# Comparative and Functional Genomics

## Practical session 3 - Extra

## Bash commands

2018

ESCI upf.
School of International Studies

# Permissions and Ownership

## List directories and files

```
$ ls -al # shows something like this for each file/dir: drwxrwxrwx
         # d: directory
         # rwx: read write execute
         # first triplet: user permissions (u)
         # second triplet: group permissions (g)
         # third triplet: world permissions (o)
```

# Permissions and Ownership

## Assign write and execute permissions to user and group

```
chmod ug+rx my_file
```

## To remove all permissions from all three user groups

```
chmod ugo-rwx my_file
            # '+' causes the permissions selected to be added
            # '-' causes them to be removed
            # '=' causes them to be the only permissions that the file has.

chmod +rx public_html/ or $ chmod 755 public_html/ # Example for number system:
```

## Change ownership

```
chown <user> <file or dir>              # changes user ownership
chgrp <group> <file or dir>             # changes group ownership
chown <user>:<group> <file or dir>      # changes user & group ownership
```

# Permissions and Ownership

You are the owner of a file (myfile), and you want to set its permissions:

- the **u**ser can **r**ead, **w**rite, and e**x**ecute it
- members of your **g**roup can **r**ead and e**x**ecute it
- **o**thers may only **r**ead it.

# Permissions and Ownership

You are the owner of a file (myfile), and you want to set its permissions:

- the **u**ser can **r**ead, **w**rite, and e**x**ecute it
- members of your **g**roup can **r**ead and e**x**ecute it
- **o**thers may only **r**ead it.

This command will do the trick:

```
chmod u=rwx,g=rx,o=r myfile
```

# Permissions and Ownership

You are the owner of a file (myfile), and you want to set its permissions:

- the **u**ser can **r**ead, **w**rite, and e**x**ecute it
- members of your **g**roup can **r**ead and e**x**ecute it
- **o**thers may only **r**ead it.

This command will do the trick:

```
chmod u=rwx,g=rx,o=r myfile
```

Here is the equivalent command using octal permissions notation:

```
chmod 754 myfile
```

# Permissions and Ownership

You are the owner of a file (myfile), and you want to set its permissions:

- the **u**ser can **r**ead, **w**rite, and e**x**ecute it
- members of your **g**roup can **r**ead and e**x**ecute it
- **o**thers may only **r**ead it.

This command will do the trick:

```
chmod u=rwx,g=rx,o=r myfile
```

Here is the equivalent command using octal permissions notation:

```
chmod 754 myfile
```

Here the digits **7**, **5**, and **4** each individually represent the permissions for the user, group, and others, in that order. Each digit is a combination of the numbers **4**, **2**, **1**, and **0**:

- **4** stands for read
- **2** stands for write
- **1** stands for execute
- **0** stands for no permission

# Text Viewing

```
more <my_file>          # views text, use space bar to browse, hit 'q' to exit

less <my_file>          # a more versatile text viewer than 'more', 'q' exits, 'G' moves to end of text,
                        # 'g' to beginning, '/' find forward, '?' find backwards

cat  <my_file>          # concatenates files and prints content to standard output
```

# Text Editors

- Vi and Vim:Non-graphical (terminal-based) editor. Vi is guaranteed to be available on any system. Vim is the improved version of vi.

# Text Editors

- Vi and Vim:Non-graphical (terminal-based) editor. Vi is guaranteed to be available on any system. Vim is the improved version of vi.
- Emacs: Non-graphical or window-based editor. You still need to know keystroke commands to use it. Installed on all Linux distributions.

# Text Editors

- Vi and Vim:Non-graphical (terminal-based) editor. Vi is guaranteed to be available on any system. Vim is the improved version of vi.
- Emacs: Non-graphical or window-based editor. You still need to know keystroke commands to use it. Installed on all Linux distributions.
- XEmacs: More sophisticated version of emacs, but usually not installed by default. All common commands are available from menus. Very powerful editor.

# Text Editors

- Vi and Vim:Non-graphical (terminal-based) editor. Vi is guaranteed to be available on any system. Vim is the improved version of vi.
- Emacs: Non-graphical or window-based editor. You still need to know keystroke commands to use it. Installed on all Linux distributions.
- XEmacs: More sophisticated version of emacs, but usually not installed by default. All common commands are available from menus. Very powerful editor.
- Pico: Simple terminal-based editor available on most versions of Unix. Uses keystroke commands, but they are listed in logical fashion at bottom of screen.

# Text Editors

- Vi and Vim:Non-graphical (terminal-based) editor. Vi is guaranteed to be available on any system. Vim is the improved version of vi.
- Emacs: Non-graphical or window-based editor. You still need to know keystroke commands to use it. Installed on all Linux distributions.
- XEmacs: More sophisticated version of emacs, but usually not installed by default. All common commands are available from menus. Very powerful editor.
- Pico: Simple terminal-based editor available on most versions of Unix. Uses keystroke commands, but they are listed in logical fashion at bottom of screen.
- Nano: A simple terminal-based editor which is default on modern Debian systems.

# Text Editor

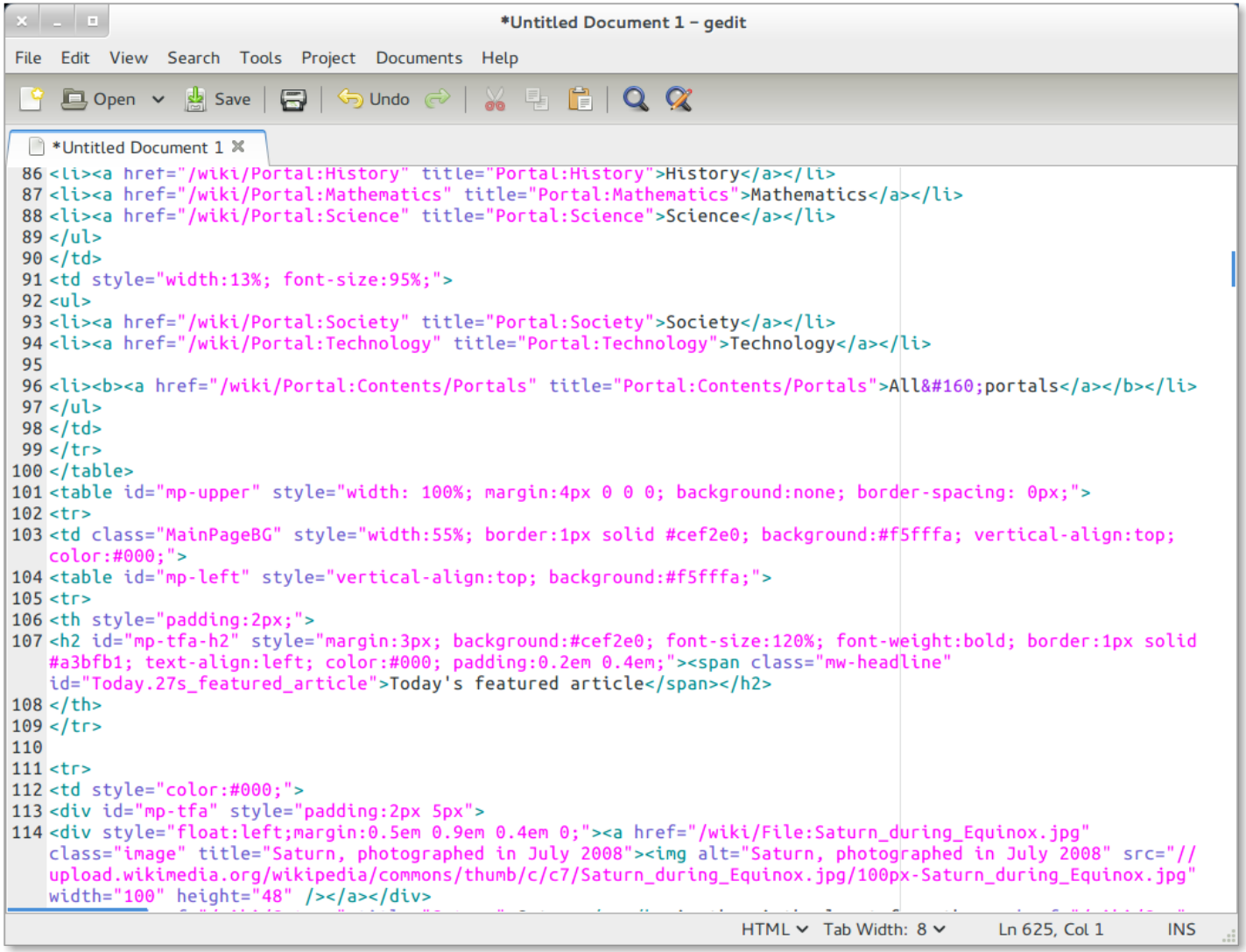
## Gedit

Is a GUI-based text editor for the GNOME desktop environment, Mac OS X and Microsoft Windows. Designed as a general purpose text editor, gedit emphasizes simplicity and ease of use. It includes tools for editing source code and structured text such as markup languages.

Includes

- syntax highlighting for various program code and text markup formats.
- GUI tabs for editing multiple files
- full undo and redo system as well as search and replace
- line numbering, bracket matching, text wrapping, current line highlighting, automatic indentation and automatic file backup

# Text Editor




ESCI upf. School of International Studies

# Unix commands applied to bioinformatics

The following examples use primarily two files: arrayDat.txt and arrayAnnot.txt.

## Download data

You can download these files from ▸ Here

**arrayDat.txt** contains some microarray data, with rows corresponding to probe IDs and columns corresponding to four samples.
**arrayAnnot.txt** contains the annotation for the probe IDs, specifically the gene description (i.e. geneTitle) and the gene symbol (i.e. geneSymbol).

# Unix commands applied to bioinformatics

```
Create a directory named arrays

Put arrayDat.txt and arrayAnnot.txt files in it

cd arrays (full path)

ls -l
```

# Text viewing - Practice

```
# show file content (more)

# show file content (less)

# Assign write and execute permissions to ugo of arrayDat.txt

# list files and see permissions

# Try to edit arrayDat using a text editor (gedit).

# Assign read, write and execute permissions to all users of arrayDat.txt. List files to see permissions


# Edit arrayDat using a text editor (gedit)
```

# Text viewing - Practice

```
# show file content (more)
more arrayDat.txt

# show file content (less)
less arrayDat.txt

# Assign write and execute permissions to ugo of arrayDat.txt
chmod 111 arrayDat.txt

# list files and see permissions
ls -l

# Try to edit arrayDat using a text editor (gedit).
gedit arrayDat.txt

# Assign read, write and execute permissions to all users of arrayDat.txt. List files to see permissions
chmod 777 arrayDat.txt
ls -l

# Edit arrayDat using a text editor (gedit)
gedit arrayDat.txt
```

# STDIN, STDOUT, STDERR and Redirections

By default, UNIX commands read from standard input (STDIN) and send their output to standard out (STDOUT).
You can redirect them by using the following commands:

```
ls > file                        # prints ls output into specified file

command < my_file                # uses file after '<' as STDIN

command >> my_file               # appends output of one command to file

command | tee my_file            # writes STDOUT to file and prints it to screen

command > my_file; cat my_file   # writes STDOUT to file and prints it to screen

command > /dev/null              # turns off progress info of applications by redirecting
                                 # their output to /dev/null

grep my_pattern my_file | wc                       # Pipes (|) output of 'grep' into 'wc'
grep my_pattern my_non_existing_file 2 > my_stderr # prints STDERR to file
```

# Multiple commands

Run multiple commands. it does not matter if a command returns to error, all the commands you use will be executed eventually.

```
command1; command2; command3
```

Run multiple commands in the condition that the previous command must complete successfully first before running the next command.

```
command1 && command2 && command3
```

Run multiple commands in the condition that the next command will be executed if the previous one fails.

```
command1 || command2 || command3
```

# Unix commands applied to bioinformatics

- head
- cut
- paste
- sort
- wc
- uniq
- grep
- split
- sed
- awk

# head

- You can use this command to view the content of the top part of a file.
- The switch "-n" allows you to specify how many lines to display, starting from the first. If you specify a negative number N, then all the lines except the bottom N will be displayed.

```
# print first 5 lines


# print everything except the last line
```

# head

- You can use this command to view the content of the top part of a file.
- The switch "-n" allows you to specify how many lines to display, starting from the first. If you specify a negative number N, then all the lines except the bottom N will be displayed.

```
# print first 5 lines
head -n 5 arrayDat.txt
head -n 5 arrayAnnot.txt

# print everything except the last line
head -n -1 arrayDat.txt
head -n -1 arrayAnnot.txt
```

# cut

- This command extracts portions of lines of a file.
- The switch "-f" allows you to specify which field (by default column) to extract.
- Multiple fields can be selected either by specifying a range (e.g. 1-3) or by listing the fields (e.g. 1,2,3).

```
# get 1st column (i.e. probeID of arrayDat.txt)

# get 3rd column (i.e. geneSymbol of arrayAnnot.txt)

# get the first three columns (arrayDat.txt)

# get the 1st and 3rd columns (arrayAnnot.txt) and redirect the  output into a file named probe2gene.txt
```

# cut

- This command extracts portions of lines of a file.
- The switch "-f" allows you to specify which field (by default column) to extract.
- Multiple fields can be selected either by specifying a range (e.g. 1-3) or by listing the fields (e.g. 1,2,3).

```
# get 1st column (i.e. probeID of arrayDat.txt)
cut -f 1 arrayDat.txt

# get 3rd column (i.e. geneSymbol of arrayAnnot.txt)
cut -f 3 arrayAnnot.txt

# get the first three columns (arrayDat.txt)
cut -f 1-3 arrayDat.txt

# get the 1st and 3rd columns (arrayAnnot.txt) and redirect the  output into a file named probe2gene.txt
cut -f 1,3 arrayAnnot.txt > probe2gene.txt
```

# paste

- This command merges the lines of files and puts them side by side.

```
# put each column in a separate tmp files (arrayAnnot.txt)



# merge lines in a new column order (arrayAnnotOrdered.txt)
```

# paste

- This command merges the lines of files and puts them side by side.

```
# put each column in a separate tmp files (arrayAnnot.txt)
cut -f 1 arrayAnnot.txt > tmp1
cut -f 2 arrayAnnot.txt > tmp2
cut -f 3 arrayAnnot.txt > tmp3

# merge lines in a new column order (arrayAnnotOrdered.txt)
paste tmp3 tmp1 tmp2 > arrayAnnotOrdered.txt
```

# sort

- This command sorts the lines of a text file.
- By default the sort is in lexicographic order according to the first field.
  - In a lexicographic order, letters follow number: 0-9 then aA-zZ.
  - Note that numbers are to be treated as strings, so 10 comes before 2 because there is no positional weighting (the symbol 1 comes before 2).
- Other sorting criteria are available:
  - the switch -k lets you specify which field to use as key for sorting.
  - the switch -n specifies a numeric sort.
  - the switch -r specifies a sort in reverse order (either lexicographic or numeric).
  - the switch -R specifies a random sort.

# sort

```
# sort by 1st field (probeID) (probe2gene.txt)

# sort by the 2nd field (geneSymbol) (probe2gene.txt)

# sort by 2nd field (default lexicographic sort) (arrayDat.txt)

# sort by 2nd field (numeric sort) (arrayDat.txt)
```

# sort

```
# sort by 1st field (probeID) (probe2gene.txt)
sort probe2gene.txt

# sort by the 2nd field (geneSymbol) (probe2gene.txt)
sort -k 2 probe2gene.txt

# sort by 2nd field (default lexicographic sort) (arrayDat.txt)
sort -k 2 arrayDat.txt

# sort by 2nd field (numeric sort) (arrayDat.txt)
sort -n -k 2 arrayDat.txt
```

# WC

- This command counts the lines, words and bytes in a text file.
- It is often used in conjunction to other filtering operations to count the number of items that pass the filter.

```
# print number of lines, words and bytes (arrayDat.txt)


# print number of lines only


# print the number of txt files in the current directory
```

# WC

- This command counts the lines, words and bytes in a text file.
- It is often used in conjunction to other filtering operations to count the number of items that pass the filter.

```
# print number of lines, words and bytes (arrayDat.txt)
wc arrayDat.txt

# print number of lines only
wc -l arrayDat.txt

# print the number of txt files in the current directory
ls *.txt | wc -l
```

# uniq

- This command reports or filters repeated lines in a file.
- It compares adjacent lines and reports those lines that are unique. Because repeated lines in the input will not be detected if they are not adjacent, it might be necessary to sort the input file before invoking this command.
- The switch "-f" specifies how many fields (starting from the first one), to skip when performing the comparisons.
- The switch "-c" specifies to return a count of how many occurrences for each distinct line.
- The switch "-d" specifies to print only duplicates lines.
- The switch "-u" specifies to print only unique lines (i.e. occurring only once).
- The switch "-i" specifies a case insensitive comparison.

# uniq

```
# print all distinct rows then count (arrayAnnot.txt)

# count distinct rows without considering the 1st field (probeID)

# report distinct gene symbols and count number of occurrences

# report repeated gene symbols

# report list of unique gene symbols
```

# uniq

```
# print all distinct rows then count (arrayAnnot.txt)
uniq arrayAnnot.txt | wc -l

# count distinct rows without considering the 1st field (probeID)
uniq -f 1 arrayAnnot.txt |wc -l

# report distinct gene symbols and count number of occurrences
uniq -f 1 -c arrayAnnot.txt

# report repeated gene symbols
uniq -f 1 -d arrayAnnot.txt

# report list of unique gene symbols
uniq -f 1 -u arrayAnnot.txt
```

# grep

This command prints the lines of the input file that match the given pattern(s). (arrayAnnot.txt)

```
# print lines that match chr (lines with the words "chromosome" and/or  "cytochrome" are matched).
```

The switch "-w" specifies that the match has to occur with a whole word, not just part of a word. Thus, in the example input file, no line matches the pattern "chr" as whole word.

```
# print lines that match chr as a whole word
```

# grep

This command prints the lines of the input file that match the given pattern(s). (arrayAnnot.txt)

```
# print lines that match chr (lines with the words "chromosome" and/or  "cytochrome" are matched).
grep chr arrayAnnot.txt
```

The switch “-w” specifies that the match has to occur with a whole word, not just part of a word. Thus, in the example input file, no line matches the pattern “chr” as whole word.

```
# print lines that match chr as a whole word
grep -w chr arrayAnnot.txt
```

# grep

The switch -i can be used to specify a case-insensitive match (by default this command is case sensitive).
In the example below, when the switch "-i" is used for the pattern SS, lines containing the words class, repressor, associated, expressed as well as the gene symbol "PRSS33" are matched. (arrayAnnot.txt)

```
# print lines that match "SS" (case sensitive)


# print lines that match "SS"  (case insensitive, that is "SS" or ss" or "Ss" or "sS")
```

The switch "-c" returns the number of lines where the specified pattern is found.

```
# print how many lines match the pattern orf
```

# grep

The switch -i can be used to specify a case-insensitive match (by default this command is case sensitive).
In the example below, when the switch "-i" is used for the pattern SS, lines containing the words class, repressor, associated, expressed as well as the gene symbol "PRSS33" are matched. (arrayAnnot.txt)

```
# print lines that match "SS" (case sensitive)
grep SS arrayAnnot.txt

# print lines that match "SS"  (case insensitive, that is "SS" or ss" or "Ss" or "sS")
grep -i SS arrayAnnot.txt
```

The switch "-c" returns the number of lines where the specified pattern is found.

```
# print how many lines match the pattern orf
grep -c orf arrayAnnot.txt
```

# grep

The switch "-v" performs a "reverse-matching" and prints only the lines that do not match the pattern. (arrayAnnot.txt)

```
# print lines that do NOT match the pattern "orf"
```

The switch "-n' prints out the matching line with its number.

```
# preceed the matching line with the line number
```

When multiple patterns must be matched, you can use the switch "-f" and pass as argument a file containing the patterns to match (one per line).

```
# make a list with 5 gene symbols


# use list "tmp" to match lines in arrayAnnot.txt
```

# grep

The switch "-v" performs a "reverse-matching" and prints only the lines that do not match the pattern. (arrayAnnot.txt)

```
# print lines that do NOT match the pattern "orf"
grep -v orf arrayAnnot.txt
```

The switch "-n' prints out the matching line with its number.

```
# preceed the matching line with the line number
grep -n orf arrayAnnot.txt
```

When multiple patterns must be matched, you can use the switch "-f" and pass as argument a file containing the patterns to match (one per line).

```
# make a list with 5 gene symbols
cut -f 3 arrayAnnot.txt|head -n 5 > tmp

# use list "tmp" to match lines in arrayAnnot.txt
grep -f tmp arrayAnnot.txt
```

# split

- This command splits a file into a series of smaller files.
- The content of the input file is split into lexically ordered files named with the prefix “x”, unless another prefix is provided as argument to the command.
- The switch “-l” lets you specify how many lines to include in each file. (arrayDat.txt)

```
# split the content into separate files of 50 lines each
```

# split

- This command splits a file into a series of smaller files.
- The content of the input file is split into lexically ordered files named with the prefix “x”, unless another prefix is provided as argument to the command.
- The switch “-l” lets you specify how many lines to include in each file. (arrayDat.txt)

```
# split the content into separate files of 50 lines each
split -l 50 arrayDat.txt
```

# sed

- This command invoke a stream editor that modifies the input as specified by a list of commands.
- The pattern replacement syntax is: s/pattern/replacement/ (arrayAnnot.txt)

```
# substitute  the word "chromosome" with the string "chr"
```

# sed

- This command invoke a stream editor that modifies the input as specified by a list of commands.
- The pattern replacement syntax is: s/pattern/replacement/ (arrayAnnot.txt)

```
# substitute  the word "chromosome" with the string "chr"
sed s/chromosome/chr/g arrayAnnot.txt
```

# awk

- awk is more than just a command, it is a text processing language.
- the input file is treated as sequence of records. By default, each line is a record and is broken into fields.
- An awk command is a sequence of condition-action statements, where:
  - condition is typically an expression
  - action is a series of commands

```
awk 'condition {action}' inputfile
```

# awk

- Typically, awk reads the input one line at a time; when a line matches the provided condition, the associated action is executed.
- There are several built-in variables:
  - field var: $1 refers to the first field, $2 refers to the second field, etc.
  - record var: $0 refers to the entire record (by default the entire line).
  - NR represents the current count of records (by default the line number).
  - NF represents the total number of fields in a record (by default the last column).
  - FILENAME represents the name of the current input file.
  - FS represents the field separator character used to parse fields of the input record. By default, FS is the white space (both space and tab).

# awk

- Condition:
  - when the condition part of an awk statement is missing, it is assumed that the each record satisfies the condition, therefore the action is executed for each record.
  - when condition is “BEGIN”, the action is executed before all records have been read.
  - when condition is “END”, the action is executed after all records have been read.
- print is a simple awk command that prints the argument. By default the argument is $0.

# awk

```
# print every line of the input file (missing condition) (arrayDat.txt)

# print every line of the input file (default argument for print is $0)

# print 1st field only

# rearrange the fields, separated by a blank space

# rearrange the fields, separated by a tab

# print the number of fields for each record

# print the last field of each record
```

# awk

```
# print every line of the input file (missing condition) (arrayDat.txt)
awk '{print $0}' arrayDat.txt

# print every line of the input file (default argument for print is $0)
awk '{print}' arrayDat.txt

# print 1st field only
awk '{print $1}' arrayDat.txt

# rearrange the fields, separated by a blank space
awk '{print $1, $3, $2}' arrayDat.txt

# rearrange the fields, separated by a tab
awk '{print $1, "\t", $3, "\t", $2}' arrayDat.txt

# print the number of fields for each record
awk '{print NF}' arrayDat.txt

# print the last field of each record
awk '{print $NF}' arrayDat.txt
```

# awk

```
# print a string

#print the result of an arithmetic operation


# print first 5 lines (default action statement is to print)

# print first line (i.e. column headers)

# print first 5 records, excluding headers (NR ==1)

# print the total number of records
```

# awk

```
# print a string
awk 'BEGIN {print "Hello world!"}'

#print the result of an arithmetic operation
awk 'BEGIN {print 2+3}'
awk 'BEGIN {print "2+3=" 2+3}'

# print first 5 lines (default action statement is to print)
awk 'NR < 6' arrayDat.txt

# print first line (i.e. column headers)
awk 'NR == 1' arrayDat.txt

# print first 5 records, excluding headers (NR ==1)
awk 'NR > 1 && NR < 7' arrayDat.txt

# print the total number of records
awk 'END {print NR}' arrayDat.txt
```

# awk

By using loops and conditional statements, awk allows to perform quite sophisticated actions. For example, we can compute the sum or the mean of the intensity values for each probe across all samples:

- Compute the sum of intensity values for each gene: for all records except the column headers, we initialize the variable "s" to zero, then we loop through the all the values in a given records and sum cumulatively. When all the values have been considered, we print the sum.

```
# print sum of values for each gene (arrayDat.txt)
```

# awk

By using loops and conditional statements, awk allows to perform quite sophisticated actions. For example, we can compute the sum or the mean of the intensity values for each probe across all samples:

- Compute the sum of intensity values for each gene: for all records except the column headers, we initialize the variable "s" to zero, then we loop through the all the values in a given records and sum cumulatively. When all the values have been considered, we print the sum.

```
# print sum of values for each gene (arrayDat.txt)
awk 'NR > 1 {s=0; for (i=2; i<=NF; i++) s=s+$i; print s}' arrayDat.txt
```

# awk

- Compute the mean of intensity values for each gene: we compute the sum of the intensity values as before, but we divide the sum by the number of values considered before printing the result. Note that the number of values considered is the number of fields in each record minus one (the probe ID).

```
# print mean of values for each gene (arrayDat.txt)
```

# awk

- Compute the mean of intensity values for each gene: we compute the sum of the intensity values as before, but we divide the sum by the number of values considered before printing the result. Note that the number of values considered is the number of fields in each record minus one (the probe ID).

```
# print mean of values for each gene (arrayDat.txt)
awk 'NR > 1 {s=0; n=NF-1; for (i=2; i<=NF; i ++) s=s+$i; s=s/n; print s}' arrayDat.txt
```

# awk

Because the condition can be a pattern matching expression, awk can easily emulate grep.

```
# print the lines that match the string "orf" (arrayAnnot.txt)


# print the probe IDs whose annotation contain the string "orf"


# print number of lines matching "orf" (emulates grep -c)
```

# awk

Because the condition can be a pattern matching expression, awk can easily emulate grep.

```
# print the lines that match the string "orf" (arrayAnnot.txt)
awk '/orf/' arrayAnnot.txt

# print the probe IDs whose annotation contain the string "orf"
awk '/orf/ {print $1}' arrayAnnot.txt

# print number of lines matching "orf" (emulates grep -c)
awk '/orf/ {n++}; END {print n+0}' arrayAnnot.txt
```

# Finding files

```
find -name "*pattern*"            # searches for *pattern* in and below current directory
find /usr/local -name "*blast*"   # finds file names *blast* in specfied directory
find /usr/local -iname "*blast*"  # same as above, but case insensitive
```

additional useful arguments: -user <user name>, -group <group name>, -ctime <number of days ago changed>

```
find ~ -type f -mtime -2   # finds all files you have modified in the last two days
locate <pattern>           # finds files and dirs that are written into update file
which <application_name>   # location of application
whereis <application_name> # searches for executeables in set of directories
dpkg -l | grep mypattern   # find Debian packages and refine search with grep pattern
```

# Finding things in files

```
grep pattern file           # provides lines in 'file' where pattern 'appears',
                            # if pattern is shell function use single-quotes: '>'


grep -H pattern             # -H prints out file name in front of pattern
grep 'pattern' file | wc    # pipes lines with pattern into word count wc (see chapter 8)
                            # wc arguments: -c: show only bytes, -w: show only words,
                            # -l: show only lines; help on regular expressions:
                            # $ man 7 regex or man perlre


find /home/my_dir -name '*.txt' | xargs grep -c ^.*  # counts line numbers on many
                            # files and records each count along with individual file
                            # name; find and xargs are used to circumvent the Linux
                            # wildcard limit to apply this function on thousands of files.
```

# Process Management

```
top               # view top consumers of memory and CPU (press 1 to see per-CPU statistics)
who               # Shows who is logged into system
w                 # Shows which users are logged into system and what they are doing
ps                # Shows processes running by user
ps -e             # Shows all processes on system; try also '-a' and '-x' arguments
ps aux | grep <user_name> # Shows all processes of one user
ps ax --tree      # Shows the child-parent hierarchy of all processes
ps -o %t -p <pid> # Shows how long a particular process was running.
                  # (E.g. 6-04:30:50 means 6 days 4 hours ...)
Ctrl z <enter>    # Suspend (put to sleep) a process
fg                # Resume (wake up) a suspended process and brings it into foreground
bg                # Resume (wake up) a suspended process but keeps it running
                  # in the background.
Ctrl c            # Kills the process that is currently running in the foreground
kill <process-ID> # Kills a specific process
kill -9 <process-ID> # NOTICE: "kill -9" is a very violent approach.
                     # It does not give the process any time to perform cleanup procedures.
kill -l           # List all of the signals that can be sent to a proccess
kill -s SIGSTOP <process-ID> # Suspend (put to sleep) a specific process
kill -s SIGCONT <process-ID> # Resume (wake up) a specific process
renice -n <priority_value> # Changes the priority value, which range from 1-19,
                           # the higher the value the lower the priority, default is 10.
```

# Useful shell commands

```
cmp <file1> <file2>                              # tells you whether two files are identical

diff <fileA> <fileB>                             # finds differences between two files

split -l <number> <file>                         # splits lines of file into many smaller ones

csplit -f out fasta_batch "%^>%" "/^>/" "{*}" # splits fasta batch file into many files at '>'
egrep                                            # grep + extended regular expression
```

# wget

## File Download from the Web

```
wget ftp://ftp.ncbi.nih.... # file download from www; add option '-r' to download entire directories
```

# scp

## Secure Copy Between Machines

```
scp source target # Use form 'userid@machine_name' if your local and remote user ids are different.
                  # If they are the same you can use only 'machine_name'.
```

## Examples

```
scp user@remote_host:file.name . # Copies file from server to local machine (type from local
                                 # machine prompt). The '.' copies to pwd, you can specify
                                 # here any directory, use wildcards to copy many files.

scp file.name user@remote_host:~/dir/newfile.name
                                 # Copies file from local machine to server.

scp -r user@remote_host:directory/ ~/dir
                                 # Copies entire directory from server to local machine.
```

# Nice FTP

```
ncftp
ncftp> open ftp.ncbi.nih.gov
ncftp> cd /blast/executables
ncftp> get blast.linux.tar.Z (skip extension: @)
ncftp> bye
```

# Creating Archives

```
tar -cvf my_file.tar mydir/    # Builds tar archive of files or directories. For directories, execute command in parent directo
tar -czvf my_file.tgz mydir/   # Builds tar archive with compression of files or directories. For
                               # directories, execute command in parent directory. Don't use absolute path.
zip -r mydir.zip mydir/        # Command to archive a directory (here mydir) with zip.
tar -jcvf mydir.tar.bz2 mydir/ # Creates *.tar.bz2 archive
```

# Viewing Archives

```
tar -tvf my_file.tar
tar -tzvf my_file.tgz
```

# Extracting Archives

```
tar -xvf my_file.tar
tar -xzvf my_file.tgz
gunzip my_file.tar.gz # or unzip my_file.zip, uncompress my_file.Z,
                      # or bunzip2 for file.tar.bz2
find -name '*.zip' | xargs -n 1 unzip # this command usually works for unzipping
                      # many files that were compressed under Windows
tar -jxvf mydir.tar.bz2 # Extracts *.tar.bz2 archive
```

# Extracting Archives

Important options:

- f: use archive file
- p: preserve permissions
- v: list files processed
- x: exclude files listed in FILE
- z: filter the archive through gzip

# Script

## Script

A **script** is a list of commands that can be executed without user interaction. A **script language** is a simple programming language with which you can write scripts.

A **shell script** is a script written for the shell, or command line interpreter, of an operating system.(eg. Bash script)

# Bash script

Traditional hello world script Create a file named **script1.sh** and write ...

```
#!/bin/bash
echo Hello World
```

But, the script should have execute permissions for the correct owners in order to be runnable.

```
chmod u+x script1.sh
ls -l script1.sh
```

Run the script

```
./script1.sh
```

# Bash script

A script can also explicitly be executed by a given shell, but generally we only do this if we want to obtain special behavior, such as checking if the script works with another shell or printing traces for debugging:

```
rbash script1.sh
sh script1.sh
bash -x script1.sh
```

When running a script in a subshell, you should define which shell should run the script. For the purpose of this course, all scripts will start with the line

```
#!/bin/bash
```

Comments in C shell scripts begin at a pound sign

```
#!/bin/bash
clear # clear terminal window
```

# Bash script

```
#!/bin/bash
# This script clears the terminal, displays a greeting and gives information
# about currently connected users.  T

clear # clear terminal window
echo "The script starts now."

echo "Hi, $USER!" # dollar sign is used to get content of variable
echo

echo "I will now fetch you a list of connected users:"
echo
w # show who is logged on and
echo # what they are doing

echo "I'm setting two variables now."
COLOUR="black" # set a local shell variable
VALUE="9" # set a local shell variable
echo "This is a string: $COLOUR" # display content of variable
echo "And this is a number: $VALUE" # display content of variable
echo

echo "I'm giving you back your prompt now."
echo
```

# Bash script

## A very simple backup script

```
#!/bin/bash
tar -cZf /var/my-backup.tgz /home/me/
```