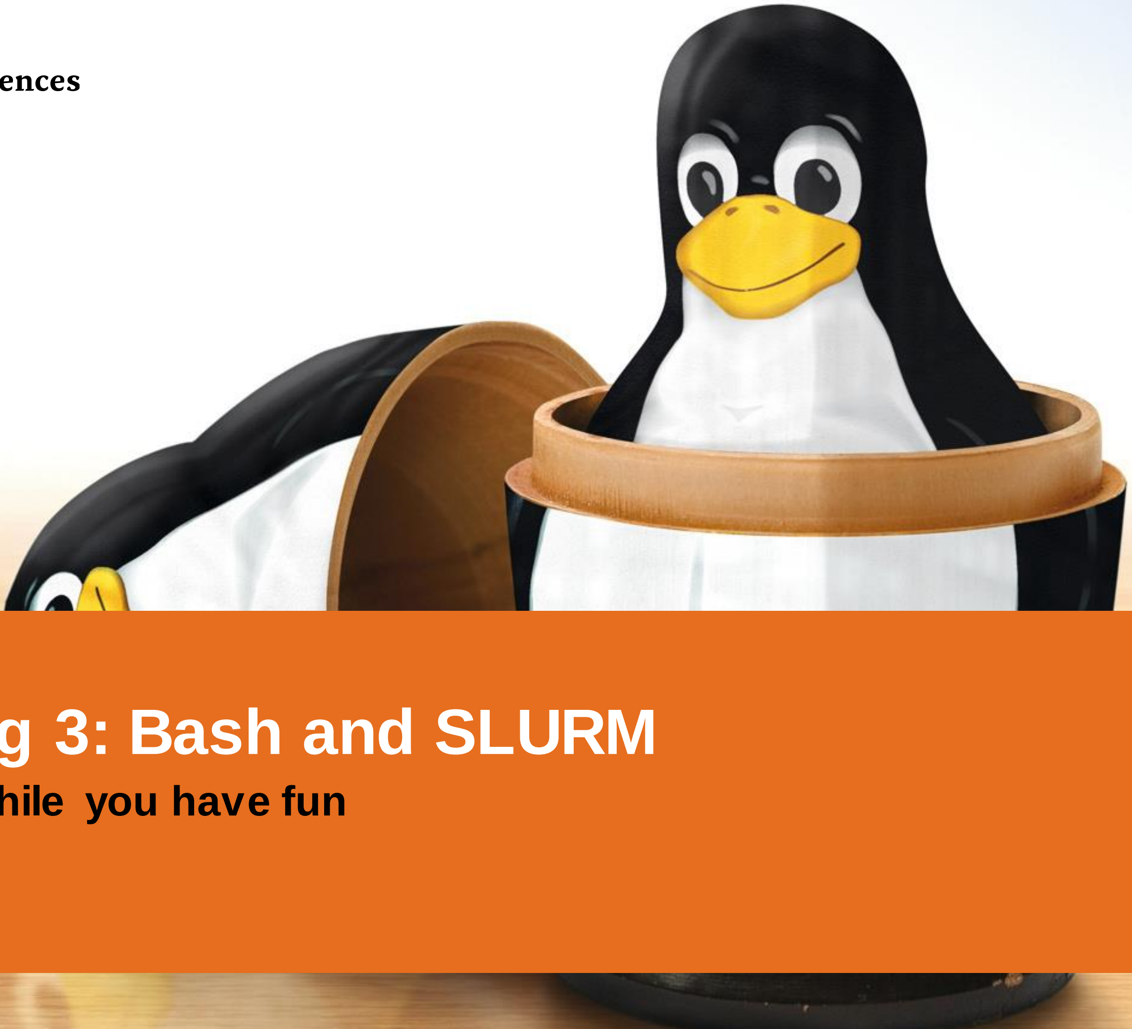
Hanze
University of Applied Sciences
Groningen



Programming 3: Bash and SLURM

Making stuff run while you have fun

share your talent.
move the world.

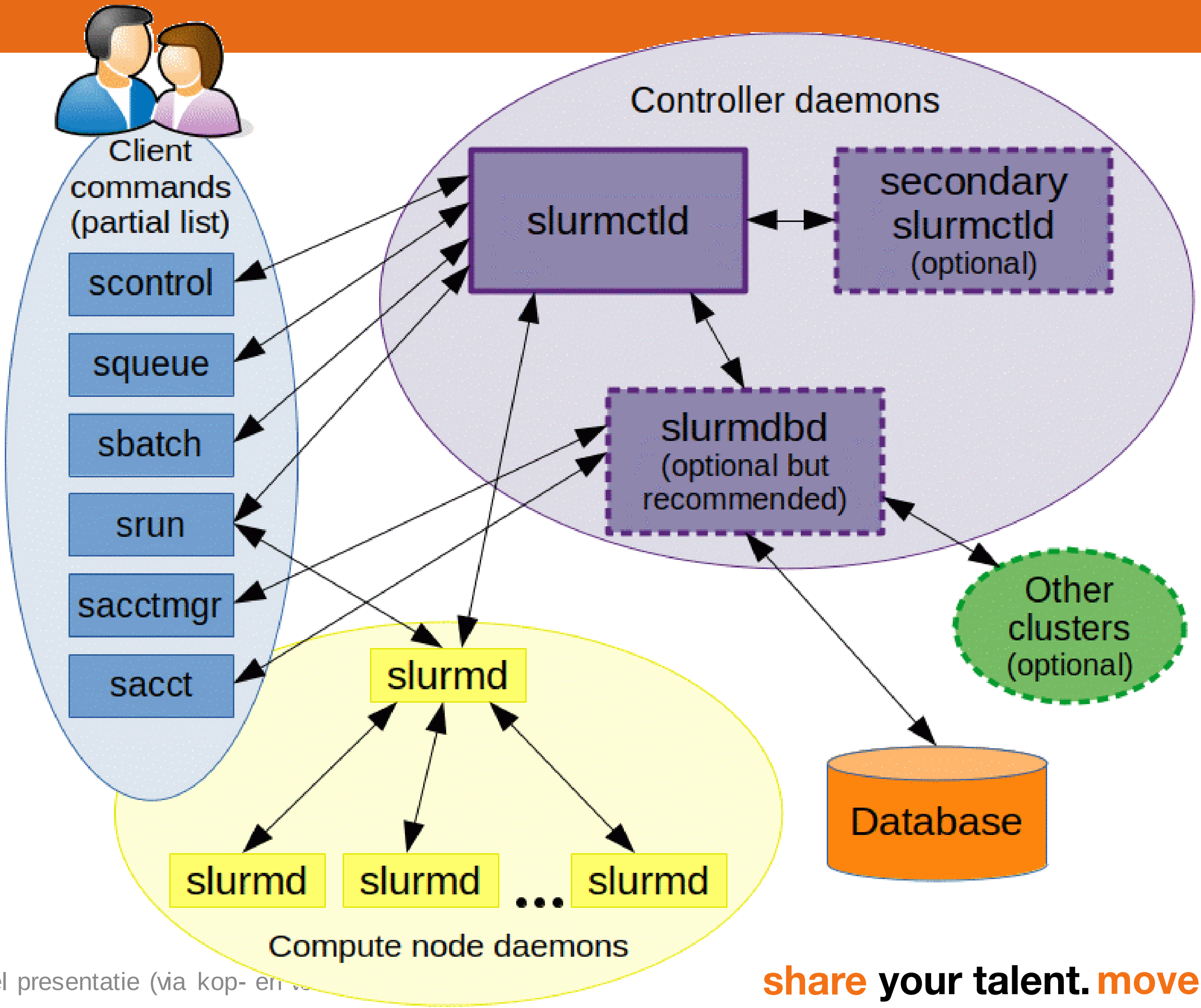
SLURM

share your talent. move the world.

SLURM cluster basics

Sharing is caring

- Since most programs for users run a solid block of time, and then sit around waiting for input, it is efficient to *schedule* them on the cluster
- The main job of the cluster software, called SLURM (https://slurm.schedmd.com), is to make sure the cluster is working at full capacity the whole time
- This means that if your job ”fits” in with the current cluster load, it can run on a node (one whole computer), several nodes, or parts of these (the individual ”cores”)
- In addition, for fairness, if you’ve just used a lot of resources, you’re placed back in the queue and have to wait your turn for a while


Client commands (partial list)
scontrol
squeue
sbatch
srun
sacctmgr
sacct
Controller daemons
slurmctld
secondary slurmctld (optional)
slurmdbd (optional but recommended)
Other clusters (optional)
Database
slurmd
slurmd
slurmd
...
slurmd
Compute node daemons

Peregrine basics

Sharing is caring

- In order to run programs, you need to submit them as a Bash ”script”. It sounds complicated, but it doesn’t have to be. You can treat it as a one-at-a-time way to run programs, just like pressing ”enter” on the command line. You just need to save that command in a file.
- In order for the SLURM scheduler to do its job properly, you also need to tell it about the resources you expect to use and the maximum time you expect it to take.
- Finally, you need to be aware that you can’t interact with your program directly, but only through *job control* commands provided by SLURM
- And you need to catch output explicitly! (Hence, focus on redirection)

Defining jobs

The syntax of the submit script

- To submit a job, you need to open a file, write some information for the scheduler in it and the command you want to run
- NOTE: you can actually submit jobs using only the command line too, but it’s not good (bioinformatics) practice
- You need to start the script with the line ”#!/bin/bash” (all scripts, really)
- Information for the scheduler is ”prepended” with #SBATCH
- The most basic information is how long you estimate your job will take, so-called ”wall time”
- For instance: #SBATCH --time 2:00:00
- This will say your jobs takes (maximum) two hours: *it is deleted after!*

share your talent. move the world.

Defining jobs

The syntax of the submit script

- After you’ve specified the maximum time, your program is ready to run; it will then by default run on one computer (node) using one task (CPU)
- Normally, however, it’s wise to also indicate how many extra resources you expect to use
- This is a bit of a black art: you need to know something about your programs. For now, you can assume almost all bioinformatics can use multiple *cores* (CPU’s) but not multiple *nodes* unless very specifically noted
- You specify the number of CPU’s to use with ” --cpus-per-task”

Defining jobs

The syntax of the submit script

- So let’s say you want to define a job that should take at most two hours, on one node, with 16 cores in use:

```
#!/bin/bash
#SBATCH --time 2:00:00
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
```

Defining jobs

The syntax of the submit script

- So let’s say you want to define a job that should take at most two hours, on one node, with 16 cores in use:

```
#!/bin/bash
#SBATCH --time 2:00:00
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
```

- We still have to also provide the maximum amount of memory we expect the process to use, with ”#SBATCH --mem=*size_in_MB”*
- You can either just specify some part of the machine (remember, most of them have 128GB), so say you reserve half if you use half the CPUs
- But it is better to experiment a bit and make an estimate.

Defining jobs

...and now for the most important part...

- So you've estimated the resources and time your job will need and sent them to SLURM using the ”#SBATCH” directives...
- ... how do you now actually run a job?
- Easy: just write a command in the file *just like you would on the command line*
  - In fact, you can try it out on the command line first, and then just copy-paste it into the file
- Be aware that any modules you need you *either* have active when you start the job (they're ”copied” to wherever the job runs) *-or-* you need to also use some ”module load *mod”* commands in your script!

Defining jobs

...and now for the most important part...

- So let’s say we have the following:

```
#!/bin/bash
#SBATCH --time 2:00:00
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
module load Bowtie2
export BOWTIE2_INDEXES=/data/p225083/BOWTIE2_INDEXES
export DATA=/data/p225083
bowtie2 -x human -U $DATA/all.fq -p 16 -S ${DATA}/output.sam
```

- What does this do?

Start a job

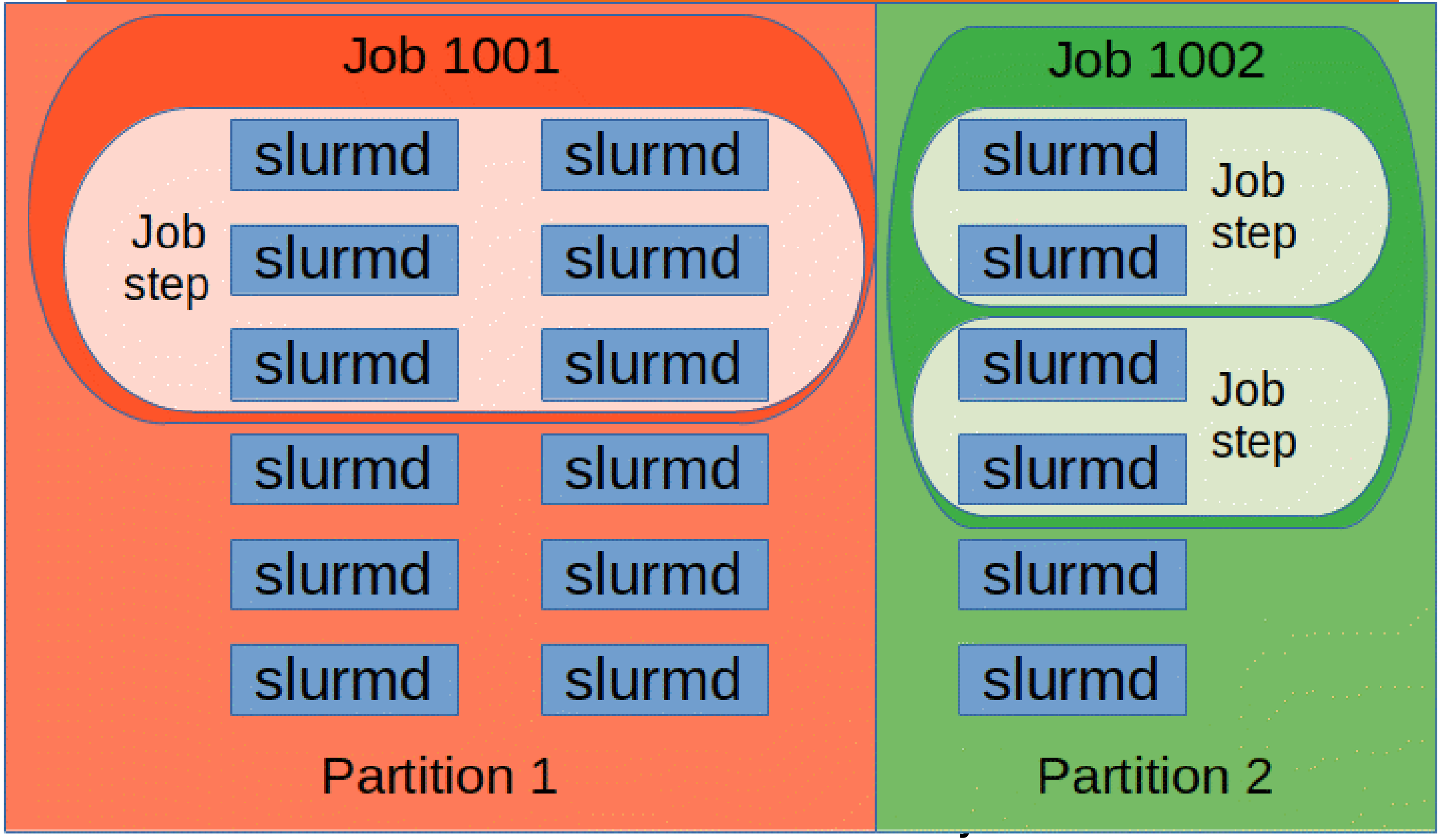
I’ve made my file already, how do I get going?

- To tell the cluster your want to run a job, submit it: ”sbatch *scriptfile*”
- You’ll be told your script has been submitted (unless there was an error)
- So how do you know what’s going on?
- 2 methods:
  - Mail: by adding the ”#SBATCH --mail-type=ALL” as well as specifying ”#SBATCH --mail-user=*you@yours.com*”to get mail about what is going on
  - Specify where the output is going with ”#SBATCH --output=somefile” and then see the output go by with ”tail” (or ”watch”, see manpage!)

Using SLURM’s information facilities

Sometimes you just want to know *more*

- The running job will mail you and copy output, but SLURM/Peregrine also has other commands to tell you more.
- You needed to remember the jobid, but what if you forgot it?
- Run the “squeue -u *p-number*” command to see all your jobs (if running or waiting)
- You can also use the ”sacct” command to see what all your jobs are doing, if they are running (and with ”sacct -a” all the jobs on the cluster!)
- You can run the ”jobinfo *jobnumber*” command to see exactly what is going on.
- You can stop a job with ”scancel *jobnumber”*


Job 1001
Job step
slurmd
slurmd
slurmd
slurmd
slurmd
slurmd
slurmd
slurmd
slurmd
slurmd
slurmd
slurmd
Partition 1
Job 1002
Job step
slurmd
slurmd
Job step
slurmd
slurmd
slurmd
slurmd
Partition 2

One last thing…

The cluster is divided in *partitions*

- The cluster consists of different types of nodes, some of them very large, while some of them are reserved for short jobs
- With the ”#SBATCH --partition=*partition_type*” you can specify that your job runs only on that type of node
- For instance, many bioinformatics tools (eg. mapping, blasting) run short periods of time (compared to the Molecular Dynamics people ☺)
- So it pays off to use the ”#SBATCH --partition=short” directive to say you’re probably only going to use a resource for 30 minutes at most
- On the other hand, many assembly jobs take massive amounts of memory: use “#SBATCH --partition=himem” to request a big node.

What *is* Peregrine, exactly?

You're almost a programmer now…

- Peregrine is a *cluster* of Linux servers, 172 of them
- Each of them (or, most) is the same configuration as the login node that you've used so far: 24 cores, 128GB memory
- One of these is already enough for a lot of bioinformatics work, but there are a number of larger cores too
- The real value is in using multiple machines at once *if your program can handle that*

What *is* Peregrine, exactly?

You're almost a programmer now…

- If you are working at the UMCG or RuG you can use it by registering; see the website https://www.rug.nl/society-business/centre-for-information-technology/research/services/hpc/facilities/peregrine-hpc-cluster?lang=en
- Also see their docs for many examples; https://wiki.hpc.rug.nl/peregrine/start

share your talent. move the world.

Bash: Redirecting output

Redirecting output

Where do you want me spit this out?

- You’ve seen output from commands fly by so far. You could copy it from Mobaxterm and paste it into a file, but Linux gives you the option of *redirecting* the output into a file, rather than letting it scroll by.
- *The operator to do that is the ”>” (greater than) character.*
- For instance, try ”cat NC_008120.fna > outputfile.txt”
- You’ll now see a new file with ”ls”, and you won’t see the output from the cat command.
- If you look at the new file with less, you’ll see it contains the same information as the original -- essentially a fancy way to do ”cp”!

Redirecting output

Where do you want me spit this out?

- This operator, like the ”mv” command comes with a sting though: try “cat NC_008121.fna > outputfile.txt”
- It will now replace the contents of the file outputfile.txt *without asking you*.
- You can see this more clearly if you try ”cat NC_*.fna > outputfile.txt”
  - Only the contents of the last (alphabetically) file will be saved!
- So be sure to check that the file where you’re redirecting doesn’t exist!
  - We’ll see an automated way to do this on day 3.

Redirecting output

Where do you want me spit this out?

- Alternatively, there is also an ”additive” redirector: ”>>”. This means that you *add* the output of the command to the file you specify. If the file is empty, it works the same as a single ”>”.
- Try ”cat NC_008120.fna >> outputfile.txt” and check with less what is in there. Also do an ”ls -hl” on the file to check its size.
- Now add the next file to it with ”cat NC_008121.fna >> outputfile.txt”
    - If you check with ”less now”, you’ll see both .fna files’ contents are in the outputfile.txt Also, an ls -hl will tell you its size has increased.
    - How would you add all the Yersinia *.fna files to a multifasta file ”yersinia.fna” ?

Redirecting output

Where do you want me spit this out?

- We’ll see on day 4 that not *all* output is in fact the same; some of it is ”standard” (what you’ve seen so far”) and some of it is ”errors”. You can separate these two types of output using separate ”>” signs.
  - However, since we don’t have an immediate use for this now, we’ll skip this until day 4 when it will come in handy during compilation.

Redirecting output

From program to program

- Finally, we will see how to connect programs together in pipelines.
- The operator to do this is the ”|” symbol that so far must have seemed to be a pretty useless key on your keyboard to you.
- *This ”pipe” symbol allows you to build pipelines*
- You ”pipe” output from one program to the next, each of them doing something to the input and spitting out altered output to the next program.
- The results are not saved in temporary files: ”streaming processing”
- *This* is what makes Linux so great for text processing.

Bash:
The *for* loop

Doing this repeatedly with *for*

You’re almost a programmer now…

- The terminal allows you to automate repetitive tasks by writing *loops* using a *for* statement
- For instance, you want to rename all the fastq files in a directory so that they include the date the experiment was carried out. But you don’t want to type *almost* the same command so many times…
- Or, you want to start a number of assembly jobs equal to the number of nodes in your network…
- Or, you want to split up a huge file into smaller one…
- …the list is endless!

Doing this repeatedly with *for*

You’re almost a programmer now…

- A for loop has the following format:

```
for x in … ; do something ; done
```

- NB: when writing scripts, you can write this down more clearly, but for writing it down on the command line you need to separate with ”;”
- The … is a so-called sequence, a simple on would be:

```
for x in {1..10} ; do something ; done
```

- The ”x” is a variable, which is set consecutively from 1 to 10
- Finally, the ”something” is anything you can do on the command line!

About variables

You’re almost a programmer now…

- A trivial example

```
for x in {1..10} ; do echo $x ; done
```

- Try it now…
- ”echo” is the command for printing something
- If you type ”echo hello” on the command line, it’ll write ”hello” to the output
- You want to print the constantly changing values in the for-loop
- You need to refer to ”x” as a variable… prepend it with a $ sign
- (see what happens if you don’t!)

Variables

You’re almost a programmer now…

- Variables are everywhere on the command line; to see a list of them, type ”export”
- You can echo the value of any of them; some are useful to know; e.g. the $PATH variable; ”echo $PATH”
- If you make a new variable, you only need to assign it a value; for instance x=10
- However, once you’ve pressed enter, its value is lost
- To keep the variable around after 1 line of code, you also use ”export”; type ”export x=10” and when you type export again it will show up

Variables

You’re almost a programmer now…

- It is important to realize that what’s in a variables is copied when you ”assign them”;
- Type: ”export x=10 ; export y=20 export z=$x ; echo $x $y $z”
- Then type: ”x=100 ; echo $x $y $z”
- Can you explain what just happened?
- Why is it not ”$x=100”? (Try to see what happens if you do that).
- There is much more to learn (esp. variable editing), some of which we’ll cover as we need it later in the course.

Doing this repeatedly with *for*

You’re almost a programmer now…

- We won’t be using variables much more like that, but we will be using them in our loops.
- If we want to use the variable as part of another word (”string”), we need to do something extra: ${x}
- For instance, this loop will make 10 files (try it out!):

```
for x in {1..10} ; do touch file${x} ; done
```

- It often comes in handy in (re-)naming files.

Doing this repeatedly with *for*

You’re almost a programmer now…

- You can also specify letter ranges, or words in the range that your variable is set to.
- For instance, this loop will make 3 files (try it out!):

```
for x in {file1, file2, file3} ; do touch $x ;
done
```

- Or to make a large number of fasta files:

```
for x in {a..z} ; do touch ${x}.fa ; done
```

Doing this repeatedly with *for*

You’re almost a programmer now…

- You can also modify the range to loop in steps of 2 instead:

```
for x in {1..10..2} ; do touch file${x} ; done
```

- You can also easily combine ranges, see what the following does. (Do this in a new directory in your home)

```
for x in {a..z}{1..10..2} ; do touch file${x} ;
done
```

Doing this repeatedly with *for*

You're almost a programmer now…

- So we see how to set a variable... but what else to set it *to?*
- In our current examples, we create a *range* of numbers.
- Often, we want to match filenames: for this we use *globbing*:

```
for x in *.py ; do chmod a+x $x ; done
```

- This would make all your Python scripts executable by anyone.

```
for x in *.fq ; do fastqc $x ; done
```

- Would start the fastqc program on all fastqfiles (not installed yet).

Writing multiple statements in a *for* loop

You’re almost a programmer now…

- So far we’ve ”done” only one thing in the for loop.
- The shell allows you to do multiple things in the loop, but when writing them out on the command line (as opposed to a script), this can be awkward:

```
for x in *.faa ; do grep -v > ${x} ; wc -c ${x}
; done
```

- Simply separate all statements with semicolons ”;” and end with done.
- I recommend you ”construct” such a line in another editor (mobaxterm) and only paste the final results into the terminal.
- But really, this is where you would want to start scripting in earnest!

Looping over the output of a command

You’re almost a programmer now…

- Using the ”backtick” notation `command` we can substitute the output from a command *in place*
- The for loop will now loop over every item output by the command:

```
for x in `cut -f 3 *.gff`; do echo ${x} ; done
```

- You could put whole pipelines in there!
- Usually I recommend that you do your processing inside the body of the loop though, it’s more versatile.
- But really, this is where you would want to start scripting in earnest!

Recap

You’re almost a programmer now…

- The *for* construct allows you to automate a repetitive task by performing a command (or a number of them separated by ”;”) on a variable which can be specified by a *range* or by file *globbing*.
- Specifying ranges is very flexible; it can be a simple list, you can count, you can list numbers, and determine the increment.
- You can also loop over the output of a command by using backticks.
- If you construct really complicated *for* loops, you’re better off editing them in an editor (mobaxterm, notepad) or writing a proper script.
- …but that’s for another course!

Redirecting output

From program to program

- Here’s a trivial example: type ”ls -l /home | wc -l”
  - *What do you think this does?*

Redirecting output

From program to program

- So that’s a quick way to count the number of users who have accounts on the system.
- *How about counting the number of genes in a GFF file?*
- *Cut -f 3 *.gff | grep gene | wc -l*
- *How about finding only your favorite PFAM domains?*
- *Cut -f 3 -d ’;’ *.gff | grep favorite_name*
- *How many proteins are there in a protein fasta file?*
- *Grep ’>’ *.faa | wc -l*

Redirecting output

From program to program

- As you can see, pipelines allow you to construct arbitrary ”filters” for these files.
    - Construct a filter that tells you how many amino acids are in all *.faa files for Yersinia.
    - Construct a filter that tells you what the largest file in a directory is.
    - Construct a filter that counts the number of *DnaX* genes in a GFF file.

Redirecting output

From program to program

- Another use of pipes is to avoid using temporary files. Here's the canonical example for an RNAseq mapping and analysis:
  - bwa sampe ./hg19.fasta ./s1_1.sai ./s1_2.sai ./s1_1.fastq ./s1_2.fastq | \ samtools view -Shu - | \ samtools sort - - | \ samtools rmdup -s - - | \ tee s1_sorted_nodup.bam | \ bamToBed > s1_sorted_nodup.bed
  - It looks horribly complicated, but by the end of the course, you'll be writing these too ;-)
  - Right now, we don't have the programs we need to run that command, yet.

Exercises

Practice what you (or I) preach

- How would you use the cut and grep commands to find out if there are duplicate genes in a genome?
- How would you combine cut, grep and wc to count the number of tRNA's annotated in a *.gff file?
- How would you sort the PHRED quality scores *at the end of a quality string* in a fastq file?
- How could you use protein to find a gene your looking for in all the *.faa files in a directory? In two directories?

Doing stuff to variables

...unlike labrats, you don’t need a license to mangle them.

- We’ve been using variables extensively in for loops, to refer to whichever file or part of a sequence the for loop is currently processing.
  - E.g.

```
for c in {1..10} ; do touch ${c}.txt ; done
```

- What does this do?
- Whatever is in the variable $c is added to the ”.txt” extension, and together touch is told to make this new file.
- Oftentimes, we want to do *more* with whatever is in the variables.

Doing stuff to variables

Removing the extension

- In the previous example, we made 10 files, and let’s say we now want to rename them to ”*.png” files instead?

```
for f in *.txt ; do mv $f ${f}.png ; done
```

- This does not do what we want! We want to remove the ”*.txt” first!
- Variable editing to the rescue!

```
for f in *.txt ; do mv $f ${f%txt}png ; done
```

- This is construction takes the curly-brace {} syntax for referring to variables that we’ve used before, and adds *operators* inside it that match patterns.
    - NB: in this case, *not* regex!

Doing stuff to variables

Regular expressions

- Here’s the general ”formula” to edit these kinds of (string) variables:

`${var%pattern}`	Looks for pattern on the end, removes shortest match
`${var%%pattern}`	Looks for pattern on the end, removes longest match

- You can use these constructions to remove, typically, file extensions.
- Usually matching the ”shortest” is safest!

Doing stuff to variables

A sed-like construction

- Here’s a puzzle I ran in to while preparing exercises; I had some bowtie2 index files with annoying long names:

```
GCA_000001405.15_GRCh38_no_alt_analysis_set.fna.bowtie_index.1.bt2
GCA_000001405.15_GRCh38_no_alt_analysis_set.fna.bowtie_index.2.bt2
GCA_000001405.15_GRCh38_no_alt_analysis_set.fna.bowtie_index.3.bt2
GCA_000001405.15_GRCh38_no_alt_analysis_set.fna.bowtie_index.4.bt2
GCA_000001405.15_GRCh38_no_alt_analysis_set.fna.bowtie_index.rev.1.bt2
GCA_000001405.15_GRCh38_no_alt_analysis_set.fna.bowtie_index.rev.2.bt2
```

- Annoying because when running bowtie2, you need to specify the whole name, up to the ”.1.bt2” part! I wanted all that to just be ”human”.

Doing stuff to variables

A sed-like construction

- Enter the pattern-matching-and-replace variable editing:

```
${f/pattern/replacement}
${f//pattern/replacement}
```

Replace all

- And incorporating that into my for loop, I could ”fix” the filenames easily:

```
for f in *.bt2 ; do mv $f ${f/*_index/human} ; done
ls
human.1.bt2  human.2.bt2  human.3.bt2  human.4.bt2  human.rev.1.bt2  huma
n.rev.2.bt2
```

share your talent. move the world.

Doing stuff to variables

Removing paths

- Here's how you match the beginning of a string variable:

```
${var#pattern}
${var##pattern}
```

- This is useful when removing the directory part of a long filename:

```
${filename_with_dirs##*/}
```

- Why do you use ## and not # in this case?

Copying out parts of the variable

If you know what you're looking for...just come and get it.

```
${var:start:length}
```

- Decide where you want to start "cutting".
- And decide how long you want to cut for (or until you go off the end)
  - NB: offset is 0-based!
  - If you leave out length, it assumes "until the end", too.

Taking the output of a command and save it

The equivalent of backticks, just for variables

- Many times, you want to use the output of a command as part of a string. We've seen on the command line we can use "backticks":

```
echo `date +%D_%R`
```

- But how can we get this kind of output into a variable?

```
$(command)
```

- We can now use the output of a command in-place or save it for later:

```
$(date +%D_%R)
var=$(date +%D_%R)
echo $var
```

More on redirection

Choosing where your input comes from, or output goes

- We've seen how using the ">" and ">>" operators either redirects output from a command into a file, or possible *appends* to it.

```
grep ”>.*\sY\w{3}\s*” protein.faa > ygenes.txt
```

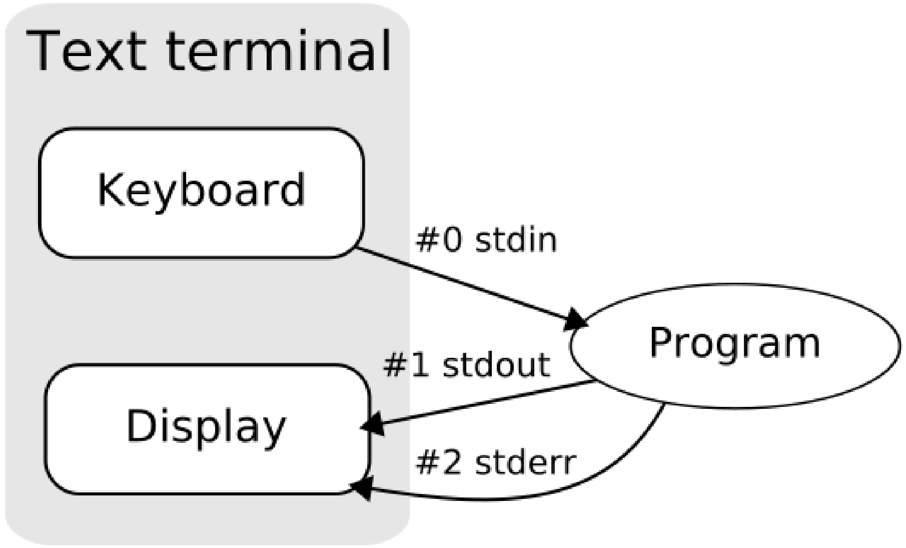
- And we've seen how to *pipe* input into a command from another:

```
cat dnafile.fna | sed ”s/T/U/g” > rnafile.fna
```

More on redirection

What’s really going on

- Actually, all command-line programs are associated with 3 ”stream”:
  - input
  - output
  - standard error



share your talent. move the world.

More on redirection

Catching those errors

- To catch *only* the errors from a command, and leave the normal output to stream to the console, use the ”2>” operator:

```
bowtie2 -x ecoli -U reads.fq -S output.sam 2> errors.txt
```

- This can be modified to also put the standard output in a separate file:

```
bowtie2 -x ecoli -U reads.fq -S output.sam 2> errors.txt > log.txt
```

- But what if you wanted to do the opposite?

More on redirection

Combining streams

- You can combine these streams to the same place by using ”2>&1”:

```
bowtie2 -x ecoli -U reads.fq -S output.sam 2>&1
```

- So why does the above not seem to do anything?

```
bowtie2 -x ecoli -U reads.fq -S output.sam > output.txt 2>&1
```

- You need to redirect the standard output stream to a file first, before being able to see any difference! (Normally, errors go to the console!)

More on redirection

Here ya go!

- The following is a slightly advanced example for scripts, but we'll meet some of them in a minute, so that's fair game.
- If you redirect the input with "<", you basically read from a file, not having to use grep:

```
grep "TATA" < nucleotide.fna
```

- This is not that useful for grep, since it allows you to specify an input file, but some programs don't. In addition, this allows you to construct filenames from variables.

More on redirection

Here ya go!

- An advanced topic are ”here documents”, using the double ”<< *label*”.

- What this does is pretend that the text coming after *label* is treated as if it were you, typing at the command line, until you type the actual *label*.
- It allows you in a script to construct a ”textfile” on the fly, inside your script, using all the variable tricks and commands we’ve learned so far.

- You probably won’t be doing this anytime soon, but you could for instance generate a ”fastafile” inside your script, using information you’ve gathered, and then use that as input to a command.

More on redirection

The good old bitbucket...

- …or what to do if a program doesn't have a nice and quiet (-q) option.

```
any_program_without_quiet_option > /dev/null 2>&1
```

- ...and it will never bother you again!
- /dev/null is a special "file" (device) on all Linux systems, and it's just a black hole for data; anything redirected to it dissappears, anything read from it is just zeroes.
- In practice, be careful throwing away standard error; you might need those messages later when figuring out how your analysis went wrong!

Redirection is crucial in scripts for SLURM

In the cluster, no one can hear you scream

- As we’ll see in the next section, programs run on the SLURM cluster are not actually connected to any terminal; in fact, you won’t even know what computer they are on. (Or rather, control).
- Anything your analysis pipeline produces is going to to be copied into one giant file, unless you take steps using redirection to separate output (standard and error) into files that are useful to you. (And throw away useless output by redirection to /dev/null)
- Using pipes in your scripts is often a good idea, but some pipelines (especially on several distinct computers) can move data in this way. Then you can pass data around using redirection, variable editing.

Exercises

Redirect your way out of this bro!

- If you wanted to make a logfile from the standard error, so appending all the errors from your programs, what operator would you use?
- Say you wanted to consolidate naming of fastq files; sometimes they're called "*.fastq" and sometimes "*.fq". How would you write a for loop to rename them to one or the other?
- Say you have paired-end reads in fastq files, and the second part of the filename (in between underscores _) is the barcode. You want to combine them into one fastqfile (simply appending). How would you use variable editing to extract the barcode and use that as the filename, appending a ".fq" extension? (So: "barcode.fq" containing all the reads)

Recap

Variables and redirection

- Using the special ${x%}, ${x#} and ${x/} you can slice and extract pieces of your variable's contents to re-use in your scripts.
- If you want to capture output from a command and do something with it, you can use it in-place in a variable with "$( )"
- All programs have a standard input, standard output and standard error. These so-called "streams" can redirected separately or together, depending on which one is useful.