Advanced Computing 2 – UNIX shell and Regular Expressions

Stefano Allesina

Advanced Computing 2

- **Goal:** Learn to write pipelines for data manipulation and analysis using the `UNIX` shell. Show how to interface the `UNIX` shell and `R`. Introduce the use of regular expressions.
- **Audience:** experienced `R` users, familiar with `dplyr` and `ggplot2`.
- **Installation:** Windows users need to install a UNIX shell emulator (e.g., `Git Bash`); for regular expressions, we are going to use the `R` package `stringr`.

The UNIX Shell

`UNIX` is an operating system (i.e., the software that lets you interface with the computer) developed in the 1970s by a group of programmers at the AT&T Bell laboratories. Among them were Brian Kernighan and Dennis Ritchie, who also developed the programming language `C`. The new operating system was an immediate success in academic circles, with many scientists writing new programs to extend its features. This mix of commercial and academic interest led to the many variants of `UNIX` available today (e.g., OpenBSD, Sun Solaris, Apple OS X), collectively denoted as *nix systems. Linux is the open source `UNIX` clone whose "engine" (kernel) was written from scratch by Linus Torvalds with the assistance of a loosely-knit team of hackers from across the internet.

All *nix systems are multi-user, network-oriented, and store data as plain text files that can be exchanged between interconnected computer systems. Another characteristic is the use of a strictly hierarchical file system.

Why use UNIX?

Many biologists are not familiar with coding in *nix systems and, given that the learning curve is initially fairly steep, we start by listing the main advantages of these systems over possible alternatives.

First, `UNIX` is an operating system written by programmers for programmers. This means that it is an ideal environment for developing your code and storing your data.

Second, hundreds of small programs are available to perform simple tasks. These small programs can be strung together efficiently so that a single line of `UNIX` commands can perform complex operations, which otherwise would require writing a long and complex program. The possibility of creating these pipelines for data analysis is especially important for biologists, as modern research groups produce large and complex data sets, whose analysis requires a level of automation that would be hard to achieve otherwise. For instance, imagine working with millions of files by having to open each one of them manually to perform an identical task, or try opening your single 80Gb whole-genome sequencing file in a software with a graphical user interface! In `UNIX`, you can string a number of small programs together, each performing a simple task, and create a complex pipeline that can be stored in a script (a text file containing all the commands). Then, you can let the computer analyze all of your data while you're having a cup of coffee.

Third, text is the rule: almost anything (including the screen, the mouse, etc.) in `UNIX` is represented as a text file. Using text files means that all of your data can be read and written by any machine, and without

the need for sophisticated (and expensive) proprietary software. Text files are (and always will be) supported by any operating system and you will still be able to access your data decades from today (while this is not the case for most commercial software). The text-based nature of `UNIX` might seem unusual at first, especially if you are used to graphical interfaces and proprietary software. However, remember that `UNIX` has been around since the early 1970s and will likely be around at the end of your career. Thus, the hard work you are putting into learning `UNIX` will pay off over a lifetime.

The long history of `UNIX` means that a large body of tutorials and support web sites are readily available online. Last but not least, `UNIX` is very stable, robust, secure, and—in the case of Linux—freely available.

In the end, entirely avoiding to work in a `UNIX` "shell" is almost impossible for a professional scientist: basically all resources for High-Performance Computing (computer clusters, large workstations, etc.) run a `UNIX` or Linux operating system. Similarly, the transfer of large files, websites, and data between machines is typically accomplished through command-line interfaces.

Directory structure

In `UNIX` we speak of "directories", while in a graphical environment the term "folder" is more common. These two terms are interchangeable and refer to a structure that may contain sub-directories and files. The `UNIX` directory structure is organized hierarchically in a tree. As a biologist, you can think of this structure as a phylogenetic tree. The common ancestor of all directories is also called the "root" directory and is denoted by an individual slash (`/`). From the root directory, several important directories branch:

- `/bin` contains several basic programs.
- `/etc` contains configuration files.
- `/dev` contains the files connecting to devices such as the keyboard, mouse and screen.
- `/home` contains the home directory of each user (e.g., `/home/yourname`; in OS X, your home directory is stored in `/Users/yourname`).
- `/tmp` contains temporary files.

You will typically work in your home directory. From there, you can access the Desktop, Downloads, Documents, and other directories you are likely familiar with. When you navigate the system you are in one directory and can move deeper in the tree or upward towards the root.

Using the terminal

Terminal refers to the interface that you use to communicate with the kernel (the core of the operating system). The terminal is also called *shell*, or *command-line interface* (CLI). It processes the commands you type, translates them for the kernel, and shows you the results of your operations. There are several shells available. Here, we concentrate on the most popular one, the `bash` shell, which is the default shell in both Ubuntu and OS X.

In Ubuntu, you can open a shell by pressing `Ctrl + Alt + t` or by opening the dash (press the `Super` key) and typing `Terminal`. In OS X, you want to open the application `Terminal.app`, which is located in the folder *Utilities* within *Applications*. Alternatively, you can type `Terminal` in *Spotlight*. In either system, the shell will automatically start in your home directory. Windows users can launch `Git Bash` or another terminal emulator.

The command line prompt ends with a "dollar" (`$`) sign. This means the terminal is ready to accept your commands. Here, a `$` sign at the beginning of a line of code signals that the command has to be executed in your terminal. You do not need to type the `$` sign in your terminal, just copy the command that follows it.

In `UNIX`, you can use the `Tab` key to reduce the amount you have to type, which in turn reduces errors caused by typos. When you press `Tab` in a (properly configured) shell, it will try to automatically complete your

command, directory or file name (if multiple completions are possible, you can display them all by hitting the `Tab` key twice). Additionally, you can navigate the history of commands you typed by using the up/down arrows (you do not need to re-type a command that you recently executed). There are also shortcuts that help when dealing with long lines of code:

- `Ctrl + A` Go to the beginning of the line
- `Ctrl + E` Go to the end of the line
- `Ctrl + L` Clear the screen
- `Ctrl + U` Clear the line before the cursor position
- `Ctrl + K` Clear the line after the cursor
- `Ctrl + C` Kill the command that is currently running
- `Ctrl + D` Exit the current shell
- `Alt + F` Move cursor forward one word (in OS X, `Esc + F`)
- `Alt + B` Move cursor backward one word (in OS X, `Esc + B`)

Mastering these and other keyboard shortcuts will save you a lot of time. You may want to print this list and keep it next to your keyboard—in a while you will have them all memorized and will start using them automatically.

Basic UNIX commands

Here we introduce some of the most basic (and most useful) `UNIX` commands. We write the commands in `fixed-width font` and specific, user-provided input is capitalized in square brackets. Again, the brackets and special formatting are not required to execute a command in your terminal.

Many commands require some arguments (e.g., copy which file to where), and all can be modified using the several options available. Typically, options are either written as a dash followed by a single letter (older style, e.g., `-f`) or two dashes followed by words (newer style, e.g., `--full-name`). A command, its options and arguments are separated by a space.

How to get help in UNIX

`UNIX` ships with hundreds of commands. As such, it is impossible to remember them all, let alone all their possible options. Fortunately, each command is described in detail in its manual page, which can be accessed directly from the shell by typing `man [COMMAND OF YOUR CHOICE]` (not available in `Git Bash`). Use arrows to scroll up and down and hit `q` to close the manual page. Checking the exact behavior of a command is especially important, given that the shell will execute any command you type without asking whether you know what you're doing (such that it will promptly remove all of your files, if that's the command you typed). You may be used to more forgiving (and slightly patronizing) operating systems in which a pop-up window will warn you whenever something you're doing is considered dangerous. In `UNIX`, it is always better to consult the manual rather than improvising.

If you want to interrupt the execution of a command, press `Ctrl + C` to halt any command that is currently running in your shell.

Navigating the directory system

You can navigate the hierarchical `UNIX` directory system using these commands:

- `pwd` print the path of the current working directory.
- `ls` list the files and sub-directories in the current directory. `ls -a` list all (including hidden) files. `ls -l` return the long list with detailed information. `ls -lh` provide file sizes with units (B, M, K, etc.).}

- `cd [NAMEOFDIR]` change directory. `cd ..` move one directory up; `cd /` move to the root directory; `cd ~` move to your home directory; `cd -` go back to the directory you visited previously (like "Back"" in a browser).

Handling directories and files

Create and delete files or directories using the following commands:

- `cp [FROM] [TO]` copy a file. The first argument is the file to copy. The second argument is where to copy it (either a directory or a file name).
- `mv [FROM] [TO]` move or rename a file. Move a file by specifying two arguments: the file, and the destination directory. Rename a file by specifying the old and the new file name in the same directory.
- `touch [FILENAME]` Update the date of last access to the file. Interestingly, if the file does not exist, this command will create an empty file.
- `rm [TOREMOVE]` remove a file. `rm -r` deletes the contents of a directory recursively (i.e., including all files and sub-directories in it; use with caution!). Similarly, `rm -f` removes the file and suppresses any prompt asking whether you are sure you want to remove the file.
- `mkdir [DIRECTORY]` make a directory. To create nested directories, use the option `-p` (e.g., `mkdir -p d1/d2/d3`).
- `rmdir [DIRECTORY]` remove an empty directory.

Printing and modulating files

`UNIX` was especially designed to handle text files, which is apparent when considering the multitude of commands dealing with text. Here are a few popular ones:

- `less [FILENAME]` progressively print a file on the screen (press `q` to exit). Funny fact: there is a command called `more` that does the same thing, but with less flexibility. Clearly, in `UNIX`, `less` is `more`.
- `cat [FILENAME]` concatenate and print files.
- `wc [FILENAME]` line, word, and byte (character) count of a file.
- `sort [FILENAME]` sort the lines of a file and print the result to the screen.
- `uniq [FILENAME]` show only unique lines of a file. The file needs to be sorted first for this to work properly.
- `file [FILENAME]` determine the type of a file.
- `head [FILENAME]` print the `head` (i.e., first few lines of a file).
- `tail [FILENAME]` print the tail (i.e., last few lines of a file).
- `diff [FILE1] [FILE2]` show the differences between two files.

  Exercise To familiarize yourself with these commands, try the following:

  - Go to the `data` directory for this tutorial.
  - How many lines are in file `Marra2014_data.fasta`?
  - Go back to the `code` directory.
  - Create the empty file `toremove.txt`.
  - List the content of the directory.
  - Remove the file `toremove.txt`.

Miscellaneous commands

- `echo "[A STRING]"` print the string `[A STRING]`.
- `time` time the execution of a command.

- `wget [URL]` download the webpage at `[URL]`. (Available in Ubuntu; for OS X look at `curl`, or install `wget`).
- `history` list the last commands you executed.

Advanced UNIX commands

Redirection and pipes

So far, we have printed the output of each command (e.g., `ls`) directly to the screen. However, it is easy to direct the output to a file (*redirect*) or use it as the input of another command (*pipe*). Stringing commands together in pipes is the real power of `UNIX`—the ability to perform complex processing of large amounts of data in a single line of commands. First, we show how to redirect the output of a command into a file:

```
$ [COMMAND] > [FILENAME]
```

Note that if the file `[FILENAME]` exists, it will be overwritten. If instead we want to append to an existing file, we can use the `>>` symbol as in the following line:

```
$ [COMMAND] >> [FILENAME]
```

When the command is very long and complex, we might want to redirect the content of a file as input to a command, "reversing" the flow:

```
$ [COMMAND] < [FILENAME]
```

To run a few examples, let's start by moving to our `code` directory:

```
$ cd ~/BSD-QBio2/tutorials/advanced_computing2/code
```

The command `echo` can be used to print a string on the screen. Instead of printing to the screen, we redirect the output to a file, effectively creating a file containing the string we want to print:

```
$ echo "My first line" > test.txt
```

We can see the result of our operation by printing the file to the screen using the command `cat`:

```
$ cat test.txt
```

To append a second line to the file, we use `>>`:

```
$ echo "My second line" >> test.txt
$ cat test.txt
```

We can redirect the output of any command to a file. For example, it is quite common to have to determine how many files are in a directory. The files could have been created by an instrument or provided by a collaborator. Before analyzing the data, we want to get a sense of how many files will we need to process. If there are thousands of files, it is quite time consuming to count them by hand or even open a file browser that can do the counting for us. It is much simpler and faster to just type a command or two. To try this, let's create a file listing all the files contained in `data/Saavedra2013`:

```
$ ls ../data/Saavedra2013 >> filelist.txt
$ cat filelist.txt
```

Now we want to count how many lines are in the file. We can do so by calling the command `wc -l` (count only the lines):

```
$ wc -l filelist.txt
$ rm filelist.txt
```

However, we can skip the creation of the file by creating a short pipeline. The pipe symbol `|` tells the shell to take the output on the left of the pipe and use it as the input of the command on the right of the pipe. To take the output of the command `ls` and use it as the input of the command `wc` we can write:

```
$ ls  ../data/Saavedra2013 | wc -l
```

We have created our first, simple pipeline. In the following sections, we are going to build increasingly long and complex pipelines. The idea is always to start with a command and progressively add one piece after another to the pipeline, each time checking that the result is the desired one.

Selecting columns using `cut`

When dealing with tabular data, you will often encounter the Comma Separated Values (CSV) Standard File Format. The CSV format is platform and software independent, making it the standard output format of many experimental devices. The versatility of the file format should also make it your preferred choice when manually entering and storing data.

The main `UNIX` command you want to master for comma-, space-, tab-, or character-delimited text files is `cut`. To showcase its features, we work with data on generation time of mammals published by Pacifici *et al.*. First, let's make sure we are in the right directory (`advanced_computing2/data`). Then, we can print the header (the first line, specifying the content of each column) of the CSV file using the command `head`, which prints the first few lines of a file on the screen, with the option `-n 1`, specifying that we want to output only the first line:

```
$ head -n 1 Pacifici2013_data.csv
TaxID;Order;Family;Genus;Scientific_name;...
```

We now pipe the header to `cut`, specify the character to be used as delimiter (`-d ';'`), and use the `head` command to extract the name of the first column (`-f 1`), or the names of the first four columns (`-f 1-4`):

```
$ head -n 1 Pacifici2013_data.csv | cut -d ';' -f 1
TaxID

$ head -n 1 Pacifici2013_data.csv | cut -d ';' -f 1-4
TaxID;Order;Family;Genus
```

Remember to use the `Tab` key to auto-complete file names and the arrow keys to access your command history.

In the next example, we work with the file content. We specify a delimiter, extract specific columns, and pipe the result to the `head` command—to display only the first few elements:

```
$ cut -d ';' -f 2 Pacifici2013_data.csv  | head -n 5
Order
Rodentia
Rodentia
Rodentia
Macroscelidea

$ cut -d ';' -f 2,8 Pacifici2013_data.csv  | head -n 3
Order;Max_longevity_d
Rodentia;292
Rodentia;456.25
```

Now, we specify the delimiter, extract the second column, skip the first line (the header) using the `tail -n +2` command (i.e., return the whole file starting from the second line), and finally display the first five entries:

```
$ cut -d ';' -f 2 Pacifici2013_data.csv  | tail -n +2 | head -n 5
Rodentia
Rodentia
Rodentia
Macroscelidea
Rodentia
```

We pipe the result of the previous command to the `sort` command (which sorts the lines), and then again to `uniq`, (which takes only the elements that are not repeated). Effectively, we have created a pipeline to extract the names of all the Orders in the database, from Afrosoricida to Tubulidentata (a remarkable Order, which today contains only the aardvark).

```
$ cut -d ';' -f 2 Pacifici2013_data.csv  | tail -n +2 | sort | uniq
Afrosoricida
Carnivora
Cetartiodactyla
...
```

This type of manipulation of character-delimited files is very fast and effective. It is an excellent idea to master the `cut` command in order to start exploring large data sets without the need to open files in specialized programs (if you don't want to modify the content of a file, you should not open it in an editor!).

Exercise:

- If we order all species names (fifth column) of `Pacifici2013_data.csv` in alphabetical order, which is the first species? Which the last?
- How many families are represented in the database?

Substituting characters using `tr`

We often want to substitute or remove a specific character in a text file (e.g., to convert a comma-separated file into a tab-separated file). Such a one-by-one substitution can be accomplished with the command `tr`. Let's look at some examples in which we use a pipe to pass a string to `tr`, which then processes the text input according to the search term and specific options.

Substitute all characters `a` with `b`:

```
$ echo 'aaaabbb' | tr 'a' 'b'
bbbbbbb
```

Substitute every number in the range `1` through `5` with `0`:

```
$ echo '123456789' | tr 1-5 0
000006789
```

Substitute lower-case letters with upper-case (note the one-to-one mapping):

```
$ echo 'ACtGGcAaTT' | tr actg ACTG
ACTGGCAATT
```

We achieve the same result using bracket expressions that provide a predefined set of characters. Here, we use the set of all lower-case letters `[:lower:]` and translate into upper-case letters `[:upper:]`:

```
$ eecho 'ACtGGcAaTT' | tr [:lower:] [:upper:]
ACTGGCAATT
```

We can also indicate ranges of characters to substitute:

```
$ echo 'aabbccddee' | tr a-c 1-3
112233ddee
```

Delete all occurrences of `a`:

```
$ echo 'aaaaabbbb' | tr -d a
bbbb
```

"Squeeze" all consecutive occurrences of `a`:

```
$ echo 'aaaaabbbb' | tr -s a
abbbb
```

Note that the command `tr` reads standard input, and does not operate on files directly. However, we can use pipes in conjunction with `cat`, `head`, `cut`, etc. to create input for `tr`:

```
$  tr ' ' '\t' < [INPUTFILE] > [OUTPUTFILE]
```

In this example we input `[INPUTFILE]` to the `tr` command to replace all spaces with tabs. Note the use of quotes to specify the space character. The tab is indicated by `\t` and is called a "meta-character". We use the backslash to signal that the following character should not be interpreted literally, but rather is a special code referring to a character that is difficult to represent otherwise.

Now we can apply the command `tr` and the commands we have showcased earlier to create a new file containing a subset of the data contained in `Pacifici2013_data.csv`, which we are going to use in the next section.

First, we change directory to the `code`:

```
$ cd ../code/
```

Now, we want to create a version of `Pacifici2013_data.csv` containing only the `Order`, `Family`, `Genus`, `Scientific_name`, and `AdultBodyMass_g` (columns 2-6). Moreover, we want to remove the header, sort the lines according to body mass (with larger critters first), and have the values separated by spaces. This sounds like an awful lot of work, but we're going to see how this can be accomplished piping a few commands together.

First, let's remove the header:

```
$ tail -n +2 ../data/Pacifici2013_data.csv
```

Then, take only the columns 2-6:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6
```

Now, substitute the current delimiter (`;`) with a space:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6 | tr -s ';' ' '
```

To sort the lines according to body size, we need to exploit a few of the options for the command `sort`. First, we want to sort numbers (option `-n`); second, we want larger values first (option `-r`, reverse order); finally, we want to sort the data according to the sixth column (option `-k 6`):

```
$ tail -n +2 ../data/Pacifici2013_data.csv |cut -d ';' -f 2-6 | tr -s ';' ' ' | sort -n -r -k 6
```

That's it. We have created our first complex pipeline. To complete the task, we redirect the output of our pipeline to a new file called `BodyM.csv`.

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6 | tr -s
  ';' ' ' | sort -r -n -k 6 > BodyM.csv
```

You might object that the same operations could have been accomplished with a few clicks by opening the file in a spreadsheet editor. However, suppose you have to repeat this task many times, e.g., to reformat every file that is produced by a laboratory device. Then it is convenient to automate this task such that it can be run with a single command.

Similarly, suppose you need to download a large CSV file from a server, but many of the columns are not needed. With `cut`, you can extract only the relevant columns, reducing download time and storage.

Selecting lines using `grep`

`grep` is a powerful command that finds all the lines of a file that match a given pattern. You can return or count all occurrences of the pattern in a large text file without ever opening it. `grep` is based on the concept of regular expressions, which we will cover just below (but using `R`, in which the syntax is slightly different).

We will test the basic features of `grep` using the file we just created. The file contains data on thousands of species:

```
$ wc -l BodyM.csv
5426 BodyM.csv
```

Let's see how many wombats (family Vombatidae) are contained in the data. First we display the lines that contain the term "Vombatidae":

```
$ grep Vombatidae BodyM.csv
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus krefftii 31849.99
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus latifrons 26163.8
Diprotodontia Vombatidae Vombatus Vombatus ursinus 26000
```

Now we add the option `-c` to count the lines:

```
$ grep -c Vombatidae BodyM.csv
3
```

Next, we have a look at the genus `Bos` in the data file:

```
$ grep Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
Cetartiodactyla Bovidae Boselaphus Boselaphus tragocamelus 182253
```

Besides all the members of the `Bos` genus, we also match one member of the genus `Boselaphus`. To exclude it, we can use the option `-w`, which prompts `grep` to match only full words:

```
$ grep -w Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
```

Using the option `-i` we can make the search case-insensitive (it will match both upper- and lower-case instances):

```
$ grep -i Bos BodyM.csv
Proboscidea Elephantidae Loxodonta Loxodonta africana 3824540
Proboscidea Elephantidae Elephas Elephas maximus 3269794
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
...
```

Sometimes, we want to know which lines precede or follow the one we want to match. For example, suppose we want to know which mammals have body weight most similar to the gorilla (*Gorilla gorilla*). The species are already ordered by size, thus we can simply print the two lines before the match using the option `-B 2` and the two lines after the match using `-A 2`:

```
$ grep -B 2 -A 2 "Gorilla gorilla" BodyM.csv
Cetartiodactyla Bovidae Ovis Ovis ammon 113998.7
Cetartiodactyla Delphinidae Lissodelphis Lissodelphis borealis 113000
Primates Hominidae Gorilla Gorilla gorilla 112589
Cetartiodactyla Cervidae Blastocerus Blastocerus dichotomus 112518.5
Cetartiodactyla Iniidae Lipotes Lipotes vexillifer 112138.3
```

Use option `-n` to show the line number of the match. For example, the gorilla is the 164th largest mammal in the database:

```
$ grep -n "Gorilla gorilla" BodyM.csv
164:Primates Hominidae Gorilla Gorilla gorilla 112589
```

To print all the lines that do not match a given pattern, use the option `-v`. For example, to get the other species of the genus *Gorilla* with the exception of *Gorilla gorilla*, we can use:

```
$ grep Gorilla BodyM.csv | grep -v gorilla
Primates Hominidae Gorilla Gorilla beringei 149325.2
```

To match one of several strings, use `grep "[STRING1]\|[STRING2]"`

```
$ grep -w "Gorilla\|Pan" BodyM.csv
Primates Hominidae Gorilla Gorilla beringei 149325.2
Primates Hominidae Gorilla Gorilla gorilla 112589
Primates Hominidae Pan Pan troglodytes 45000
Primates Hominidae Pan Pan paniscus 35119.95
```

You can use `grep` on multiple files at a time! Simply, list all the files to use instead of just one file.

Interfacing `R` and the `UNIX` shell

Note: this will work if you're using Mac OSX or UNIX; in Windows options are much more limited.

Calling `R` from the command line

In `R`, you typically work in "interactive" mode — you type a command, it gets executed, you type another command, and so on. Often, we want to be able to re-run a script on different data sets or with different parameters. For that purpose you can store all the commands in a text file (typically, with extension `.R`), and then re-run the analysis by typing in the `UNIX` command line

```
$ Rscript my_script_file.R
```

To properly automate our analysis and figure generation, however, we can additionally pass command-line arguments to `R`. This allows us for instance to perform the analysis using a specific input file, or save the figure using a specific file name.

`Rscript` accepts command-line arguments, that need to be parsed within `R`. The code at the beginning of the following script shows how this is accomplished:

```r
# Get all the command-line arguments
args <- commandArgs(TRUE)
# Assign each argument to a variable,
# making sure to convert it to the right
# type of variable (string by default)

# check the number of arguments
num.args <- length(args)
print(paste("Number of command-line arguments:", num.args))
# print all the arguments
if (num.args > 0) {
```

```
    for (i in 1:num.args) {
        print(paste(i, "->", args[i]))
    }
}

# We can initially set to default values
# (but pay attention to the order,
# the optional arguments should be at the end)
input.file <- "test.txt"
number.replicates <- 10
starting.point <- 3.14

if (num.args >= 1) {
    input.file <- args[1]
}
if (num.args >= 2) {
    number.replicates <- as.integer(args[2])
}
if (num.args >= 3) {
    starting.point <- as.double(args[3])
}

print(c(input.file, number.replicates, starting.point))

# Save this script as my_script.R
# Run the script in bash with different arguments
# Rscript my_script.R abc.txt 5 100.0
# Rscript my_script.R abc.txt 5
# Rscript my_script.R abc.txt
# Rscript my_script.R
```

Calling the command line from R

You can call the operating system from within R (assuming you're in /advanced_computing_2/code):

```
system("wc -l < ../../basic_computing_1/data/H938_Euro_chr6.geno")
```

You can also capture the output from the shell commands and save it into R. Everything is treated as text (convert to numeric if necessary):

```
numlines <- system("wc -l < ../../basic_computing_1/data/H938_Euro_chr6.geno",
                   intern = TRUE)
numlines
```

```
## [1] "43142"
```

You can also use a combination of shell commands and read.table to capture more complex output:

```
mydf <- system("grep rs125283 ../../basic_computing_1/data/H938_Euro_chr6.geno",
                     intern = TRUE)
mydf <- read.table(file = textConnection(mydf))
mydf
```

```
##   V1         V2 V3 V4 V5 V6  V7
## 1  6 rs12528302  G  A 26 59  39
## 2  6 rs12528322  G  A  0 21 103
## 3  6 rs12528313  G  T  1 25  98
## 4  6 rs12528341  C  T  3 31  90
```

Regular expressions in R

Sometimes data is hidden in free text. Think of citations in a manuscript, mentions of DNA motifs in tables, etc. You could copy and paste data from these unstructured texts yourself, but if you have much text, the task is very boring and error-prone. What you need is a way to describe a text pattern to a computer, and then have it extract the data automatically. Regular Expressions do exactly that.

Because you want to describe a text pattern using text, a level of abstraction is inevitable. What you want to do is to construct a pattern using **literal** characters and **metacharacters**.

For all our examples, we will use the package `stringr`, which makes the regular expression syntax consistent (there are many *dialects*), and provides a set of easy-to-use functions:

```
library(stringr)
```

All the functions have a common structure. For example, `str_extract` extracts text matching a pattern: `str_extract(text, pattern)`. The simplest possible expression is one in which the pattern is described literally (i.e., we want to find exactly the text we're typing):

```
str_extract("a string of text", "t")
```

```
## [1] "t"
```

```
str_extract_all("a string of text", "t")
```

```
## [[1]]
## [1] "t" "t" "t"
```

Of course, you need to be able to describe much more general patterns. Use the following metacharacters:

- `\d` Match a digit character (`0-9`)
- `\D` Match any character that is not a digit
- `\n` Match a newline
- `\s` Match a space
- `\t` Match a `Tab`
- `\b` Match a "word boundary"
- `\w` Match a "word" character (alphanumeric)
- `.` Match any character

Some examples (note that to escape characters, you want to use two backslashes — you need to escape the backslash itself!):

```
# find the first digit
str_extract("123.25 grams", "\\d")
```

```
## [1] "1"
```

```
# find word separator + word character + word separator
str_extract("Albert Einstein was a genius", "\\b\\w\\b")
```

```
## [1] "a"
```

```
# find all digits
str_extract_all("my cell is 773 345 6789", "\\d")
```

```
## [[1]]
##  [1] "7" "7" "3" "3" "4" "5" "6" "7" "8" "9"
```

```
# extract all characters
str_extract_all("for example, this and that", ".")
```

```
## [[1]]
##  [1] "f" "o" "r" " " "e" "x" "a" "m" "p" "l" "e" "," " " "t" "h" "i" "s"
## [18] " " "a" "n" "d" " " "t" "h" "a" "t"
```

Of course, you don't want to type `\\w` fifteen times, in case you are looking for a string that is 15 characters long! Rather, you can use quantifiers:

- `*` Match zero or more times. Match as many times as possible.
- `*?` Match zero or more times. Match as few times as possible.
- `+` Match one or more times. Match as many times as possible.
- `+?` Match one or more times. Match as few times as possible.
- `?` Match zero or one times. In case both zero and one time match, prefer one.
- `??` Match zero or one times, prefer zero.
- `{n}` Match exactly `n` times.
- `{n,}` Match at least `n` times. Match as many times as possible.
- `{n,m}` Match between `n` and `m` times.

Exercise

What does this do? Try to guess, and then type the command into `R`

```
str_extract_all("12.06+3.21i", "\\d+\\.?\\d+")
my_str <- "most beautiful and most wonderful have been, and are being, evolved."
str_extract(my_str, "\\b\\w{6,10}\\b")
str_extract(my_str, "\\b\\w+\\b")
str_extract(my_str, "w\\w*")
str_extract(my_str, "b\\w*")
str_extract(my_str, "b\\w+?")
str_extract(my_str, "\\s\\wn\\w+")
```

What if you want to match the characters `?`, `+`, `*`, `.`? You will need to escape them: for example, `\\.` matches the "dot" character.

You can specify anchors to signal that the match has to be in certain special positions in the text:

- `^` Match at the beginning of a line.
- `$` Match at the end of a line.

```r
str_extract("Ah, ba ba ba ba Barbara Ann", "\\w{2,}$")
```

```
## [1] "Ann"
```

```r
str_extract("Ah, ba ba ba ba Barbara Ann", "^\\w{2,}")
```

```
## [1] "Ah"
```

To match one of several characters, list them between brackets:

```r
str_extract("01234567890", "[3120]+")
```

```
## [1] "0123"
```

```r
str_extract("01234567890", "[3-5]+")
```

```
## [1] "345"
```

```r
str_extract("supercalifragilisticexpialidocious", "[a-i]{3,}")
```

```
## [1] "agi"
```

To match either of two patterns, use alternations:

```r
str_extract_all("The quick brown fox jumps over the lazy dog", "fox|dog")
```

```
## [[1]]
## [1] "fox" "dog"
```

If you need more complex alternations, use parentheses to separate the patterns.

Parentheses can also be used to define **groups**, which are used when you want to capture unknown text that is however flanked by known patterns. For example, suppose you want to save the user name of a UofC email:

```r
str_match_all("sallesina@uchicago.edu mjsmith@uchicago.edu",
              "\\b([a-zA-Z0-9]*?)@uchicago.edu")
```

```
## [[1]]
##      [,1]                    [,2]
## [1,] "sallesina@uchicago.edu" "sallesina"
## [2,] "mjsmith@uchicago.edu"   "mjsmith"
```

Note that you have to use `str_match` or `str_match_all` to obtain information on the groups.

Useful functions

Many functions have a similar `str_***_all` version, returning all matches.

- `str_detect(strings, pattern)` do the `strings` contain the `pattern`? Returns a logical vector.
- `str_locate(strings, pattern)` find the character position of the pattern
- `str_extract(strings, pattern)` extracts the first match
- `str_match(strings, pattern)` like `extract` but capture groups defined by parentheses
- `str_replace(strings, pattern, newstring)` replaces the first matched `pattern` with `newstring`

Exercises in groups

Extract primers and polymorphic sites

In the file `data/Ptak_etal_2004.txt` you find a text version of the supplementary materials of Ptak *et al.* (PLoS Biology 2004, doi:10.1371/journal.pbio.0020155).

- Take a look at the file. You see that it first lists the primers used for the study (e.g., `TAP2-17-3' -> CTTGGATATAACACCAAACGCA`), and then the polymorphic sites for 24 chimpanzees (e.g., .
- Read the text as a single string, intervaled by new lines

```
my_txt <- paste(readLines("../data/Ptak_etal_2004.txt"), collapse="\n")
```

- Use regular expressions to produce a data frame containing the primers used in the study:

```
head(primers)
```

```
         ID               Sequence
1 TAP2-1-5' GAGAATCACTTGAACCTGGGAG
2 TAP2-2-5' TTGTCCACAGTGTACCACATGA
3 TAP2-3-5' TATTTCTTCCTGGGGTTTCCTT
4 TAP2-4-5' CATGATGTGTCATGCTGAATTG
5 TAP2-5-5' ATAGAACAAGAACCAAAGCCCA
6 TAP2-6-5' GGACAACAGATAAAGTTGCCCT
```

- Write another regular expression to extract the polymorphic region for each chimp. Use `str_replace_all` to remove extra spaces and newlines. The results should look like:

```
  Chimp                                                Sequence
1   311 ATACCCTGGAGGCAAGAATCTTCCGATAGACGCCAGTCCCTAGTTGT...
2   312 GCACCCTGGAGGCAAGAGTCTTCCGATAGACGCCAGTCCCTAGTTGC...
3   313 GCACCCTGGAGGCAAGAGTCTTCCGATAGACGCCAGTCCCCAGTTGT...
4   314 GCACCCTGGAGGCAAGAGTCTTCCGATAGACCCCAGTCCCCAGTTGC...
5   317 GCACCCTGGAGGTGGGGGCCCCCCAGGAGGCCCCAGCCCCCGGTCGT...
6   320 GCACCCTGGAGATAAGGGCCCCCCAGGAGGCCCCAGCCCCCGGTCGT...
```

A map of *Science*

Where does science come from? This question has fascinated researchers for decades and even led to the birth of the field of "Science of Science", where researchers use the same tools they invented to investigate nature to gain insights on the development of science itself. In this exercise, you will build a map of *Science*, showing where articles published in *Science* magazine have originated. You will find two files in the directory `data/MapOfScience`. The first, `pubmed_results.txt`, is the output of a query to PubMed listing all the papers published in Science in 2015. You will extract the US ZIP codes from this file, and then use the file `zipcodes_coordinates.txt` to extract the geographic coordinates for each ZIP code.

- Read the file `pubmed_results.txt`, and extract all the US ZIP codes.
- Count the number of occurrences of each ZIP code using `dplyr`.
- Join the table you've created with the data in `zipcodes_coordinates.txt`
- Plot the results using `ggplot2` (either use points with different colors/alphas, or render the density in two dimensions)